Univeristé d'Évry Val d'Essonne

M1 GENIOMHE Mention Bioinformatique

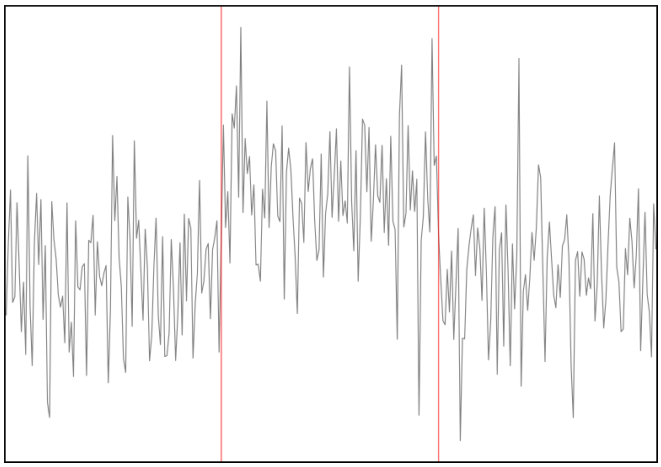
(2018/2019)

Développement d'un algorithme exact de segmentation pour des données génomiques et application à l'analyse du nombre de copies d'ADN

Auteur :
Liehrmann Arnaud

Maître de stage :
Rigaill Guillem

Tuteur Universitaire :
Ambroise Christophe



3 août 2019

Résumé

Dans ce rapport je présente une extension de ***FPOP***, un algorithme exact de segmentation de données s'appuyant sur de l'élagage fonctionnel. Cette extension implémente une pénalité dépendante de la taille des segments. Plusieurs arguments statistiques et biologiques suggèrent que cette pénalité permet une meilleure segmentation des données que ***FPOP***. Les résultats de mes simulations sur des profils avec un bruit Gaussien ou un bruit réaliste vérifient ces suggestions. J'ai participé à la conception de ***FpopPSD*** et je propose une implémentation efficace en C++ de cette méthode permettant de traiter de grands profils rapidement.

Summary

In this report I present an extension of ***FPOP***, an exact data segmentation algorithm based on functional pruning. This extension implements a penalty depending on the size of the segments. Several statistical and biological arguments suggest that this penalty allows a better segmentation of the data than ***FPOP***. The results of my simulations on profiles with Gaussian noise or realistic noise verify these suggestions. I participated in the design of ***FpopPSD*** and I propose an effective implementation in C ++ of this method allowing to process large profiles quickly.

Table des matières

1 Introduction **2**

1.1 Contexte scientifique 2

1.2 Définition du modèle statistique de détection de ruptures multiples 3

1.3 Définition du problème d'optimisation pénalisée 4

2 Résolution par programmation dynamique **5**

2.1 Optimal partitioning (*OP*) 5

2.2 Functional Pruning Optimal Partioning (*FPOP*) 5

2.3 Extension de *FPOP* au cas d'une pénalité dépendante de la taille des segments (*FpopPSD*) 7

3 Implémentation de la méthode *FpopPSD* **10**

3.1 Généralités 10

3.2 Présentation des classes 11

3.3 Débogage 15

4 Résultats **16**

4.1 Comparaison des stratégies d'échantillonnage 16

4.2 Calibrage de γ et comparaison de FPOP/FpopPSD sur données simulées 17

4.3 Calibrage de γ et comparaison de FPOP/FpopPSD sur données réalistes 17

5 Discussion **18**

6 Conclusion **19**

7 Références **20**

8 Annexes **21**

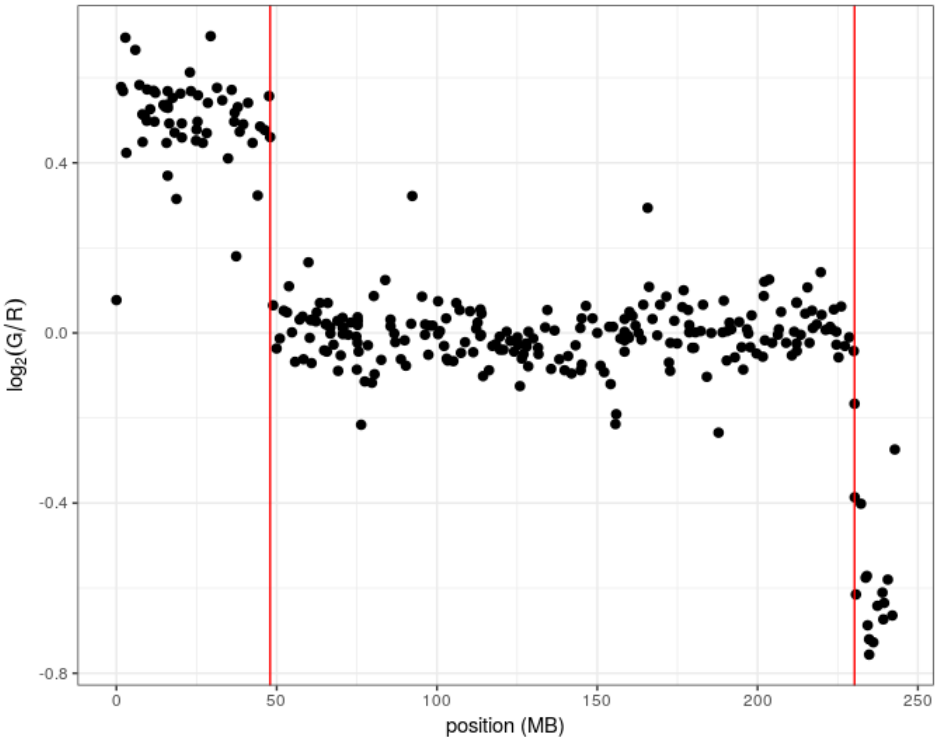


FIGURE 1 – Représentation du log_2 ratio des intensités normalisées de fluorescence le long du chromosome 2 issu des cellules du neuroblastome d'un patient de l'Institut Curie. Ces données ont été obtenues grâce à la technologie CGH array et sont disponibles dans le package R *"neuroblastoma"* via le profil 102. Le log_2 ratio est positif dans la région 5' du chromosome 2 ce qui indique un gain de matériel génétique dans cette région chez les cellules tumorales par rapport à la condition normale. Inversement log_2 ratio est négatif dans la région 3' du chromosome 2 ce qui indique une perte de matériel génétique dans cette région chez les cellules tumorales par rapport à la condition normale. Entre les positions 50Mb et 230Mb le ratio ne semble pas significativement différent de 0, il n'y pas de variation structurale déséquilibrée dans cette région entre les deux conditions. Les droites verticales rouges correspondent aux localisations des ruptures dans les données. En ces points nous constatons un changement abrupt dans la moyenne du signal.

1 Introduction

1.1 Contexte scientifique

L'ADN du génome humain est soumis à une variété d'altérations de différents types. Ces altérations contribuent de manière significative aux différences phénotypiques rencontrées au sein des populations humaines. Outre les polymorphismes nucléotidiques (SNP), ces changements génétiques incluent également les variations structurales chromosomiques, telles que les insertions, les délétions, les duplications, les inversions et les translocations, à différentes échelles génomiques. Des études récentes ont montré que les insertions, les délétions et les duplications de segments d'ADN de 1 kb ou plus dans le génome, souvent appelées variants de nombre de copies (CNV), se produisent à une fréquence beaucoup plus élevée que prévue [1, 2, 3]. Il est maintenant largement admis que les CNV sont aussi importants que les SNPs dans leur contribution à la variation phénotypique dans les populations humaines. En oncologie, un domaine où les CNV sont particulièrement étudiés, il est bien connu que des gains et des pertes de parties ou de chromosomes entiers [4] peuvent être observés dans les cellules tumorales. Ces modifications peuvent affecter directement ou indirectement l'expression des gènes. Plusieurs oncogènes peuvent par exemple être dupliqués et aggraver la prolifération des cellules tumorales.

Actuellement, les variants structuraux déséquilibrés peuvent être identifiés expérimentalement par des méthodes basées sur la technologie des micropuces à ADN ou les technologies de séquençage de nouvelle génération (NGS) [5]. La puce d'hybridation génomique comparative (CGH array) est une extension naturelle de l'analyse par hybridation génomique comparative (CGH), qui a été développée à l'origine pour révéler les pertes d'allèles ou les aneuploïdies par microscopie à fluorescence [6]. La technologie CGH array offre une résolution génomique plus élevée que le CGH classique, 1 à 3 Mb contre 5 à 10 Mb respectivement, et est moins coûteuse que les NGS. Aujourd'hui les analyses combinant CGH array et SNP array (une autre technologie de micropuce à ADN permettant de mettre en évidence les pertes d'hétérozygotie dont la densité des sondes varie entre entre 1kb à 2kb) sont fréquentes dans le cadre des études sur les instabilités chromosomiques dans les cellules tumorales [7, 8, 9].

Les technologies de CGH array, SNP array et NGS, via l'information sur la profondeur de couverture pour cette dernière, génèrent toutes les trois des données sur le nombre de copies génomiques dans un format très similaire : un signal indexé par des positions génomiques. Dans le cas de la technologie CGH array, le signal correspond au log-ratio des intensités normalisées de fluorescence. Pour la technologie SNP array, le signal correspond au log R ratio comme défini par Ichiro Nakachi et al. [10]. Pour les NGS, les signal correspond au log-ratio du comptage normalisé des reads. Les objectifs de l'analyse de telles données sont multiples, par exemple, détecter les CNV en identifiant les régions avec un signal toujours supérieurs ou inférieurs à la normale, évaluer la stabilité du génome en estimant le nombre de remaniements, localiser précisément les ruptures dans les gènes. Les modèles de détection de ruptures multiples ou de segmentation semblent naturels pour traiter ce genre de signaux [11, 12]. Ces modèles visent à localiser des points où les propriétés statistiques, notamment la moyenne du signal, diffèrent avant et après. Ces points sont appelés ruptures *(cf. figure 1)*.

La recherche de CNV n'est pas la seule application de la segmentation en génomique. En effet cette dernière est aussi utilisée pour localiser des blocs de liaisons dans les données Hi-C [13], pour améliorer la localisation des sites d'interaction ADN/protéines via des données

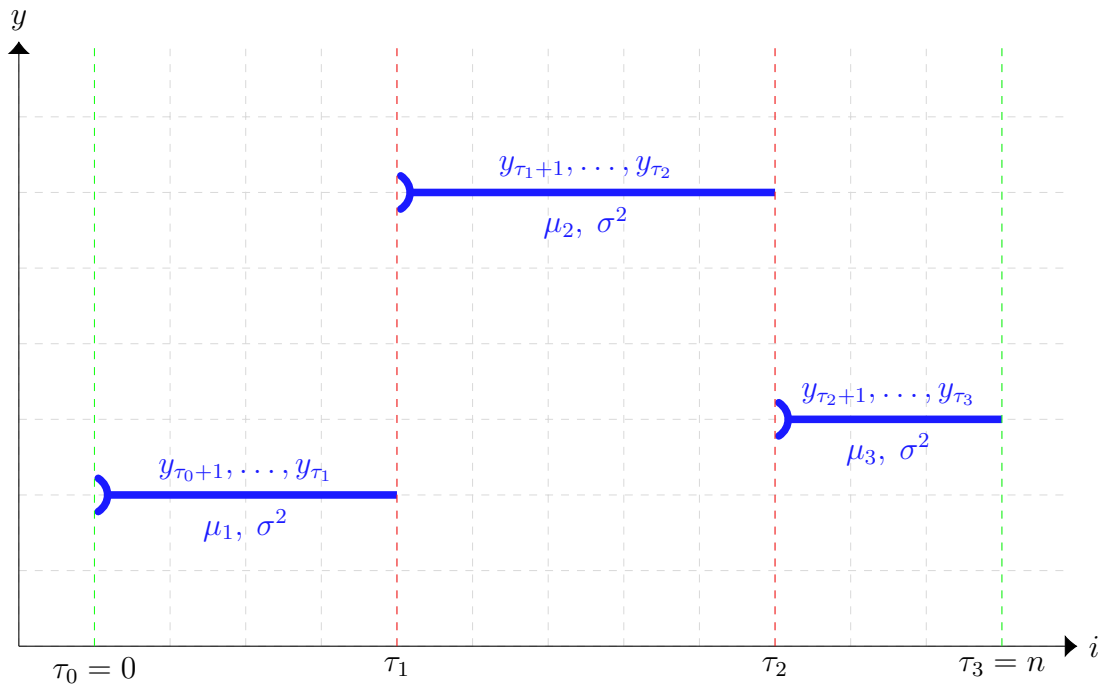


FIGURE 2 – Représentation graphique du modèle statistique. Les droites verticales rouges correspondent aux ruptures. Les droites verticales vertes correspondent aux indices factices. Les droites horizontales bleues correspondent à la moyenne μ_j des points formant le $j^{ème}$ segment.

Chip-Seq [14] ou encore identifier les sites alternatifs 3' et 5' d'épissages via des données RNA-Seq [15]. Une application, envisagée dans la suite de mon stage, est d'utiliser la segmentation sur des profils RNA-seq chloroplastique.

Plusieurs méthodes de segmentation ont déjà été appliquées à la recherche de CNV avec des résultats disparates comme en témoigne l'étude de Hocking et al. 2013 [16]. Dans cette étude, Hocking et al. ont comparé 17 modèles de segmentation disponibles dans la littérature en déterminant pour chacun le paramètre de lissage le plus favorable. Pour ce faire, ils ont utilisé 575 profils de variation du nombre de copies d'ADN provenant de neuroblastomes. Les 575 profils ont été générés via des expériences de CGH array et ont dans un deuxième temps été annotés par des experts (biologistes et bioinformaticiens). Via un protocole de cross validation, ils ont estimé pour ces 17 modèles les performances de détection des ruptures. Dans les résultats de cette étude, ***FPOP*** (Functional Pruning Optimal Partitioning) [17] et ***PELT*** (Pruned Exact Linear Time) [18], les deux méthodes avec les meilleures performances, ont une erreur moyenne par profil analysé de 2.2, où le nombre d'erreurs est égal à la somme des faux positifs et des faux négatifs. Ces derniers sont calculés en comparant le nombre de ruptures estimées par la méthode dans chaque région au nombre de ruptures dont l'annotation est expertisée par des biologistes et des bioinformaticiens dans ces mêmes régions. Le taux de faux positifs et de faux négatifs, qui sont les mêmes pour les deux méthodes [1], sont respectivement de 0.6% et 11.6%. Dans leur conclusion les auteurs insistent sur le fait que même pour les meilleurs modèles avec les paramètres de lissage les plus favorables, 11.6% des ruptures ne sont pas détectées par ces derniers. Le taux d'erreur est encore assez élevé pour compliquer l'interprétation biologique des profils.

L'objectif de mon stage est donc d'implémenter une extension de ***FPOP*** qui est une méthode de segmentation maximisant la vraisemblance pénalisée en s'appuyant sur un algorithme exact de programmation dynamique et de l'élagage fonctionnel. Cette extension implémente une pénalité qui dépend de la taille des segments. Intuitivement, cette pénalité permet de défavoriser les petits segments. Des arguments statistiques, biologiques ainsi que des simulations que j'ai réalisées, laissent penser que cette extension conduit à une meilleure segmentation des données. L'ensemble de ces points sont détaillés plus tard dans ce rapport. Dans la prochaine section, je présenterai un modèle statistique classiquement utilisé dans le cadre de la détection des ruptures multiples [12].

1.2 Définition du modèle statistique de détection de ruptures multiples

On note $Y = (y_1, \ldots, y_n)$ une série de n observations ordonnées selon un attribut. Dans notre cas, chaque observation correspond à un signal relatif au nombre de copies associées à la sonde i. On suppose que ces observations sont des réalisations de variables indépendantes suivant une loi normale de variance constante σ^2. On assume que la moyenne de ces lois est affectée par K ruptures, ce qui correspond à une segmentation du signal en $K+1$ segments. On note τ_j la localisation de la $j^{ème}$ rupture pour $j = 1, \ldots, K$. Par convention on introduit les indices factices $\tau_0 = 0$ et $\tau_{K+1} = n$. Le $j^{ème}$ segment est formé par les observations $y_{\tau_{j-1}+1}, \ldots, y_{\tau_j}$. La moyenne de ces observations notée μ_j est un paramètre spécifique au $j^{ème}$ segment. Formellement on suppose donc le modèle suivant [19] *(cf. figure 2)* :

$$\forall i \quad | \quad \tau_{j-1} + 1 \leq i \leq \tau_j, \quad Y_i \sim \mathcal{N}(\mu_j, \sigma^2) \quad iid. \tag{1}$$

1. ***FPOP*** et ***PELT*** résolvent le même problème d'optimisation avec des stratégies différentes et ont une complexité différente.

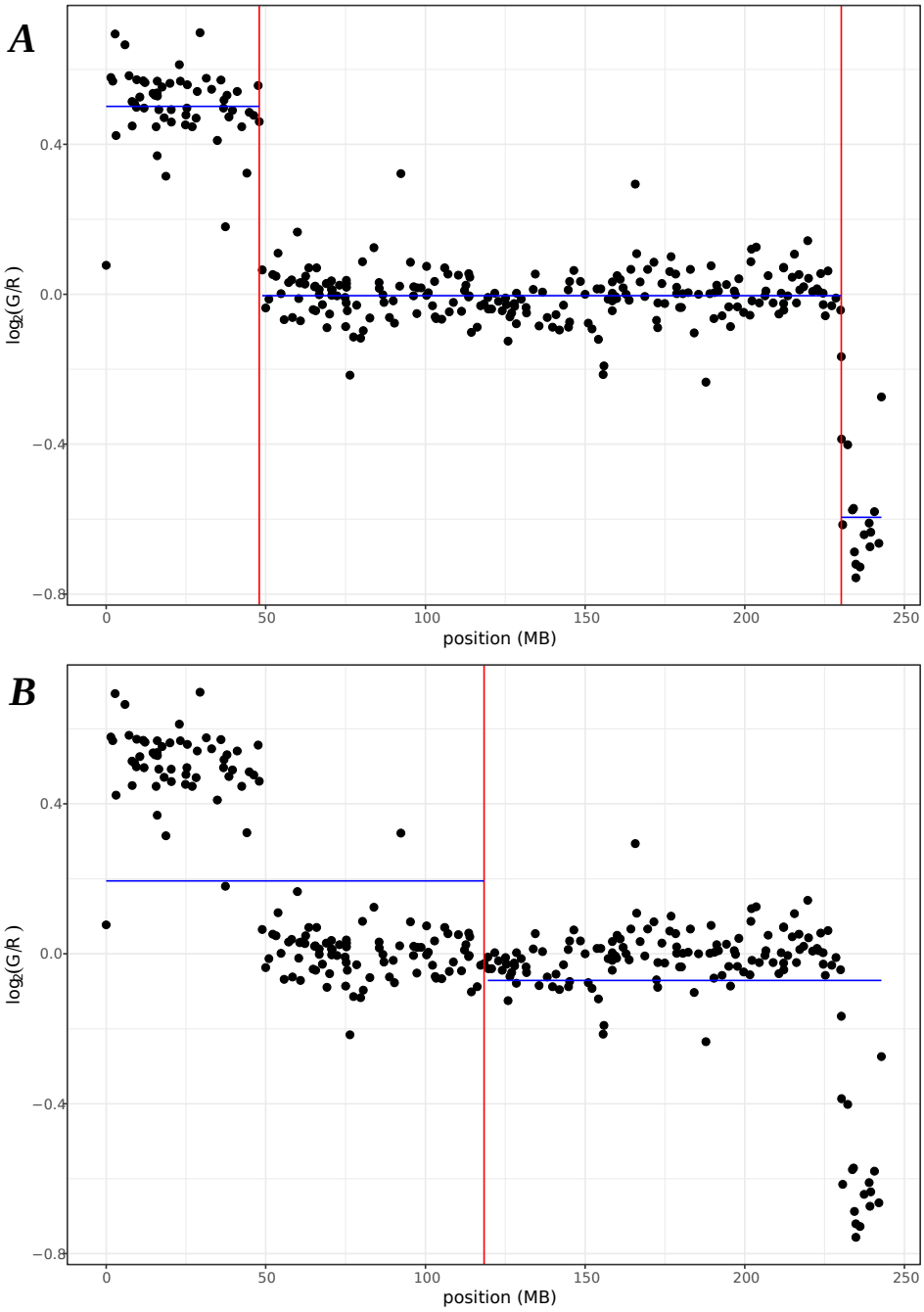


FIGURE 3 – L'origine des données est présentée dans la figure 1. ***(A)*** et ***(B)*** représentent 2 segmentations différentes des données. La localisation de chaque rupture est symbolisée par une droite rouge verticale. La moyenne de chaque segment est symbolisée par une droite bleue horizontale. ***(A)*** est la segmentation expertisée des données. ***(B)*** est une mauvaise segmentation où l'unique rupture a été choisie arbitrairement au milieu des données. Comme attendu, le coût de la segmentation A ($SCE_{(A)} = 1.87$) est inférieure au coût de la segmentation B ($SCE_{(B)} = 15.5$).

Le modèle (1) dépend d'un vecteur de paramètres $\theta = (\mu_1, \ldots, \mu_{K+1}, \sigma^2, \tau_1, \ldots, \tau_K)$. On note $\mathcal{L}(y_1, \ldots, y_n; \theta)$ la fonction de log-vraisemblance dérivée de ce modèle,

$$\mathcal{L}(y_1, \ldots, y_n; \theta) = \sum_{j=1}^{K+1} \log(f(y_{\tau_{j-1}+1}, \ldots, y_{\tau_j}; \mu_j, \sigma^2)) \tag{2}$$

avec $f(y_{\tau_{j-1}+1}, \ldots, y_{\tau_j}; \mu_j, \sigma^2)$ la distribution conjointe des observations indépendantes. Sachant l'hypothèse sur la normalité et l'indépendance des données on obtient :

$$\mathcal{L}(y_1, \ldots, y_n; \theta) = -\frac{n}{2} log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{j=1}^{K+1} \sum_{i=\tau_{j-1}+1}^{\tau_j} (y_i - \mu_j)^2. \tag{3}$$

Pour $K > 0$, on définit $\mathcal{M}_{1:n}^K$ l'ensemble de toutes les segmentations possibles de Y avec exactement K ruptures ainsi que $\boldsymbol{\tau}_n = \{\tau_1, \ldots, \tau_K\} \in \bigcup_{0<K<n} \mathcal{M}_{1:n}^K$, une segmentation particulière de Y. Les indices factices sont ici implicitement accessibles. On dénombre donc $\sum_{K=1}^{n-1} |\mathcal{M}_{1:n}^K| = \sum_{K=1}^{n-1} \binom{n-1}{K} = 2^{n-1}$ façons différentes de segmenter Y.

La problématique statistique considérée ici est l'inférence du nombre et de la position des ruptures à partir des données observées. Une méthode classique est de sélectionner la segmentation $\boldsymbol{\tau}_n^* \in \mathcal{M}_{1:n}^K$ optimisant un critère de vraisemblance pénalisée.

1.3 Définition du problème d'optimisation pénalisée

Dans mon stage, comme pour de nombreuses méthodes de recherches de ruptures, nous définissons le critère quantitatif qui doit être minimisé comme la somme du coût des segments formés par les ruptures d'une segmentation particulière de Y soit $\sum_{j=1}^{K} \mathcal{C}(y_{\tau_{j-1}+1}, \ldots, y_{\tau_j})$ avec $\mathcal{C}(y_{\tau_{j-1}+1}, \ldots, y_{\tau_j})$ une fonction qui renvoie le coût du $j^{ème}$ segment. Ce coût dépend de l'homogénéité des points $y_{\tau_{j-1}+1}, \ldots, y_{\tau_j}$. On s'attend à ce que ce coût soit faible quand le segment est homogène, c'est-à-dire qu'il ne contient pas de ruptures, et inversement qu'il soit fort quand le segment est hétérogène, c'est-à-dire qu'il contient une ou plusieurs ruptures.

Une approche naturelle pour obtenir $\boldsymbol{\tau}_n^*$ est de maximiser la log-vraisemblance (3). Dans (3) la variance est constante, donc maximiser la log-vraisemblance revient à minimiser la somme du carré des erreurs (SCE). Ainsi,

$$\mathcal{C}(y_{\tau_{j-1}+1}, \ldots, y_{\tau_j}) = \sum_{i=\tau_{j-1}+1}^{\tau_j} (y_i - \mu_j)^2 \quad \textit{(cf figure 3)}. \tag{4}$$

Nous nous intéressons au problème où le nombre de ruptures dans les données n'est pas connu. Le minimum de (4) est obtenu pour $\boldsymbol{\tau}_n^* = \{1, \ldots, n-1\}$. Afin d'éviter un tel surajustement aux données, une pénalité dépendante de la complexité de la segmentation est ajoutée [20, 21]. On note cette pénalité $pen(\boldsymbol{\tau}_n)$. L'objectif algorithmique est donc de résoudre le problème d'optimisation pénalisée suivant :

$$\boldsymbol{\tau}_n^* = \underset{\boldsymbol{\tau}_n \in \bigcup_{0<K<n} \mathcal{M}_{1:n}^K}{\arg\min} \left[\sum_{j=1}^{K+1} \min_{\mu} \left[\sum_{i=\tau_{j-1}+1}^{\tau j} (y_i - \mu)^2 \right] + pen(\boldsymbol{\tau}_n) \right]. \tag{5}$$

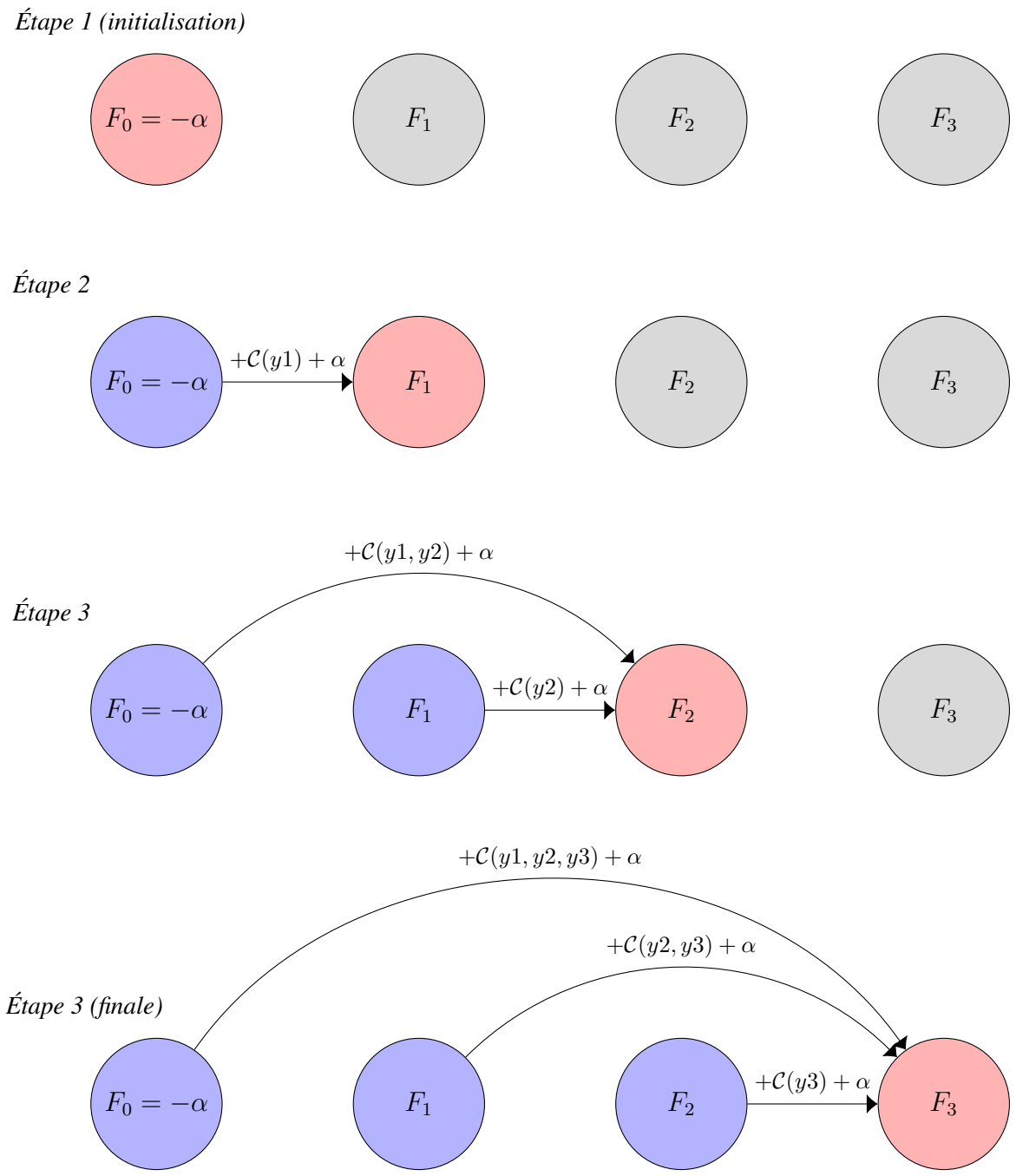


FIGURE 4 – Exemple illustré du calcul de F_3 avec l'aide de le méthode ***OP***. Les noeuds en bleu correspondent aux sous-problèmes déjà résolus. Le noeud en rouge correspond au sous-problème que l'on est en train de résoudre. Les noeuds en gris correspondent aux sous-problèmes qui ne sont pas encore résolus. Les flèches correspondent aux différentes solutions du sous-problème courant. Calculer F_t revient à chercher le plus court chemin entre F_0 et F_t

2 Résolution par programmation dynamique

L'espace des solutions du problème (5), à savoir 2^{n-1}, fait qu'on ne peut le résoudre de manière exacte dans un temps admissible en explorant toutes les solutions indépendamment. Comme je l'expliquerai un peu plus tard, (5) peut être décomposé en sous-problèmes qui ne sont pas indépendants. La programmation dynamique cherche et sauvegarde les solutions de ces sous-problèmes de plus en plus grands. Elle est aujourd'hui la seule approche connue pour résoudre exactement ce problème en un temps polynomial.

2.1 Optimal partitioning (*OP*)

En 2005, Jackson et. al [22] proposent une méthode fondée sur la programmation dynamique visant à minimiser (5). On note que cette méthode est très proche de celles proposées par Auger & Lawrence [23] et Bellman [24] pour un problème similaire mais différent. Jackson et al. ont choisi une pénalité (α) qui évolue linéairement avec le nombre de ruptures. Ainsi,

$$pen(\boldsymbol{\tau}_n) = \alpha \, |\boldsymbol{\tau}_n| \tag{6}$$

où α est classiquement de la forme $\beta \log(|Y|)$ [20].

Le coût de la meilleure segmentation des points $y_1, \ldots, y_t$, noté F_t, est obtenu en considérant les coûts des meilleures segmentations jusqu'à y_s, F_s, tel que $0 \leq s < t$ auxquels on ajoute le coût du dernier segment et la pénalité. Plus précisément, la méthode ***OP*** nous fournit une récursion pour calculer F_t ,

$$F_t = \min_{0 \leq s < t} \left[F_s + \mathcal{C}(y_{s+1}, \ldots, y_t) + \alpha \right], \quad \text{initialisée à } F_0 = -\alpha. \tag{7}$$

On peut prouver que la complexité en temps de la méthode ***OP*** est $\mathcal{O}(n^2)$ et que la complexité en mémoire est $\mathcal{O}(n)$ [22]. La figure 4 fournit un exemple illustré du calcul de F_3.

2.2 Functional Pruning Optimal Partioning (*FPOP*)

En 2016, Maidstone et al. [17] présentent une nouvelle méthode, ***FPOP***, qui améliore l'efficacité de ***OP*** en utilisant de l'élagage fonctionnel. La méthode ***FPOP*** s'appuie sur les travaux antérieurs de Rigaill 2010 [25] et Johnson et al. 2011 [26].

Les auteurs introduisent une fonction $\widetilde{f}_{t,s}(\mu)$ qui est définie comme le coût de la meilleure segmentation des données jusqu'au point t conditionnellement à la dernière rupture s et la moyenne du dernier segment μ. Ainsi,

$$\widetilde{f}_{t,s}(\mu) = F_s + \sum_{i=s+1}^{t} (y_i - \mu)^2 + \alpha, \quad \text{avec} \quad \widetilde{f}_{t,t}(\mu) = F_t + \alpha \quad \text{et} \quad F_0 = -\alpha. \tag{8}$$

Au cours de la procédure, $\widetilde{f}_{t,s}(\mu)$ est conservée et mise à jour avec la formule suivante :

$$\widetilde{f}_{t,s}(\mu) = \widetilde{f}_{t-1,s}(\mu) + (y_t - \mu)^2. \tag{9}$$

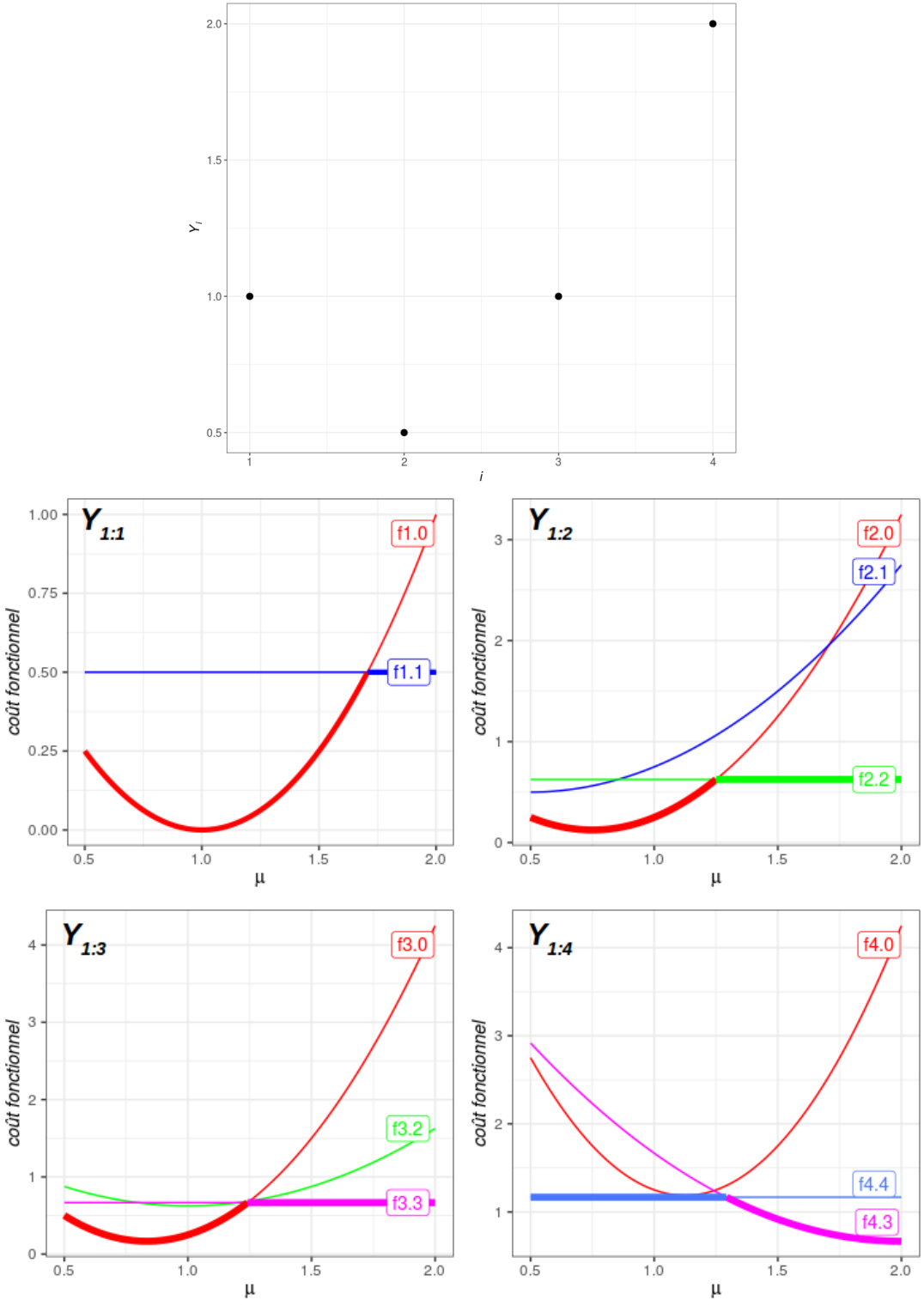


FIGURE 5 – Représentation d'une série de données Y composée de 4 points ($y_1 = 1.0,\ y_2 = 0.5,\ y_3 = 1.0,\ y_4 = 2.0$) et des fonctions de coût associées à chaque rupture candidate pour la meilleure segmentation des données $Y_{1:1},\ Y_{1:2},\ Y_{1:3}$ et $Y_{1:4}$.. À chaque étape, la fonction quadratique par morceau apparaît en gras.

À chaque étape t, l'algorithme ***FPOP*** considère le minimum des $\widetilde{f}_{t,s}(\mu)$, noté $\widetilde{F}_t(\mu)$, une fonction quadratique par morceaux.

$$\widetilde{F}_t(\mu) = \min_{s \leq t} \left\{ \widetilde{f}_{t,s}(\mu) \right\} \tag{10}$$

Parmi toutes les ruptures candidates, il existe un sous-ensemble de ces ruptures qui participent à ce minimum. Notez que F_t, la solution du problème (5), s'obtient en minimisant (10) sur μ : $F_t = \min_{\mu} \left[\widetilde{F}_t(\mu) \right]$. Maidstone et al. ont démontré que $\widetilde{F}_t(\mu)$ peut être mise à jour itérativement,

$$\widetilde{F}_t(\mu) = \min \left\{ \underbrace{\widetilde{F}_{t-1}(\mu) + (y_t - \mu)^2}_{\substack{\textit{meilleures ruptures candidates} \\ \textit{passées}}}, \quad \underbrace{F_t + \alpha}_{\substack{\textit{dernière rupture candidate} \\ \textit{introduite}}} \right\} \tag{11}$$

La récursion (11) suggère que pour mettre à jour $\widetilde{F}_{t-1}(\mu)$, il suffit de comparer l'ensemble des fonctions de coût des ruptures candidates participant à $\widetilde{F}_{t-1}(\mu)$ avec la fonction de coût de la rupture candidate dernièrement introduite, à savoir : $F_t + \alpha$. En pratique, le nombre de ruptures candidates qui participent à $\widetilde{F}_{t-1}(\mu)$ reste souvent limité et la mise à jour est rapide.

La récursion (11) justifie qu'à l'étape t, toutes les ruptures candidates s dont la fonction de coût $\widetilde{f}_{t,s}(\mu)$ ne participe pas à $\widetilde{F}_t(\mu)$ peuvent être élaguées. Plus formellement, pour chaque rupture candidate s on définit $Z^*_{t,s}$ l'ensemble des μ sur lesquels $\widetilde{f}_{t,s}(\mu)$ est égale à $\widetilde{F}_t(\mu)$.

$$Z^*_{t,s} = \{\mu \mid \widetilde{f}_{t,\tau}(\mu) = \widetilde{F}_t(\mu)\} \tag{12}$$

Sachant (11), on obtient

$$Z^*_{t,s} \supset Z^*_{t+1,s}. \quad \text{Et} \quad Z^*_{t,s} = \emptyset \implies Z^*_{t+1,s} = \emptyset \,. \tag{13}$$

Ci-dessous un exemple détaillé du calcul de la meilleure segmentation de Y, une série de données composée des points $(y_1 = 1.0,\ y_2 = 0.5,\ y_3 = 1,\ y_4 = 2)$, à l'aide de l'algorithme ***FPOP***. Nous fixons la pénalité α à 0.5.

1. ***Initialisation (cf. figure 5)*** :

   $$F_0 = -0.5$$
   $$\widetilde{f}_{0,0}(\mu) = F_0 + 0.5 = 0$$
   $$Z^*_{0,0} = [0.5, 2]$$

2. ***Étape intermédiaire (coût fonctionnel de*** $Y_{1:1}$***)*** : On met à jour la fonction de coût de la rupture candidate $s = 0$ puis on calcule F_1,

   $$F_1 = \min_{\mu} \left[F_0 + (y_1 - \mu)^2 \right] = \min_{\mu} \left[0 + (1 - \mu)^2 \right] = 0.$$

   On introduit la nouvelle rupture candidate $s = 1$ et sa fonction de coût $\widetilde{f}_{1,1}(\mu)$,
   $\widetilde{f}_{1,1}(\mu) = F_1 + 0.5 = 0.5$. La comparaison des différentes fonctions de coût montre que l'ensemble des μ sur lesquels $s = 0$ est la meilleure est $Z^*_{1,0} = [0.5, 1.7]$ et l'ensemble des μ sur lesquels $s = 1$ est la meilleure est $Z^*_{1,1} = [1.7, 2]$.

3. ***Étape intermédiaire (coût fonctionnel de*** $Y_{1:2}$***)*** : On met à jour la fonction de coût des ruptures candidates $s = 0$ et $s = 1$ puis on calcule F_2,

$$F_2 = \min_{\mu} \left[\begin{array}{l} F_0 + (y_1 - \mu)^2 + (y_2 - \mu)^2, \\ F_1 + 0.5 + (y_2 - \mu)^2 \end{array} \right] = \min_{\mu} \left[\begin{array}{l} 0 + (1 - \mu)^2 + (0.5 - \mu)^2, \\ 0 + 0.5 + (0.5 - \mu)^2 \end{array} \right] = 0.125.$$

On introduit la nouvelle rupture candidate $s = 2$ et sa fonction de coût $\widetilde{f}_{2,2}(\mu)$, $\widetilde{f}_{2,2}(\mu) = F_2 + 0.5 = 0.625$. La comparaison des différentes fonctions de coût montre que l'ensemble des μ sur lesquels $s = 0$ est la meilleure est $Z^*_{2,0} = [0.5, 1.25]$, l'ensemble des μ sur lesquels $s = 1$ est la meilleure est vide et l'ensemble des μ sur lesquels $s = 2$ est la meilleure est $Z^*_{2,2} = [1.25, 2]$. Sachant (13) nous pouvons élaguer $s = 1$ dans les étapes suivantes.

De la même manière nous pouvons ensuite calculer le coût fonctionnel de $Y_{1:3}$ puis $Y_{1:4}$.

Comme ***OP***, on peut prouver que la complexité en temps de la méthode ***FPOP*** dans le pire cas est $\mathcal{O}(n^2)$ [25] et que la complexité en mémoire est $\mathcal{O}(n)$. Cependant, empiriquement la complexité moyenne en temps est de l'ordre de $n \log(n)$. Aujourd'hui, ***FPOP*** est l'algorithme exact connu le plus rapide. Il est moins générique que ***PELT***, un autre algorithme exact de segmentation [18], mais élague plus, en particulier quand le nombre de ruptures est faible [17].

2.3 Extension de *FPOP* au cas d'une pénalité dépendante de la taille des segments (*FpopPSD*)

Comme montré dans l'étude de Hocking et al [16], les méthodes de segmentation existantes ne permettent pas d'atteindre la précision d'une annotation experte sur des données de CNV. Dans un soucis d'améliorer les performances de détection des ruptures, par exemple dans le cadre de la recherche des CNV, je propose aujourd'hui une nouvelle méthode de segmentation optimale des données, ***FpopPSD***, s'appuyant sur une pénalité dépendante de la taille des segments et qui, comme ***FPOP***, utilise de l'élagage fonctionnel.

FpopPSD se décline en deux versions. Une première version qui pénalise plus fortement les grands segments et une deuxième version qui pénalise plus fortement les petits segments.

Dans le cadre de la segmentation de signaux génomiques obtenus via des expériences de CGH array, SNP array ou encore NGS, plusieurs points nous laissent a priori penser que pénaliser plus fortement les petits segments conduirait à une meilleure segmentation des données :

- statistiquement les méthodes basées sur la vraisemblance n'incorporant pas de pénalité sur la longueur sont sensibles aux points éloignés de la moyenne (outliers). Même dans le cas Gaussien de très petits segments artéfactuels sont identifiés *(cf. figure 9.C)*. Par ailleurs, les segments de quelques points sont souvent difficiles à interpréter biologiquement car exclure qu'ils soient de simples artefacts techniques est difficile ;
- N. Verzelen à suggéré à G. Rigaill, sur la base de travaux théoriques en cours, d'utiliser une telle pénalité pour améliorer la sélection de modèle. Les résultats de mes simulations *(cf. figure 9 & 11)* valident sur quelques scénarios la pertinence de ces travaux ;
- notez que même si une telle pénalité défavorise les petits segments, elle n'empêche pas leur détection *(cf. Annexes : code R)*.

Au début de mon stage, j'ai conçu avec l'aide de G. Rigaill un nouvel algorithme qui étend l'algorithme ***FPOP*** à l'utilisation d'une pénalité dépendante de la taille des segments. Dans le reste de cette section je présente les étapes de conception de cet algorithme.

2.3.1 Définition du problème

En prenant en compte la pénalité dépendante de la taille des segments on peut réécrire le problème d'optimisation pénalisée (5) de la manière suivante :

$$\boldsymbol{\tau}_n^* = \underset{\boldsymbol{\tau}_n \in \bigcup_{0<K<n} \mathcal{M}_{1:n}^K}{\arg\min} \left[\sum_{j=1}^{K+1} \min_{\mu} \left[\sum_{i=\tau_{j-1}+1}^{\tau j} (y_i - \mu)^2 - \beta g(\tau_j - \tau_{j-1}) \right] + \alpha |\boldsymbol{\tau}_n| \right] \tag{14}$$

où α est de la forme $\gamma + \beta g(n)$, avec γ est une constante qui reste à calibrer. N. Verzelen a suggéré de fixé la valeur de β à 2.5 et de prendre $g = \log$.

En terme d'élagage fonctionnel on peut alors définir, comme dans ***FPOP***, $\widetilde{f}_{t,s}(\mu)$ de la manière suivante :

$$\widetilde{f}_{t,s}(\mu) = F_s + \sum_{i=s+1}^{t} (y_i - \mu)^2 + \alpha - \beta g\left(|t-s|\right). \tag{15}$$

avec $\widetilde{f}_{t,t}(\mu) = F_t + \alpha$ et $F_0 = -\alpha$.

Durant la procédure, comme pour ***FPOP***, cette fonction de coût est conservée et mise à jour avec la formule suivante :

$$\widetilde{f}_{t,s}(\mu) = \widetilde{f}_{t-1,s}(\mu) + (y_t - \mu)^2 + \beta g\left(|t-1-s|\right) - \beta g\left(|t-s|\right) \tag{16}$$

De manière analogue à ***FPOP*** on peut calculer F_t en minimisant $\widetilde{f}_{t,s}$ à la fois sur μ et s.

La principale différence avec ***FPOP*** est que la règle (13) n'est plus vraie pour ***FpopPSD*** car $\widetilde{f}_{t,s}(\mu) - \widetilde{f}_{t,s'}(\mu)$ dépend de t. Cela implique de ré-évaluer les comparaisons entre les ruptures candidates s et s' à divers t. On ne peut pas garantir que si $Z_{t,s}^* = \emptyset$ alors $Z_{t+1,s}^* = \emptyset$.

2.3.2 Mise à jour de la zone de vie des ruptures candidates ($Z_{t,s}$)

Plutôt que d'évaluer la zone de vie exacte $Z_{t,s}^*$ des ruptures candidates, nous cherchons à mettre à jour une zone de vie $Z_{t,s}$ incluant $Z_{t,s}^*$ et validant (13). Pour cela, les règles de mise à jour de $Z_{t,s}$ vers $Z_{t+1,s}$ doivent garantir que,

$$Z_{t+1,s} \supset Z_{t+1,s}^* \,. \tag{17}$$

Pour la suite de ce rapport, je définis l'intervalle $I_{t,s,s'}$ tel que $s < s^{'}$,

$$I_{t,s,s'} = \{\mu \mid \widetilde{f}_{t,s}(\mu) \;\leq\; \widetilde{f}_{t,s'}(\mu)\}. \tag{18}$$

La comparaison de $\widetilde{f}_{t,s}(\mu)$ avec $\widetilde{f}_{t,s'}(\mu)$ donne

$$\widetilde{f}_{t,s}(\mu) - \widetilde{f}_{t,s'}(\mu) = \underbrace{F_s - F_{s'} + \sum_{i=s+1}^{s'} (y_i - \mu)^2}_{forme\ quadratique\ constante} + \beta(\underbrace{g(t-s') - g(t-s)}_{\substack{h,\ une\ fonction\ qui\ varie \\ avec\ t,\ s\ et\ s'}}). \tag{19}$$

Le calcul des racines de (19) permet de déterminer $I_{t,s,s'}$.

J'ai pu définir deux règles de mise à jour de $Z_{t,s}$ vers $Z_{t+1,s}$ selon le comportement de $h(t,s,s') = g(t-s') - g(t-s)$. Je présente dans cette section le cas (A) où h est une fonction monotone croissante sur t et que $\lim_{t\to\infty} h(t,s,s') = 0$ donc $h(t,s,s') \leq 0$. Le cas où h est une fonction monotone décroissante et que $\lim_{t\to\infty} h(t,s,s') = 0$ n'est pas développé dans ce rapport, mais je l'ai étudié durant mon stage.

Dans le cas (A) comme h est croissante sur t,

$$I_{t+1,s,s'} \subset I_{t,s,s'} . \tag{20}$$

On définit alors $I_{\infty,s,s'}$ qui correspond à $I_{t,s,s'}$ lorsque $t \to \infty$. Le calcul des racines de $F_s - F_{s'} + \sum_{i=s+1}^{s'} (y_i - \mu)^2$ permet de déterminer $I_{\infty,s,s'}$.

Avec G. Rigaill nous proposons la règle de mise à jour :

$$Z_{t+1,s} = Z_{t,s} \cap \overbrace{(\bigcap_{s'} I_{t+1,s,s'})}^{comparaisons\ avec\ le\ futur} \setminus \overbrace{(\bigcup_{s''} I_{\infty,s'',s})}^{comparaisons\ avec\ le\ passé} . \tag{21}$$

J'ai montré que la règle (21) valide (17) et permet d'élaguer *(voir le Théorème en annexe)*. En effet, si $Z_{t,s}$ est vide alors on peut élaguer s car pour tout $k \geq 0$ la zone de vie exacte $Z^*_{t+k,s}$ est inclue dans $Z_{t,s}$ et par conséquent est vide.

2.3.3 Comparaison des ruptures candidates et simplification de la règle de mise à jour

La Règle (21) suggère que pour chaque s, il faut la comparer à des ruptures candidates futures s' et des ruptures candidates passées s''. Pour les ruptures candidates passées il faut considérer t qui tend vers l'infini ($I_{\infty,s'',s}$). Dans ce cas, l'ensemble des comparaisons peuvent être effectuées une fois que la rupture candidate est introduite. Pour les ruptures candidates futures il faut régulièrement comparer s à des s'. Effectuer à chaque étape, pour chaque s, une comparaison avec chaque s'' est source de complexité. Plus précisément, la complexité de chaque étape est en $\mathcal{O}$(nombre de ruptures candidates $\times$ complexité de la mise à jour des ruptures candidates). Idéalement, pour chaque s, on aimerait effectuer le minimum de comparaisons qui entraîneraient son élagage. Dans l'algorithme 1, via la fonction *ech*, je propose différentes stratégies d'échantillonnage des s'' dans l'intention d'accélérer ce dernier. Cette fonction est détaillée dans la section 3.2.6. Notez que l'échantillonnage des s'' ne change pas l'exactitude de l'algorithme.

Algorithm 1: ***FpopPSD***

Input: $Y = (y_1, ..., y_n),\ \alpha, \beta,\ g = log(.)$
Output: l'ensemble des ruptures cp_n

$n \leftarrow |Y|$;
$F_0 \leftarrow -\alpha$;
$cp_0 \leftarrow \emptyset$;
$R_1 \leftarrow \{0\}$;
$D \leftarrow [min(Y),\ max(Y)]$;
$Z_{0,0} \leftarrow D$;
$\widetilde{f}_{0,0} \leftarrow F_0 + \alpha\ (= 0)$;
for $t \leftarrow 1, ..., n$ **do**
    for $s \in R_t$ **do**
        $\widetilde{f}_{t,s}(\mu) \leftarrow \widetilde{f}_{t,s}(\mu) + (y_t - \mu)^2 + \beta \times g(t-1-s) - \beta \times g(t-s)$;
    end
    $F_t \leftarrow_{s \in R_t} (\min_{\mu \in Z_{t,s}}(\widetilde{f}_{t,s}(\mu)))$;
    $s_t \leftarrow \arg\min_{s \in R_t}(\min_{\mu \in Z_{t,s}}(\widetilde{f}_{t,s}(\mu)))$;
    $cp_t \leftarrow (cp_{s_t}, s_t)$;
    $\widetilde{f}_{t,t} \leftarrow F_t + \alpha$;
    $Z_{t,t} \leftarrow D$;
    for $s \in R_t$ **do**
        $Z_{t,t} \leftarrow Z_{t,t} \setminus I_{\infty,s,t}$;
        $set \leftarrow ech(\{s^{'} \in \{R_t \cup \{t\}\} : s^{'} > s\})$;
        $Z_{t,s} \leftarrow Z_{t,s} \cap (\bigcap_{s' \in\ set} I_{t,s,s'})$;
    end
    $R_{t+1} \leftarrow \{s \in \{R_t \cup \{t\}\} : Z_{t,s} \neq \emptyset\}$;
end

Le pseudocode de l'algorithme (dans sa version littérale) qui met en oeuvre (21) est disponible ci-dessous. J'en donne une version détaillée dans l'algorithme 1.

Algorithm 2: ***FpopPSD*** (*traduction littérale de l'algorithme 1*)

Input: *les données à segmenter et des paramètres de lissage*
Output: *les ruptures formant la meilleure segmentation des données telle que définie par (14)*

initialisation $(instructions\ 1, 2, 3, 4, 5, 6, 7,)$;
for *chaque nouveau point considéré* **do**
    - *Mise à jour de la fonction de coût de chaque rupture candidate* $(instruction\ 10)$;
    - *Recherche du meilleur candidat* $(instructions\ 12, 13, 14)$;
    - *Initialisation de la zone de vie de la nouvelle rupture candidate par comparaison avec les ruptures candidates passées* $(instruction\ 15, 16, 18)$;
    - *Mise à jour des ruptures candidates par comparaison avec un échantillon de ruptures candidates futures* $(instruction\ 19, 20)$;
    - *Élagage des ruptures candidates dont la zone de vie est vide* $(instruction\ 22)$;
end

3 Implémentation de la méthode *FpopPSD*

3.1 Généralités

Les concepts que la méthode ***FpopPSD*** est amenée à manipuler (Ex : fonction quadratique, candidats, intervalles) sont de natures très diverses. Sur ce constat, j'ai choisi d'orienter mon choix d'implémentation vers la programmation objet. Ce type de programmation, à travers sa structure en classes où sont regroupées toutes les informations sur un objet et les méthodes pour le manipuler, permet notamment de faciliter la compréhension et la relecture du code. La programmation orientée objet offre aussi un avantage certain quant à la modularité du code permettant ainsi d'éviter la création de code redondant.

J'ai effectué la première implémentation de la méthode FpopPSD en python. Ce choix est en parti motivé par mon intérêt pour ce langage. Python est par ailleurs un langage de programmation orienté objet et par conséquent tout à fait adapté à notre problématique. Certains points (pas de pré-compilation, un typage dynamique, une syntaxe succincte, un debugger intégré) font de lui un langage qui permet une grande productivité. Ainsi, j'ai implémenté et testé de nombreux objets très rapidement. Cependant, même si sa bibliothèque standard contient des modules natifs écrits en C, python est un langage de programmation interprété dont les performances n'égalent pas celles des langages de programmation compilés comme le C++.

En m'appuyant sur la maquette python, j'ai effectué une deuxième implémentation en C++. Le C++ est lui aussi un langage de programmation orienté objet ce qui légitime son utilisation dans notre cas. La bibliothèque standard du C++, largement utilisée pour cette implémentation de la méthode ***FpopPSD***, met à disposition du programmeur des outils puissants comme les collections (conteneurs), les itérateurs ainsi que des fonctions de tri, de recherche, de comptage, de manipulation et bien d'autres.

Un package R du même nom que la méthode a été créé avec l'aide du package Rcpp. Rcpp permet d'appeler du code C++ dans R grâce à l'implémentation d'une fonction wrapper. Je déposerai le package ***FpopPSD*** sur ForgeMIA à la fin de mon stage.

La figure 6 correspond au diagramme de classe du projet mis à jour pendant la conception et l'implémentation de ***FpopPSD***. Dans la suite de cette section je présenterai dans une première partie les différentes classes du projet ***FpopPSD***. Je consacrerai une deuxième partie aux taches effectuées pour déboguer mon implémentation.

FIGURE 6 – Diagramme de classes du projet FpopPSD. Une flèche pleine représente une association, une flèche pointillée représente une dépendance et un losange représente une agrégation.

3.2 Présentation des classes

Six classes ont été identifiées lors de la conception de la méthode ***FpopPSD*** :

- la première classe, ***Candidate***, définit le concept de rupture candidate. Chaque rupture candidate est caractérisée par sa localisation. D'un point de vue fonctionnel, elle est associée à une fonction de coût et sa zone de vie. La fonction de coût peut être décomposée en 3 parties : le coût de la meilleure segmentation des données jusqu'à la rupture candidate, la forme quadratique et la pénalité qui dépend du segment formé par les points situés après la rupture ;
- la seconde classe, ***Interval***, définit le concept d'intervalle. De manière classique, l'intervalle est borné par deux réels. Un intervalle vide est représenté par une borne supérieure plus petite que la borne inférieure. Comme justifié dans 3.2.2, j'ai choisi de traiter les singletons [2] comme des intervalles vides ;
- la troisième classe, ***Ordered_list_of_intervals*** définit une liste d'intervalles non vides et ordonnés par leur borne inférieure. La propriété ordonnée de cette liste est exploitée pour améliorer les performances de mise à jour de la zone de vie des fonctions de coût ;
- la quatrième classe, ***FpopPSD***, est la classe principale du projet. Elle permet d'instancier un problème de segmentation à partir des données à segmenter et des paramètres de lissage. En s'appuyant sur les autres classes, elle implémente la procédure de recherche des ruptures ;
- la cinquième classe, ***Quadratic***, définit la forme quadratique $a0 + a1x + a2x^2$. Cette quadratique est une des composantes de la fonction de coût associée à chaque rupture candidate ;
- la sixième classe, ***Sampling***, implémente différentes stratégies d'échantillonnage des ruptures candidates futures. On peut diviser ces stratégies en deux grandes classes : déterministes et aléatoires.

La classe ***OpPSD***, non détaillée dans ce rapport, est une simplification de ***FpopPSD*** à laquelle on a enlevé l'étape de mise à jour et l'étape d'élagage des ruptures candidates. Comme nous le verrons dans la section suivante, cette classe, proche conceptuellement de la méthode ***OP***, est utilisée pour déboguer ***FpopPSD***.

Dans le reste de cette section je détaille les six classes présentées précédemment ainsi que les relations qui lient les objets qu'elles définissent. Afin d'améliorer la lisibilité de cette section, j'ai défini une syntaxe pour chaque concept de programmation auquel je ferai référence, à savoir : ***Une_classe***, *un_attribut / une_variable*, *Une_méthode_d_objet()*, *Une_méthode_de_classe()*. Pour obtenir la liste des paramètres de chaque méthode, on peut se référer au diagramme *(cf. figure 6)*.

3.2.1 *Candidate*

La classe ***Candidate*** définit une rupture candidate. Cette dernière est caractérisée par sa position s dans la série de données. On décompose et on sauvegarde sa fonction de coût $\widetilde{f}_{t,s}(\mu)$ (15) dans 3 attributs distincts. Ainsi, on sauvegarde la partie constante de cette fonction, à savoir $F_s + \alpha$, dans *cost_up_to_s*. On sauvegarde sa forme quadratique, à savoir $\sum_{i=s+1}^{t}(y_i - \mu)^2$, dans *quadratic* via la classe du même nom. Enfin, on sauvegarde la pénalité dépendante de la taille du segment formé par les points situés après s, c'est-à-dire $-\beta \log(|t - s|)$, dans *pen*. La zone de vie de la fonction de coût, un objet de type ***Ordered_list_of_intervals***, est sauvegardé dans z.

2. Un singleton est un intervalle de la forme [a,a].

Cette classe dispose d'un constructeur permettant d'instancier une rupture candidate en spécifiant les paramètres correspondant aux différents attributs présentés ci-dessus. Plusieurs méthodes permettent d'interagir avec la rupture candidate courante :

- *Minimum_of_cost_function()* renvoie le minimum de la fonction de coût en faisant premièrement appel à *Minimum()* sur la forme quadratique de la rupture candidate courante, qui renvoie le meilleur coût sur le dernier segment, puis en additionnant *cost_up_to_s* et *pen*, le coût de la meilleure segmentation des données jusqu'à la rupture candidate courante ;
- *Set_penalty(...)* met à jour *pen* avec la valeur spécifiée. ;
- *Add_quadratic(...)* ajoute le point courant y_i à la forme quadratique de la rupture courante par addition des coefficients ($a_0 = a_0 + y_i^2;\ a_1 = a_1 + 2y_i;\ a_2 = a_2 + 1$) ;
- *GetZ(), Get_s()* sont les accesseurs respectivement de *z* et *s* de la rupture candidate courante ;
- *Compare_to_past_candidate(...)* met à jour z de la rupture candidate courante en comparant cette dernière aux ruptures candidates introduites avant elles. La rupture candidate courante est la rupture candidate dernièrement introduite. Cette procédure correspond à l'instruction 18 de l'algorithme 1. L'approche naïve qui consiste à retrancher pour chaque rupture candidate passée s'', $I_{\infty,s'',s}$ à z peut se révéler complexe si z contient plus d'un intervalle. Je propose une autre approche. On instancie avec le constructeur approprié de ***Ordered_list_of_intervals*** l'union triée des $I_{\infty,s'',s}$. Dans un deuxième temps, on cherche le complémentaire de l'union d'intervalles triés formée précédemment dans $[\min(Y), \max(Y)]$, où $[\min(Y), \max(Y)]$ est la zone de vie par défaut des ruptures candidates dernièrement introduites. Cette étape est effectuée en appelant la méthode *Complementary_in(...)*. L'objet renvoyé de type ***Ordered_list_of_intervals*** est utilisé pour mettre à jour z ;
- *Compare_to_future_candidate(...)* met à jour z de la rupture candidate courante en comparant cette dernière à un échantillon des ruptures introduites après elle, notées s'. Cette procédure correspond à l'instruction 20 de l'algorithme 1. On commence par instancier l'intersection des $I_{t,s,s'}$ avec le constructeur approprié de la classe ***Interval***. Dans un deuxième temps on intersecte chaque intervalle contenu dans z avec l'intersection instanciée précédemment. Cette étape est effectuée en appelant la méthode *Intersect_with()*.

3.2.2 *Interval*

La classe ***Interval*** définit un intervalle borné dans $\mathbb{R}$. Les bornes inférieure et supérieure sont sauvegardées dans deux attributs de type double, respectivement *begin* et *end*. Tout intervalle de la forme $[a, b]$ tel que $a \geq b$ est considéré comme un intervalle vide. Dans le cas des singletons, ce choix se justifie par les propriétés de la fonction quadratique par morceau (10). Cette fonction est continue ce qui implique que les zones de vie des fonctions de coût adjacentes à celle associée au singleton s'intersectent en ce point. On peut donc choisir de ne plus considérer la fonction de coût associée au singleton, ce qui revient à considérer sa zone de vie comme vide.

Cette classe dispose de 3 constructeurs. Le premier permet d'instancier un intervalle vide. Le second permet d'instancier un intervalle avec des bornes fournies explicitement. Le dernier permet d'instancier un intervalle qui correspond à l'intersection des intervalles sauvegardés dans une liste non ordonnée. Pour ce faire, on recherche en temps linéaire la plus grande des bornes inférieures et la plus petite des bornes supérieures. Plusieurs méthodes permettent d'interagir avec l'intervalle courant :

- *IsEmpty_or_singleton()* renvoie vrai si l'intervalle courant est de la forme $[a, b]$ tel que $a \geq b$, faux sinon ;
- *Get_begin(), Get_end(), Set_begin(...), Set_end(...)* sont les accesseurs et les mutateurs des bornes de l'intervalle courant ;

- *operator&=(...)* modifie les bornes de l'intervalle courant pour qu'elles soient égales aux bornes de l'intersection de l'intervalle courant avec l'autre intervalle sur lequel on applique cet opérateur.

Interval implémente aussi deux méthodes de classe *Compare_begin(...)* et *Compare_end(...)* qui renvoient vrai lorsque respectivement la borne inférieure du premier intervalle comparé est plus petite que la borne inférieure du second intervalle comparé et que la borne supérieure du premier intervalle comparé est plus petite que la borne supérieure du second intervalle comparé, faux sinon. Ces deux méthodes de comparaison sont utilisées par les algorithmes de la bibliothèque standard pour manipuler des conteneurs d'intervalles (Ex :, recherche de la borne supérieure minimale, tri sur la base de la borne inférieure).

3.2.3 *Ordered_list_of_intervals*

La classe ***Ordered_list_of_intervals*** définit une liste d'intervalles non vide pour laquelle on s'assure que les intervalles sauvegardés sont ordonnés suivant leur borne inférieure. Le conteneur *list_of_intervals* est de type ***std***::***list*** et a les propriétés d'une liste doublement chaînée. La possibilité de supprimer un élément en temps constant (linéaire avec le nombre d'élément à supprimer) fait que ce conteneur se révèle particulièrement efficace pour modéliser la zone de vie dynamique de la fonction de coût des ruptures candidates.

Cette classe dispose de 3 constructeurs. Le premier permet d'instancier une liste vide. Le second permet d'instancier une liste avec un intervalle fourni explicitement. Le dernier permet d'instancier une liste d'intervalles triés et d'intersections vides à partir d'une liste d'intervalles possiblement non ordonnés et possiblement d'intersections non vides. Dans ce dernier cas, une opération de tri est premièrement effectuée sur la borne inférieure des intervalles contenus dans la liste fournie. Cette opération a une complexité en $\mathcal{O}(|list\_of\_intervals\_to\_merge| \times log(|list\_of\_intervals\_to\_merge|))$. On effectue ensuite l'union des intervalles triés. Cette opération a une complexité en $\mathcal{O}(|list\_of\_intervals\_to\_merge|)$. Plusieurs méthodes permettent d'interagir avec la liste triée d'intervalles courante :

- *Intersect_with(...)* met à jour la liste triée d'intervalles courante en intersectant sur place, via l'opérateur *&=* surchargé dans ***Interval***, chaque intervalle contenu dans cette liste avec *interval_to_intersect*. Si une intersection est vide on supprime en temps constant l'intervalle concerné de la liste courante ;
- *Complementary_in(...)* met à jour la liste triée d'intervalles courante avec le complémentaire dans *d* des intervalles contenus dans cette liste. La première étape consiste à appeler *Intersect_with(d)* sur la liste triée d'intervalles courante. La liste courante ne contient désormais que des intervalles strictement inclus dans *d*. Cette opération a une complexité en temps linéaire. L'opération de recherche du complémentaire dans *d* s'effectue ensuite elle aussi en temps linéaire ;
- *Is_empty()* renvoie vrai si la liste est vide, faux sinon ;
- *Clean()* supprime les intervalles vides de la liste triée d'intervalles courante ;
- *Size()* renvoie la taille de la liste triée d'intervalles courante ;
- *Get_list()* renvoie la liste triée d'intervalles courante.

3.2.4 *FpopPSD*

La classe ***FpopPSD*** est la classe principale de ce projet. Elle implémente plusieurs attributs permettant de paramétrer le problème de recherche de ruptures. On instancie un problème via le constructeur de cette classe en spécifiant un vecteur de données à segmenter (*y*), les constantes *alpha* et *beta* utilisées dans le calcul des pénalités, un vecteur de poids (*wt*) utilisés dans la mise à jour des formes quadratiques, une méthode d'échantillonnage (*sampling_method*) parmi celles proposées par ***Sampling*** ainsi que le paramètre associé (*sampling_method_parameter*).

La procédure *search()* recherche les ruptures dans les données du problème courant. Durant toute la procédure on maintient parallèlement à jour une liste de ruptures candidates ainsi qu'un vecteur d'itérateurs dont chaque itérateur pointe vers une rupture candidate. Grâce à l'opérateur [] implémenté par la classe ***std*::*vector***, on peut accéder en temps constant à chaque rupture candidate sans avoir besoin de parcourir cette liste.

À chaque étape t *(cf. boucle principale de l'algorithme 1)* :

- on met à jour la forme quadratique en appelant *Add_quadratic(...)* sur chaque rupture candidate avec pour arguments, le nouveau point $y[t]$ et la valeur de pondération $wt[t]$;
- on met à jour pour chaque rupture candidate la pénalité calculée sur le dernier segment actualisé avec l'introduction du point $y[t]$, en appelant *Set_penalty(...)* ;
- on cherche le coût minimum (F_t) parmi toutes les fonctions de coût en appelant *Minimum_of_function _cost()* sur chaque rupture candidate. On sauvegarde ce coût et la rupture candidate associée ;
- on instancie une nouvelle rupture candidate t via le constructeur de ***Candidate***. La forme quadratique de sa fonction de coût et la pénalité calculée sur le dernier segment sont nulles. La partie constante de sa fonction de coût est égale à $F_t + \alpha$;
- on appelle *Compare_to_past_candidate(...)* sur la rupture candidate dernièrement introduite ;
- pour chaque rupture candidate on échantillonne les ruptures candidates futures via *sampling_method* puis on appelle *Compare_to_future_candidate(...)* ;
- on supprime de la liste contenant les ruptures candidates celles dont la zone de vie de leur fonction de coût est vide.

La méthode *Retreive_changepoints()* cherche récursivement et renvoie la liste des ruptures formant la meilleure segmentation des données du problème courant.

3.2.5 *Quadratic*

La classe ***Quadratic*** définit la forme quadratique $a0 + a1x + a2x^2$. Les coefficients, $a0, a1$ et $a2$, de la forme quadratique sont sauvegardés dans *coef* un tableau de doubles de taille 3. Cette classe dispose de deux constructeurs qui permettent d'instancier une forme quadratique avec des coefficients nuls ou des coefficients fournis explicitement. Plusieurs méthodes permettent d'interagir avec la forme quadratique courante :

- *Add_coef(...)* modifie la forme quadratique courante en lui ajoutant explicitement les coefficients d'une autre forme quadratique ;
- *Minimum()* renvoie le minimum de la forme quadratique courante, soit $a_0 - \frac{a_1^2}{4a_2}$;
- *Negative_interval()* calcule dans un premier temps le discriminant de la forme quadratique courante ($a1^2 - 4 \times a2 \times a0$). Si le discriminant est strictement supérieur à 0, la méthode instancie et renvoie un objet ***Interval*** dont les bornes correspondent aux racines ordonnées de la forme quadratique courante. Sinon, la méthode instancie et retourne un objet ***Interval*** vide ;
- *operator-(...)* surcharge de l'opérateur "-". Cet opérateur instancie et renvoie une forme quadratique dont les coefficients sont égaux à la différence des coefficients de la forme quadratique courante et d'une autre forme quadratique sur lesquels on a appliqué cet opérateur.

3.2.6 *Sampling*

La classe ***Sampling*** implémente différentes méthodes d'échantillonnage des entiers positifs compris entre deux bornes exclues. Ces méthodes servent à échantillonner les ruptures candidates futures (instruction 19 de l'algorithme 1).

- *Rand_with_replacement(...)* renvoie sous forme de vecteur *nb* entiers positifs tirés aléatoirement avec remise dans $]begin, end[$.
- *Rand_without_replacement(...)* renvoie sous forme de vecteur *nb* entiers positifs tirés aléatoirement sans remise dans $]begin, end[$, si $nb < |\ ]begin, end[\ |$. Sinon, la méthode renvoie l'ensemble des entiers positifs compris dans $]begin, end[$.
- *Rand_one(...)* renvoie sous forme de vecteur un entier positif compris dans l'intervalle $]begin, end[$.
- *All(...)* renvoie sous forme de vecteur l'ensemble des entiers positifs compris dans $]begin, end[$.
- *Last(...)* renvoie sous forme de vecteur les *nb* plus grands entiers positifs de l'intervalle $]begin, end[$.
- *Only(...)* renvoie sous forme de vecteur le $nb^{ème}$ plus grand entier positif de l'intervalle $]begin, end[$, si $end - nb > begin$. Sinon, la méthode renvoie le plus grand entier positif de l'intervalle $]begin, end[$.

3.3 Débogage

La phase d'implémentation C++ de la méthode ***FpopPSD*** a été suivie d'une phase de tests visant à contrôler l'optimalité des résultats renvoyés par cette méthode ainsi que les éventuelles fuites mémoires qui peuvent entraîner l'arrêt de la procédure de recherche des ruptures.

Afin de garantir l'optimalité des résultats renvoyés par la méthode ***FpopPSD***, j'ai comparé cette dernière avec ***OpPSD***, ***FPOP*** [17] et ***PELT*** [18] sur des profils de bruit Gaussien de moyenne nulle et d'écart type 1. Chaque profil contenait 10000 points. Pour chacun des profils j'ai comparé la localisation des ruptures renvoyées par chacune de ces méthodes. Pour que les résultats soient comparables, le β contrôlant le poids de la pénalité sur la taille des segments a été fixé à 0 pour ***FpopPSD*** et ***OpPSD***. La pénalité linéaire avec le nombre de ruptures, notée α dans ***FpopPSD*** et ***OpPSD***, a été fixée pareillement pour l'ensemble des méthodes comparées.

En parallèle, via l'application valgrind (`http://www.valgrind.org/`), j'ai testé la présence de fuites mémoires dans l'implémentation de ***FpopPSD***. C++ nous autorise à allouer la mémoire de façon dynamique et celle-ci ne sera pas libérée automatiquement par le système. Ce dernier point nous oblige à libérer cette zone lorsqu'on n'en a plus besoin. Dans le cas contraire, on provoque des fuites mémoire. Dans notre cas, l'utilisation des objets proposés par la bibliothèque standard limite ce risque.

Les résultats de ces tests n'ont pas révélé d'erreurs qui m'auraient obligé à revoir plus ou moins en profondeur mon implémention.

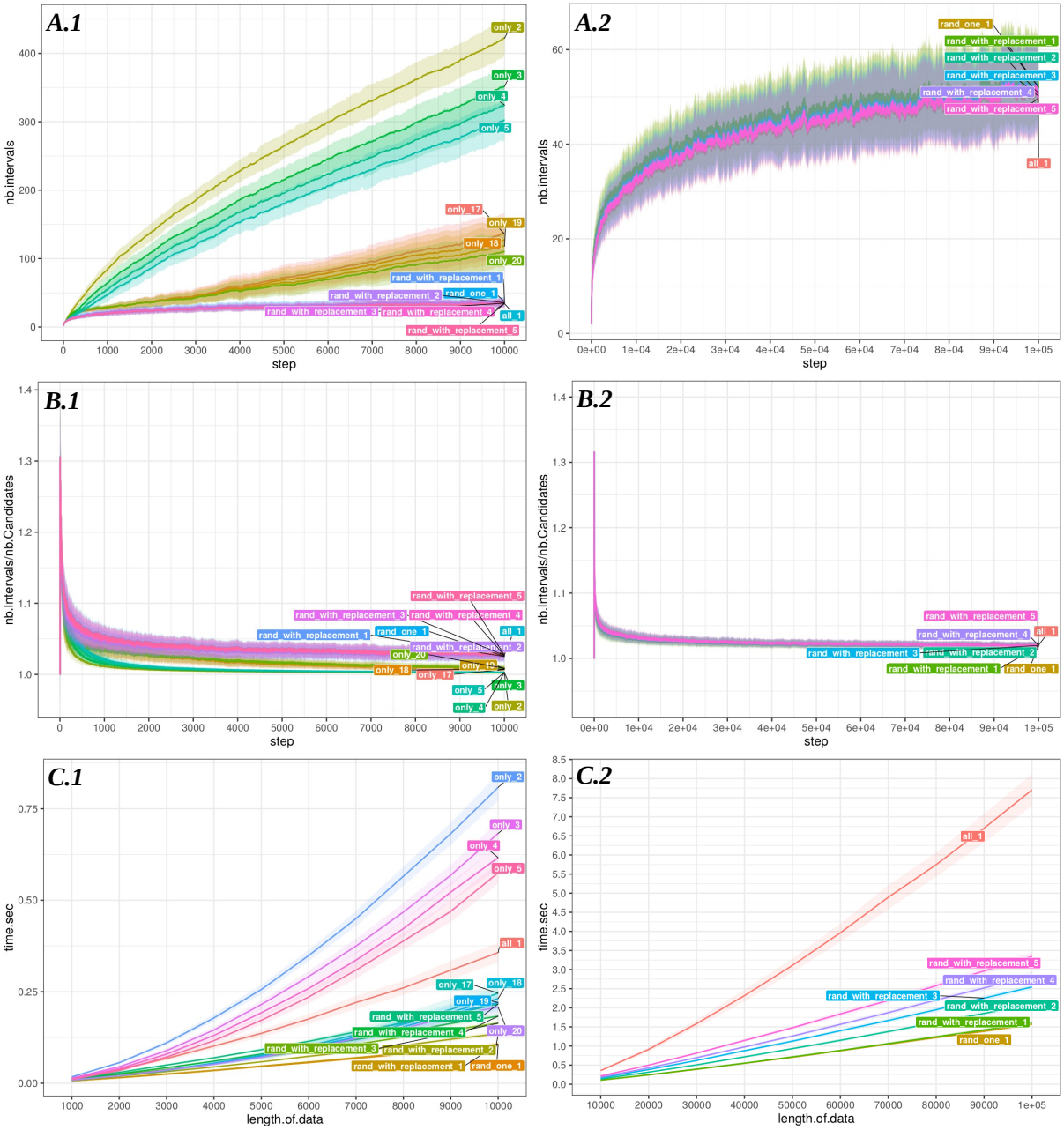


FIGURE 7 – Comparaison des méthodes d'échantillonnage only, All, Rand_with_replacement et Rand_one. 100 profils de 10^4 et 10^5 points formant un bruit Gaussien de moyenne nulle et d'écart type 1 ont été générés. ***(A.1 & A.2)*** Représentation du nombre moyen d'intervalles contenus dans l'ensemble des zones de vie des fonctions de coût des ruptures candidates à chaque étape de la segmentation. ***(B.1 & B.2)*** Représentation du nombre moyen d'intervalles sur le nombre moyen de ruptures candidates considérées à chaque étape de la segmentation. ***(C.1 & C.2)*** Représentation du temps moyen d'exécution de ***FpopPSD*** en fonction de la taille des profils générés ainsi que de la stratégie d'échantillonage utilisée. Pour ces simulations, la valeur de β et la valeur de γ ont été fixées respectivement à 2.5 et 10

4 Résultats

Dans cette section je comparerai dans un premier temps l'efficacité d'élagage des différentes stratégies d'échantillonnages des ruptures candidates futures afin de sélectionner la meilleure stratégie pour traiter les grands profils. Dans un deuxième temps, j'expliquerai comment j'ai calibré la constante γ (15) sur des données simulées avec pour objectif de comparer la capacité de détection des ruptures des méthodes ***FPOP*** et ***FpopPSD*** sur des données similaires. Immédiatement après je présenterai les résultats de ces comparaisons. En suivant la même démarche, je comparerai ***FPOP*** et ***FpopPSD*** sur des données réalistes : une première étape pour se convaincre de l'intérêt pratique de l'approche.

4.1 Comparaison des stratégies d'échantillonnage

Comme évoqué en fin de section 2.3, la complexité de chaque étape de la procédure de segmentation est en $\mathcal{O}$(nombre de ruptures candidates $\times$ complexité de la mise à jour des ruptures candidates). l'étude de h quand $g = \log$ (19) suggère qu'on peut réduire la complexité de la mise à jour des ruptures candidates tout en maintenant une efficacité d'élagage élevé. On dira des stratégies d'échantillonnages et des comparaisons qui aboutissent à ce résultat qu'elles sont plus efficientes. En réduisant la complexité de la mise à jour des ruptures candidates, on réduit la complexité de chaque étape.

En étudiant la fonction $h(t, s, s') = \log(\frac{t-s'}{t-s})$ (19), on constate :

(1) qu'elle prend de grandes valeurs quand t et s' sont proches. Dans ce cas, h défavorise fortement s' et l'intervalle $I_{t,s,s'}$ sera très grand. Il semble peu utile d'essayer de restreindre la zone de vie de s à l'aide de $I_{t,s,s'}$ quand s' à été introduite récemment ;

(2) qu'elle varie peu quand t est grand devant s et s'. Dans ce cas si l'on a comparé s et s' à l'étape t, il semble peu utile de le refaire à l'étape $t + 1$. En effet, l'intervalle $I_{t,s,s'}$ sera presque identique à $I_{t+1,s,s'}$.

À chaque étape de la segmentation, toutes les ruptures candidates et tous les intervalles formant la zone de vie de ces ruptures sont explorés. La complexité de la mise à jour des zones de vie semble donc gouvernée par le nombre d'intervalles plutôt que par le nombre de candidats. Toutefois, on peut voir sur les figures *7.B.1* et *7.B.2* que le nombre moyen d'intervalles par rupture candidate est inférieur à 1,1. Au final, étudier la complexité de mise à jour des zones de vie en terme de nombre de ruptures candidates et nombre d'intervalles est relativement proche.

Sur les figures *7.A.1* et *7.A.2*, on s'attend à ce que la stratégie qui consiste à se comparer à toutes les ruptures candidates futures soit la stratégie la plus efficace en matière d'élagage. C'est le cas de *all_1*. Plus les résultats d'une stratégie sont proches des résultats de *all_1*, plus cette stratégie est efficace. On voit dans ces deux figures que les stratégies les plus efficaces, autres que *all_1*, sont non déterministes.

Sur les figures *7.C.1* et *7.C.2*, on voit que les stratégies les plus rapides sont *rand_one_1* et *rand_with_replacement_1*. Ces stratégies font la même chose. Elles tirent aléatoirement une rupture candidate parmi les ruptures candidates futures. La complexité de mise à jour des ruptures candidates à chaque étape est fortement réduite car on effectue une seule comparaison pour chacune des ruptures candidates.

En conclusion, parmi les stratégies comparées, celle qui consiste à tirer aléatoirement une rupture candidate parmi les ruptures candidates futures semble être la plus efficiente (peu de comparaisons pour une grande efficacité d'élagage). Avec cette stratégie, on peut envisager de segmenter de très grands profils dans un temps admissible ($\sim$ 1.5sec pour 10^5 points et $\sim$ 20sec pour 10^6 points).

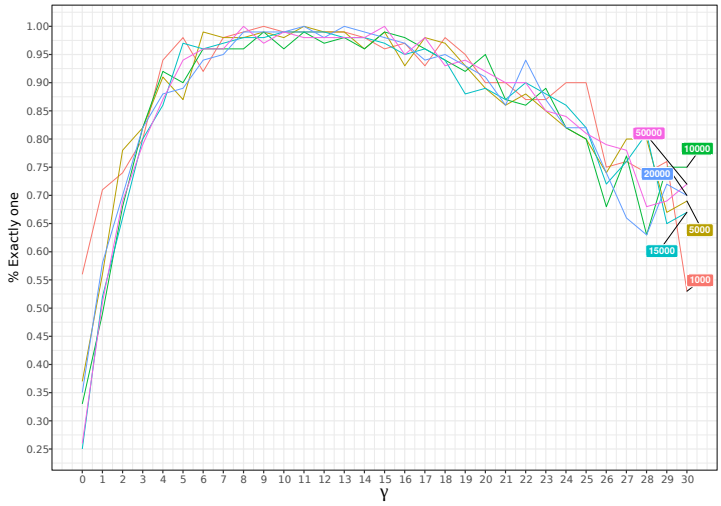


FIGURE 8 – Calibrage de la constante γ de la pénalité α (15). Des profils Gaussiens de différentes tailles (1000 à 50000 points) ont été générés avec une rupture au milieu. Chaque profil a ensuite été segmenté avec l'aide de ***FpopPSD*** pour des valeurs de γ variant entre 0 et 30. Sur 100 réplicats, j'ai comptabilisé le nombre de fois où la segmentation renvoyait une seule rupture et que cette dernière était présente dans la zone de tolérance, à savoir localisation de la vraie rupture + ou - 20 points. Pour ces deux simulations la valeur de β et de α ont été fixées respectivement à 2.5 et $\gamma + 2.5 \times log(|Y|)$

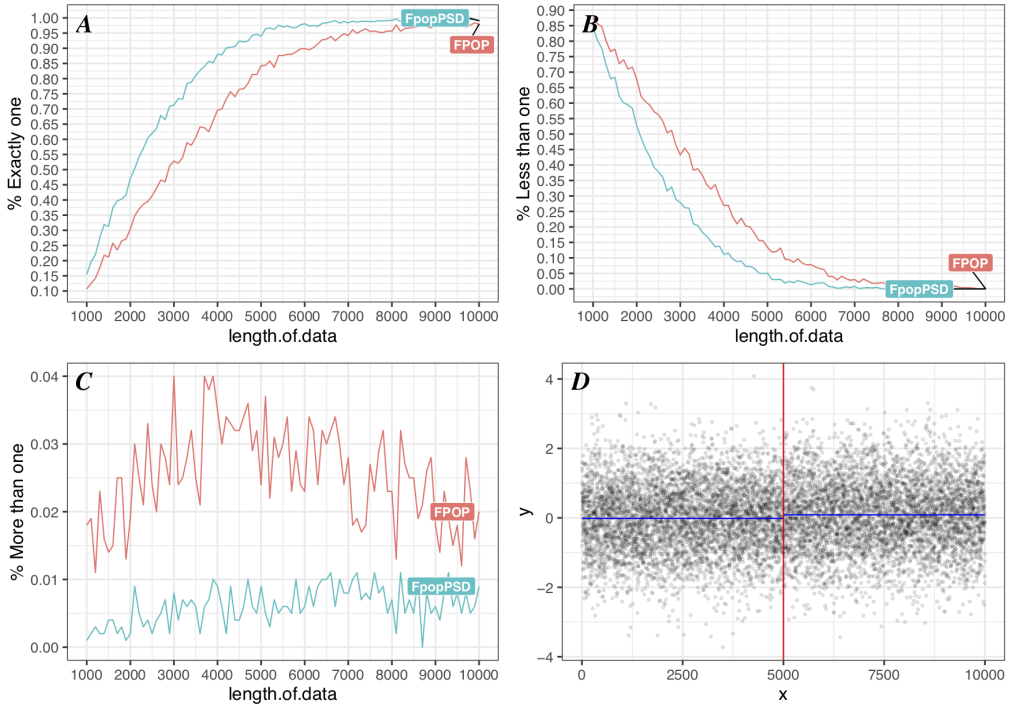


FIGURE 9 – Comparaison des méthodes ***FpopPSD*** et ***FPOP***. Des profils Gaussiens ont été générés avec une rupture au milieu. ***(A)*** Pour des tailles de profils variant entre 1000 points et 10000 points, j'ai comptabilisé sur 100 réplicats le nombre de fois où la segmentation renvoyait une seule rupture et que cette dernière était présente dans la zone de tolérance, à savoir la localisation de la vraie rupture + ou - 20 points. Sur les mêmes profils j'ai comptabilisé le nombre de fois où le nombre de ruptures renvoyées était nul ***(B)*** et supérieur à 1 ***(C)***. ***(D)*** est un exemple de profil Gaussien, avec une rupture au milieu, utilisé dans cette simulation. Pour cette simulation la valeur de β et de γ ont été fixées respectivement à 2.5 et 10 pour ***FpopPSD***. La valeur de β a été fixée à 2 pour ***Fpop***.

4.2 Calibrage de γ et comparaison de FPOP/FpopPSD sur données simulées

Comme décrit dans (15), la constante γ doit être calibrée. En effet, comme assez souvent pour les valeurs de pénalité, les résultats statistiques donnent l'existence de constantes efficaces mais pas toujours leurs valeurs. Dans cette section, j'explique comment à travers des simulations sur des données simulées *(cf. figure 8)*, je calibre cette constante. Je présenterai ensuite les résultats *(cf. figure 9)* de la comparaison des performances des méthodes ***FpopPSD*** et ***FPOP***, en matière de détection des ruptures, sur des données similaires. Pour ces simulations, la valeur de la constante β de ***FpopPSD*** (15) a été fixée à 2.5, sur la base des conseils de N. Verzelen.

4.2.1 Calibrage de γ

Le scénario imaginé est celui d'une unique rupture dans la moyenne localisée au milieu d'un profil Gaussien. J'ai ainsi segmenté à l'aide ***FpopPSD*** des profils de tailles différentes en faisant varier la valeur de la constante γ entre 0 et 30. Sur 100 réplicats, j'ai comptabilisé le nombre de fois ou la segmentation renvoyait une seule rupture présente dans une zone de tolérance librement inspirée par les travaux de Morgane et al. 2015 [27], à savoir la localisation de la vraie rupture + ou - 20 points (*ExactlyOne*). Les valeurs de γ qui nous intéressent sont celles pour lesquelles le pourcentage de *ExactlyOne* est supérieur à 95%. Le seuil est volontairement élevé au vu de la faible difficulté du scénario. À l'issue de cette première simulation, on peut voir sur la figure 8 que l'intervalle de γ pour lequel les résultats de la segmentation avec ***FpopPSD*** sont satisfaisants est $a \in [7, 15]$.

4.2.2 Comparaison FpopPSD/FPOP

En me basant sur les résultats du calibrage de γ, j'ai fixé la valeur de cette constante à 10. La constante β de ***FPOP*** (8) a quant à elle été fixée à 2, la valeur utilisée classiquement dans la littérature. Le scénario imaginé est comparable à celui utilisé pour calibrer γ. J'ai généré plusieurs profils de bruit Gaussien, de tailles différentes, avec une unique rupture dans la moyenne au milieu du profil. La différence de moyenne des segments avant et après la rupture, volontairement faible *(cf. figure 9.D)*, reste constante avec la taille des profils. Ainsi, la difficulté du profil diminue à mesure que sa taille augmente. L'ensemble de ces profils ont été segmentés à l'aide des méthodes ***FpopPSD*** et ***FPOP***. Sur 100 réplicats, pour chaque taille de profil, j'ai comptabilisé le nombre de *ExactlyOne*, le nombre de fois où le résultat de la segmentation contenait plus d'une rupture et le nombre de fois où aucune rupture n'était trouvée.

On peut voir sur la figure 9 que pour les 3 critères étudiés, la méthode ***FpopPSD*** fait au moins aussi bien que ***Fpop***. Elle surpasse cette dernière à mesure que la taille du profil diminue, donc que la difficulté de ce dernier augmente.

En conclusion, dans le cadre du scénario décrit ci-dessus, avec des paramètres favorables pour les deux méthodes, ***FpopPSD*** est meilleure que ***FPOP*** sur les données simulées.

4.3 Calibrage de γ et comparaison de FPOP/FpopPSD sur données réalistes

De la même manière que dans la section précédente j'explique comment à travers des simulations sur des données réalistes *(cf. figure 10)*, je calibre la constante γ (15). je présente ensuite les résultats *(cf. figure 11)* de la comparaison des performances des méthodes ***FpopPSD*** et ***FPOP***, en matière de détection des ruptures, sur des données similaires. Comme précédemment, pour ces simulations, la valeur de la constante β de ***FpopPSD*** (15) a été fixée à 2.5, sur la base des conseils de N. Verzelen.

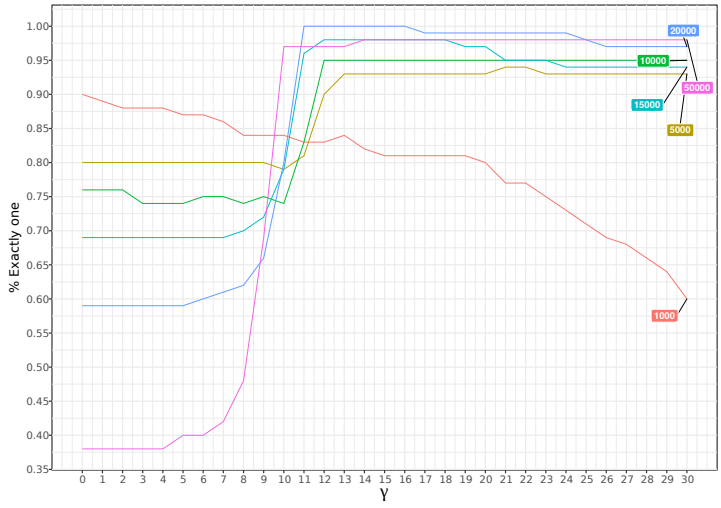


FIGURE 10 – Calibrage de la constante γ de la pénalité α (15). Des profils réalistes du nombre de copies d'ADN pour des cellules tumorales on été générés à l'aide du package R *acnr*. Le paramètre sur la fraction de cellules tumorales présentes dans l'échantillon dont a été extrait l'ADN est fixé à 1. La taille des profils varie entre 1000 et 50000 points. Dans chaque profil est présente une unique rupture dont la localisation aléatoire est connue. Chaque profil a ensuite été segmenté avec l'aide de ***FpopPSD*** pour des valeurs de γ variant entre 0 et 30. Sur 100 réplicats, j'ai comptabilisé le nombre de fois où la segmentation renvoyait une seule rupture et que cette dernière était présente dans la zone de tolérance, à savoir localisation de la vraie rupture + ou - 20 points. Pour ces deux simulations la valeur de β et de α ont été fixées respectivement à 2.5 et $\gamma + 2.5 \times log(|Y|)$

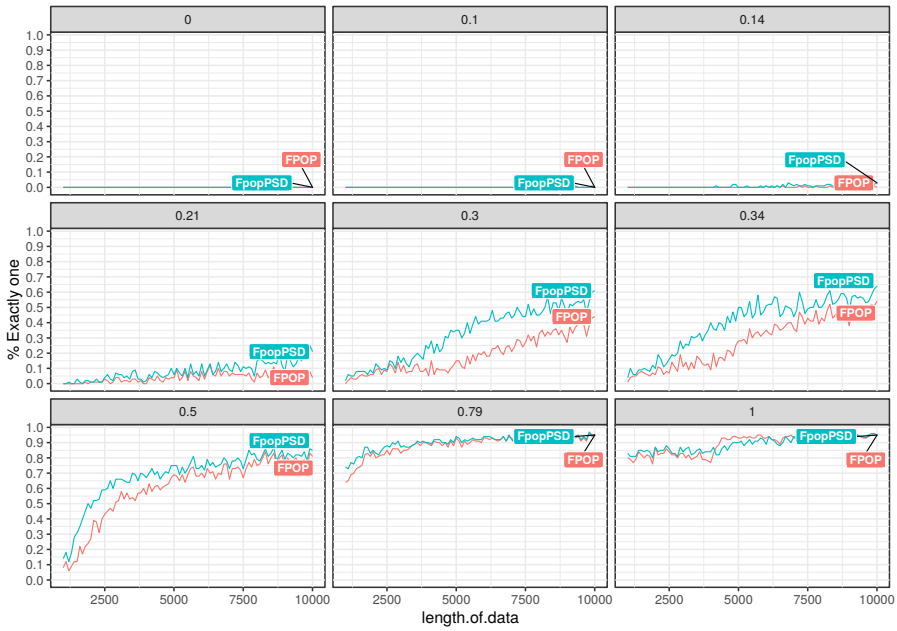


FIGURE 11 – Comparaison des méthodes ***FpopPSD*** et ***FPOP***. Des profils réalistes du nombre de copies d'ADN pour des cellules tumorales on été générés à l'aide du package R *acnr*. les profils générés ont une difficulté variable. Cette difficulté dépend de la fraction de cellules tumorales présentes dans l'échantillon dont a été extrait l'ADN (0, 0.1, 0.14, 0.21, 0.3, 0.34, 0.5, 0.79, 1). Dans chaque profil est présente une unique rupture dont la localisation aléatoire est connue. Chaque profil a ensuite été segmenté avec l'aide de ***FpopPSD*** et ***Fpop***. Pour des tailles de profils variant entre 1000 points et 10000 points, j'ai comptabilisé sur 100 réplicats le nombre de fois où la segmentation renvoyait une seule rupture et que cette dernière était présente dans la zone de tolérance, à savoir localisation de la vraie rupture + ou - 20 points. Pour cette simulation la valeur de β et de γ ont été fixées respectivement à 2.5 et 12 pour ***FpopPSD***. La valeur de β a été fixée à 3 pour ***Fpop***.

4.3.1 Calibrage de γ

Le scénario imaginé se veut plus proche de la réalité biologique que le scénario sur les données simulées et est donc plus complexe. J'ai généré, à l'aide du package R *acnr* [27], des profils réalistes du nombre de copies d'ADN pour des cellules tumorales. Ces profils sont générés par ré-échantillonnage des données réelles de SNP array qui ont été annotées par des biologistes et des bioinformaticiens. Chaque profil contient une unique rupture dans la moyenne placée aléatoirement mais dont la localisation est connue. Un des intérêts des données utilisées pour générer les profils est qu'elles ont été extraites de séries de dilutions de lignées de cellules tumorales. Le package offre ainsi la possibilité de contrôler le pourcentage de cellules tumorales présentes dans l'échantillon dont a été extrait l'ADN. Ce pourcentage contrôle lui même le rapport signal sur bruit et donc la difficulté du profil. Dans ce scénario le ratio de cellule tumorale a été fixé à 1. En reprenant la même définition de *ExactlyOne* mais en abaissant le seuil sur le pourcentage de *ExactlyOne*, on peut voir dans la figure 10 que l'intervalle de γ pour lequel les résultats de la segmentation avec ***FpopPSD*** sont satisfaisants est $a \in [12, > 30]$. En vue de la comparaison de ***FPOP*** et ***FpopPSD*** sur des données similaires, pour que les résultats soient comparables, j'ai calibré le β de la pénalité de ***FPOP*** sur les mêmes scénarios.

4.3.2 Comparaison FpopPSD/FPOP

En me basant sur les résultats du calibrage des constantes de la section précédente, j'ai fixé γ de ***FpopPSD*** à 12 et β de ***FPOP*** à 3. Comme pour le calibrage de γ, ce scénario se veut plus proche de la réalité biologique que le scénario sur les données simulées et est donc plus complexe. j'ai utilisé le package R *acnr* présenté précédemment pour générer des profils dont la taille et le paramètre sur la fraction de cellules tumorales diffèrent. Chaque profil contenait une unique rupture dans la moyenne placée aléatoirement. L'ensemble de ces profils ont été segmentés à l'aide des méthodes ***FpopPSD*** et ***FPOP***. Sur 100 réplicats pour chaque taille de profil, j'ai comptabilisé le nombre de *ExactlyOne*.

On peut voir sur la figure 11 que quand la fraction de cellules tumorales est élevée (≥ 0.79), donc que la difficulté du profil est faible, la méthode ***FpopPSD*** fait au moins aussi bien que ***FPOP***. Elle surpasse cette dernière quand la fraction de cellules tumorales diminue et donc que le profil se complexifie (le ratio signal bruit diminue).

Comme pour les données simulées, dans le cadre du scénario décrit ci-dessus, avec des paramètres favorables pour les deux méthodes, ***FpopPSD*** est meilleure que ***FPOP*** sur les données réalistes en particulier sur les profils difficiles de CNV d'*acnr*.

5 Discussion

Comme nous l'avons vu dans la section 4.1, la stratégie qui consiste à tirer aléatoirement une ruptures candidate future selon une loi uniforme est la meilleure stratégie et permet de s'attaquer à des grands profils. Il est vraisemblable que le tirage uniforme ne soit pas optimale. L'algorithme alterne entre des bons tirages (conduisant à une réduction forte de la zone de vie de la rupture candidate voir à son élagage) et des mauvais tirages (conduisant à une faible réduction de la zone de vie de la rupture candidate). En moyenne c'est suffisant. Des améliorations sont possibles. Notamment, l'intuition décrite dans la section 4.1 suggère de défavoriser des ruptures trop récentes ou des ruptures auxquelles on s'est comparé récemment. L'un des objectifs dans la suite de ce stage serait donc, sur la base de cette intuition, d'affiner cette stratégie dans le but d'améliorer les performances de ***FpopPSD*** et d'étudier sa complexité dans le pire cas.

Comme je l'évoquais dans la section 3.1, pour cette implémentation de ***FpopPSD***, j'ai largement eu recours aux classes disponibles dans la bibliothèque standard du C++. Cette dernière à l'avantage de mettre à disposition du programmeur des outils puissants et génériques. Cependant, comme me l'a suggéré V. Runge qui a implémenté plusieurs algorithmes d'élagage fonctionnel, cette généricité a un coût qui dans notre cas diminue sérieusement les performances de la méthode. Un autre objectif dans la suite du stage sera d'implémenter mes propres structures toujours dans le but d'améliorer les performances de ***FpopPSD***. Je réfléchis notamment à l'implémentation d'une structure stable qui remplacerait mon actuelle liste de ruptures candidates et le vecteur d'itérateurs qui pointent sur les éléments de cette liste.

Les profils simulés m'ont permis de montrer l'efficacité de ***FpopPSD***. Toutefois, il est évident que les simulations que j'ai réalisées pour calibrer γ et comparer ***FpopPSD*** avec ***FPOP*** sont relativement éloignées de la réalité biologique, notamment en terme de nombre et de position des ruptures. Dans la première phase de ce stage mon objectif était de montrer qu'il existe des situations pour lesquelles les performances de ***FpopPSD*** en matière de détection des ruptures étaient meilleures que celles de ***FPOP***, pour un bruit proche des données biologiques. Les résultats des sections 4.2.2 et 4.3.2 suggèrent que c'est le cas en particulier quand on augmente la difficulté des données. On peut donc espérer observer au moins les mêmes résultats, voir que cette différence s'accroisse, sur des profils encore plus complexes qui contiennent notamment plusieurs ruptures, donc des profils qui s'approchent plus de la réalité biologique. La prochaine étape est de tester d'autres scénarios plus complexes. Par la suite, via le protocole de cross validation proposé par Hocking et al. [16], je comparerai les résultats de la segmentation de ***FpopPSD*** sur les profils de variation du nombre de copies d'ADN provenant de neuroblastomes aux résultats [3] d'autres méthodes de segmentation sur ces mêmes profils. L'objectif sera alors de voir si oui ou non ***FpopPSD*** permet de réduire le nombre de ruptures non détectées, actuellement de 11.6% pour les meilleures méthodes (***FPOP*** et ***PELT***). A terme un autre objectif serait également de tester ***FpopPSD*** sur d'autres profils génomiques, notamment sur des données de RNA-seq chloroplastique en collaboration avec Etienne Delannoy (INRA) qui à pour projet de créer une base de données de profils annotés et expertisés par des biologistes et des bioinformaticiens. On pourra alors évaluer rapidement l'efficacité des approches de segmentation existantes et ***FpopPSD*** sur ces données.

6 Conclusion

J'ai participé à la conception et implémenté une extension de la méthode ***FPOP***. Cette extension, ***FpopPSD***, pénalise plus fortement les petits segments. ***FpopPSD*** est une méthode de segmentation exacte et efficace qui permet de traiter de grands profils. L'aspect algorithmique de cette méthode et le détail des classes de l'implémentation C++ sont disponibles respectivement dans les sections 2.3 et 3. Des arguments statistiques et biologiques, présentés dans la section 2.3, laissaient penser que ce type de pénalité conduirait à une meilleure segmentation des données. Je confirme via des résultats de simulations sur des profils avec un bruit Gaussien ou un bruit réaliste, que l'on peut trouver dans les sections 4.2.2 et 4.3.2, qu'il existe des situations, en particulier les profils difficiles, pour lesquelles l'utilisation ***FpopPSD*** permet de réduire le nombre de ruptures non détectées ainsi que les ruptures causées par des *outliers* par rapport à ***FPOP***. J'ai créé le package R *FpopPSD* que je déposerai à la fin de mon stage sur ForgeMIA.

Sur le plan personnel ce stage m'a permis d'approfondir mes connaissances sur la programmation dynamique et de m'initier aux techniques d'élagage fonctionnel. J'ai aussi progressé dans ma pratique du C++ via l'implémentation de la méthode *FpopPSD* en utilisant la programmation orientée objet.

3. `http://members.cbio.mines-paristech.fr/~thocking/neuroblastoma/accuracy.html`

7 Références

[1] Redon R et AL. "Global variation in copy number in the human genome." In : *Nature.* 444 (nov. 2006), p. 444–54.

[2] Zarrei M et AL. "A copy number variation map of the human genome." In : *Nat Rev Genet.* 16 (février 2015), p. 172–83.

[3] Donald F. Conrad et AL. "Origins and functional impact of copy number variation in the human genome." In : *Nature.* 464 (avril 2010), p. 704–12.

[4] Adam Shlien et AL. "Copy number variations and cancer". In : *Genome Med.* 1 (juin 2009), p. 62.

[5] Maki Fukami et AL. "Next generation sequencing and array-based comparative genomic hybridization for molecular diagnosis of pediatric endocrine disorders". In : *Ann Pediatr Endocrinol Metab.* 22 (juin 2017), p. 90–4.

[6] Forozan F et AL. "Genome screening by comparative genomic hybridization." In : *Trends Genet.* 13 (oct. 1997), p. 405–9.

[7] Michelle Gaasenbeek et AL. "Combined Array-Comparative Genomic Hybridization and Single-Nucleotide Polymorphism-Loss of Heterozygosity Analysis Reveals Complex Changes and Multiple Forms of Chromosomal Instability in Colorectal Cancers." In : *Cancer Research.* 66 (avril 2006).

[8] Kloth JN et AL. "Combined array-comparative genomic hybridization and single-nucleotide polymorphism-loss of heterozygosity analysis reveals complex genetic alterations in cervical cancer." In : *BMC Genomics.* (février 2007).

[9] Andrew G Evans et AL. "Combined comparative genomic hybridization and single-nucleotide polymorphism array detects cryptic chromosomal lesions in both myelodysplastic syndromes and cytopenias of undetermined significance." In : *Modern Pathology.* 29 (juillet 2016), p. 1183–99.

[10] Ichiro Nakachi et AL. "Application of SNP microarrays to the Genome-wide Analysis of Chromosomal Instability in Premalignant Airway Lesions." In : *Cancer Prev Res.* 7 (décembre 2013), p. 255–65.

[11] Olshen AB et AL. "Circular binary segmentation for the analysis of array-based DNA copy number data." In : 5 (oct. 2004), p. 557–72.

[12] F. Picard et AL. "A statistical approach for array CGH data analysis." In : *BMC Bioinformatics.* 6 (février 2005).

[13] Vincent Brault et AL. "Nonparametric multiple change-point estimation for analyzing large Hi-C data matrices." In : *Journal of Multivariate Analysis.* 165 (mai 2018), p. 143–65.

[14] Xing H et AL. "Genome-wide localization of protein-DNA binding and histone modification by a Bayesian change-point method with ChIP-seq data." In : *PLoS Comput Biol.* (juillet 2012).

[15] Zhang J et AL. "An empirical Bayes change-point model for identifying 3' and 5' alternative splicing by next-generation RNA sequencing." In : *Bioinformatics.* 31 (juin 2016), p. 1823–31.

[16] Hocking et AL. "Learning smoothing models of copy number profiles using breakpoint annotations." In : 14 (mai 2013).

[17] Robert Maidstone et AL. "On optimal multiple changepoint algorithms for large data." In : *Statistics and Computing.* 27 (mar. 2017), p. 519–33.

[18] R. Killick et AL. "Optimal detection of changepoints with a linear computational cost." In : *Statistics and Computing.* 107 (mar. 2012), p. 1590–98.

[19] F. Picard et AL. "A Segmentation/Clustering Model for the Analysis of Array CGH Data." In : *Biometrics.* 63 (sept. 2007), p. 758–66.

[20] Yi-Ching Yao et AL. "Least-Squares Estimation of a Step Function." In : *Indian Journal of Statistics.* 51 (nov. 1989), p. 370–81.

[21] Émilie LEBARBIER. "Detecting multiple change-points in the mean of a Gaussian process by model selection." In : *Signal Proces.* 87 (avril 2005), p. 717–36.

[22] Brad Jackson et AL. "An Algorithm for Optimal Partitioning of Data on an Interval." In : *IEEE Signal Processing Letters.* 12 (jan. 2005), p. 105–8.

[23] Auger et AL.. "Algorithms for the optimal identification of segment neighborhoods." In : *Bull Math Biol.* 51 (1989), p. 39–54.

[24] R. BELLMAN. "On the approximation of curves by line segments using dynamic programming." In : *Communications of the ACM.* 4 (1961), p. 284.

[25] Guillem Rigaill et AL. "A pruned dynamic programming algorithm to recover the best segmentations with 1 to Kmax change-points." In : *Brief Bioinform.* 156 (avril 2010), p. 180–205.

[26] Johnson et AL. "A Dynamic Programming Algorithm for the Fused Lasso and L0-Segmentation." In : *Journal of Computationaland Graphical Statistics.* 2 (2013), p. 246–260.

[27] Morgane Pierre-Jean et AL. "Performance evaluation of DNA copy number segmentation methods." In : *Brief Bioinform.* 16 (juillet 2015), p. 600–15.

8 Annexes

Lemme : La règle (21) valide (17)

Soit la propriété

$$P_t : \forall\ k \geq 0,\ Z_{t,s} \supset Z^*_{t+k,s}\ .$$

P_s est vraie en définissant

$$Z_{s,s} = [min(Y), max(Y)]\ .$$

On suppose que P_t est vraie. On vérifie que P_{t+1} l'est aussi.

(a) $Z_{t,s} \supset Z^*_{t+1+k,s}$ par hypothèse de récurrence.

(b) $\bigcap_{s'} I_{t+1,s,s'} \supset Z^*_{t+1+k,s}$. On déduit de la propriété (20) que chaque intervalle $I_{t+1,s,s'}$ inclut $Z^*_{t+1+k,s}$ et l'intersection d'intervalles qui incluent $Z^*_{t+1+k,s}$ inclut elle-même $Z^*_{t+1+k,s}$.

(c) $\bigcup_{s''} I_{t\to\infty,s'',s} \cap Z^*_{t+1+k,s} = \emptyset$. On déduit de la propriété (20) que chaque intervalle $I_{t\to\infty,s'',s}$ est d'intersection vide avec $Z^*_{t+1+k,s}$ et l'union d'intervalle d'intersection vide avec $Z^*_{t+1+k,s}$ est elle-même d'intersection vide avec $Z^*_{t+1+k,s}$.

On intersecte des ensembles qui incluent $Z^*_{t+1+k,s}$ auxquels on retranche une union d'intervalles d'intersections vide avec $Z^*_{t+1+k,s}$ donc,

$$Z_{t+1,s} \supset Z^*_{t+1+k,s}$$

Code R qui montre que la pénalité sur la longueur des segments n'empêche pas de détecter des petits segments :

```
y <- c(rnorm(100), rnorm(10000)+1)
getP <- function(i, y){
  n <- length(y)
  2.5*log(i)+ 2.5*log(n-i)
}
getC <- function(i, y){
  n <- length(y);
  var(y[1:i])*(i-1) + var(y[(i+1):n])*(n-i-1)
}
n <- length(y)
cost <- sapply(1:(n-1), FUN=getC, y=y)
pen <- sapply(1:(n-1), FUN=getP, y=y)
matplot(cbind(cost, cost-pen), type="l")
```