Une approche computationnelle pour déterminer le site oriC de Borrelia burgdorferi B31

Kenan OGGAD[1-n]

[1] Université Paris-Saclay, LDD biologie et informatique — 20230732

1 Introduction

La réplication de l'ADN est l'un des processus ubiquitaires du vivant. Elle permet de dupliquer l'information génétique de manière à ce que chaque cellule puisse réaliser sa fonction dans l'organisme.
Ce processus assure la transmission fidèle de l'information génétique lors de la division cellulaire, permettant à tous les organismes vivants de prospérer.

1.1 Contexte de la réplication de l'ADN chez les procaryotes

Chez les organismes eucaryotes la réplication se fait simultanément, sur plusieurs sites et de manière bidirectionnelle. Et ce, afin de répondre à la grande taille de leur génome linéaire.
À la différence des eucaryotes, les procaryotes possèdent un génome circulaire et une seule origine de réplication, connue sous le nom d'oriC.

Cette singularité impose des contraintes sur le processus de réplication, mais elle est compensée par la compacité du génome bactérien, ce qui garantit une transmission rapide de l'information génétique lors de la division cellulaire.

1.2 Importance de l'étude de Borrelia dans la recherche médicale

Dans le cadre de cette étude, nous nous intéressons spécifiquement à Borrelia, un genre de bactéries spirochètes, agent éthiologique de maladies telles que la maladie de Lyme. Malgré leur importance clinique, peu d'informations sont disponibles sur les mécanismes de réplication de l'ADN chez les espèces de Borrelia.
L'analyse approfondie de la séquence génomique de Borrelia peut fournir des données cruciales sur son origine de réplication et sur les mécanismes sous-jacents à sa réplication de l'ADN.

Comprendre ces mécanismes pourrait non seulement éclairer la biologie fondamentale de cette bactérie, mais aussi ouvrir de nouvelles voies pour le développement de thérapies ciblées contre les maladies qu'elle provoque.

1.3 Outils de bioinformatique pour l'analyse de la séquence génomique

L'analyse du génome de Borrelia nécessite l'utilisation d'outils spécialisés.

En particulier, les outils de bioinformatique offrent la possibilité de parcourir la séquence génomique de Borrelia en suivant des algorithmes prédéfinis.
Ces méthodes permettent ainsi de détecter des motifs associés à l'origine de réplication par l'usage de plusieures techniques — dont nous détaillerons les spécificités dans la suite de l'article.

Ces outils peuvent également nous aider à comparer la séquence génomique de Borrelia avec celles d'autres organismes, ce qui nous permettra d'arriver à une conclusion plus robuste grace au croisement des données.

1.4 Objectifs de l'étude sur la séquence génomique de Borrelia

Ainsi, cette étude vise à caractériser l'origine de réplication de Borrelia à travers une analyse approfondie de sa séquence génomique.

Ce faisant, nous espérons contribuer à une meilleure compréhension des processus de réplication de l'ADN chez cette bactérie.

2 Materiels et Méthodes

2.1 Données génomiques et format FASTA

Nous disposons du génome complet de Borrelia reccupéré sur le site du NCBI (National Center for Biotechnology Information) au format FASTA totalisant 910724 pb.
Le format FASTA (Fast Alignment and Search Tool) est l'un des formats de fichiers les plus couramment utilisés en bioinformatique pour stocker et échanger des séquences d'ADN, d'ARN ou de protéines.

Le format FASTA est relativement simple et facile à comprendre. Chaque séquence est représentée par un header qui commence par le caractère ">" suivi d'une description de la séquence sur une seule ligne. Ensuite, la séquence elle-même est écrite sur une ou plusieurs lignes, géneralement sans espaces, tabulations ou retours à la ligne dans le cas des séquences longues.

Sa simplicité en fait un format particulièrement bien exploitable en bioinformatique car il est pris en charge par de nombreux logiciels.
Contrairement au format genebank par exemple qui necessite des outils un peu plus avancés comme le regex (regular expression) pour reussir à le parser correctement (from scratch).

2.2 Environnement de developpement

Au cours de ce projet nous avons utilisé trois languages de programmation ainsi que deux IDE (integrated developpement environment):

- VScode (python et C) pour les algorithmes bioinformatiques.
- Rstudio (R) pour la modélisation graphique des résultats.

Nous avons également utilisé Debian dans WSL (Windows Subsystem for Linux) pour utiliser VScode plus efficacement.
Le code de tous les programmes est disponible en annexe par ordre d'apparition.

2.3 Avantages et inconvénients de Python

Python est un language interprété haut niveau qui met l'emphase sur la facilité d'accès de sa syntaxe se rapprochant d'avantage du language naturel.
Elle supprime délimitations de code comme le ";" au profit de l'indentation, la mémoire est gérée automatiquement, les erreurs sont facilement compréhensibles et le but assumé est de faciliter chaque aspects du developpement.
Au delà de son accessibilité la force de python réside en sa capacité à faire beaucoup avec peu, il possède des librairies complètes pour une quantité d'applications et sa communauté est très active.
Cependant par sa nature de language interprété python peut vite perdre en performance par rapport à ses homologues plus bas niveau.

2.4 Avantages et inconvénients de C

Contrairement à python, C est un language compilé plus bas niveau, c'est à dire plus proche de la machine. Il est donc beaucoup plus efficient et rapide. Il permet une manipulation précise des ressources de la machine, la mémoire est allouée manuellement, le code obéit à une syntaxe stricte et les erreurs peuvent parfois être plus compliquées à comprendre. Beaucoup de languages modernes ou de kernel de systèmes d'exploitations sont basés sur C car il permet une grande liberté.
Cependant C est un language vieillissant au mieux, obsolète au pire à cause de languages qui font à peu près tout mieux que lui comme C++ ou Rust.

2.5 Avantages et inconvénients de R

R est un language de programmation spécialement adapté à la modélisation statistique.
R est accompagné de nombreux outils de visualisation de données, tels que ggplot2 — que nous utiliserons par la suite — qui permettent de créer des graphiques et des visualisations complexes avec relativement peu de code.
Cette capacité à représenter graphiquement les données est essentielle pour l'exploration et la communication de résultats dans de nombreuses sciences.
Cependant les performances de R sont faibles pour de grandes quantités de données (même si data.table par exemple peut améliorer les performances) et sa syntaxe et son IDE associée (Rstudio) sont assez vétustes.

2.6 Technique de la fenètre glissante

La technique de la fenêtre glissante — ou moyenne mobile — permet de réveler des tendances sur des jeux de données ordonés en limitant le bruit.
Le principe est simple et il est médié par deux paramètres que sont la *taille* et le *pas*.

La taille correspond à la longueur de notre fenêtre. *i.e*: Le nombre de points de données que l'on va considérer par fenêtre. (Ici un point de donnée correspond à un nucléotide.)
Le pas correspond au nombre de point de données dont va progresser la fenêtre à chaque itération. *e.x*: Un pas de 2 décalera la fenêtre de deux points de données dans la direction de la sequence, puis de 2 autres à l'iteration suivante etc...

Le recouvrement des fenêtres (tant que le pas est inférieur à la taille de fenêtre) permet d'homogeneiser les données.
Le but etant de faire varier ces paramètres afin d'obtenir un bon compromis entre la lisibilité et la précision des données. (Après un grand nombre d'essai-erreur j'ai arrêté ma décision sur une taille de fenêtre $F = 5000$ et de pas $P = 10000$, nous utiliserons ces paramètres pour le reste de l'article)

3 Résultats

3.1 Taux de GC

Le taux de GC correspond à la proportion de guanine et de cytosine dans une séquence donnée, analyser le taux de GC et le biais de GC va nous permettre de déterminer une potentielle origine de replication.
Une des caractéristiques de la replication procaryote est qu'elle sépare l'ADN en deux brins qui sont traités differemment, là où le brin primaire est synthétisé en continu le brin secondaire est synthétisé de manière discontinue (primase>polymerase>ligase) ce qui crée des fragments d'Okazaki. Cette manière de synthétiser le génome produit plus de bases A et T (car elles ne partagent que deux liaisons hydrogènes ce qui rend la liaison plus faible). On a donc une différences de proportion entre les couples de bases (la loi de Chargaff dit que dans une séquence d'ADN A=T et C=G, mais ce qui ne veut pas dire que A+T=C+G), on peut donc analyser cette différence pour indirectement retrouver l'origine de replication.

Le taux de GC est défini par la formule:

$$GC_{content} = \frac{(G+C)}{(A+T+C+G)} * 100$$

Table 1
Extrait des résultats sur une section très courte

Position	Content
80000	15.58%
90000	18.88%
100000	13.92%
110000	14.32%
120000	16.30%
130000	17.00%
140000	14.44%
150000	15.92%
160000	20.20%
170000	17.20%
180000	18.44%
190000	18.94%

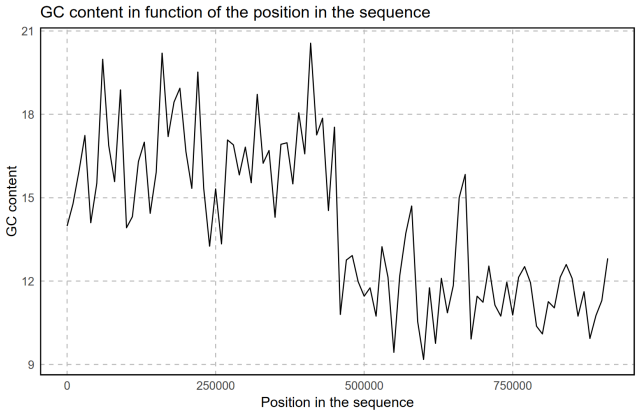

Figure 1. Taux de GC (%)

On constate un effondrement du taux de GC entre la position 450000-500000.

3.2 Biais de GC

Le biais de GC est défini par la formule:

$$GC_{bias} = \frac{(G-C)}{(G+C)}$$

Table 2
Extrait des résultats sur une section très courte

Position	Biais
480001	-0.1004
490001	-0.3110
500001	-0.2659
510001	-0.3079
520001	-0.2901
530001	-0.2887
540001	-0.2438
550001	-0.2588
560001	-0.2637
570001	-0.2108
580001	-0.0216

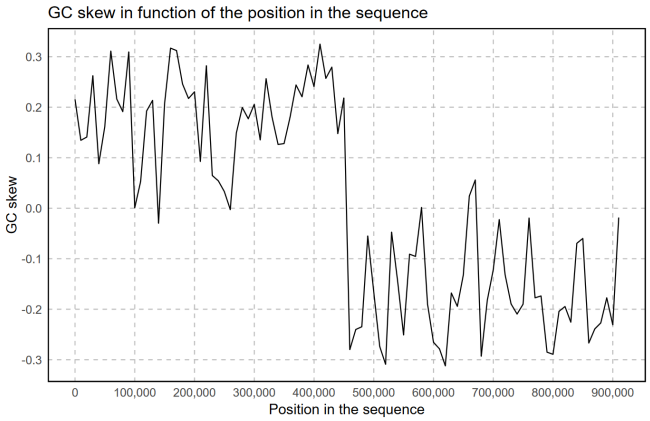

Figure 2. Biais de GC

Similairement on constate un effondrement du biais de GC entre la position 450000-500000.

3.3 Courbe en Z

La courbe en Z est définie par le système d'équation suivant:

$$Z_{curve} = \begin{cases} \frac{A_n}{len} = x_n \\ \frac{G_n}{len} = y_n \\ \frac{C_n}{len} = z_n \end{cases}$$

Avec X_n la quantité du nucléotide X à la position n et len la longueur de la séquence.

Table 3
Extrait des résultats sur une section très courte

x	y	z
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955
0.2407	0.1079	0.0955

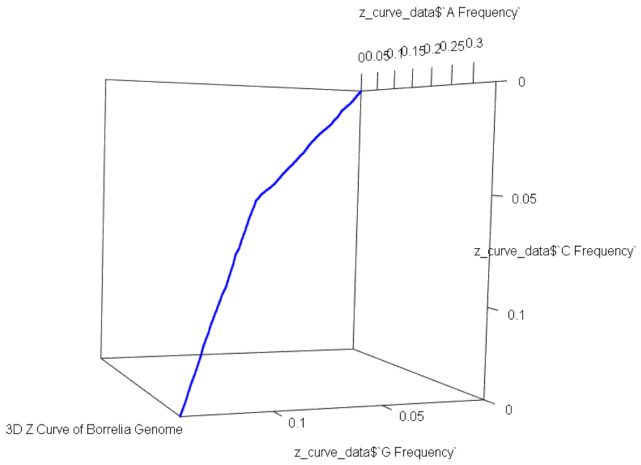

Figure 3. Courbe en Z - vue 1

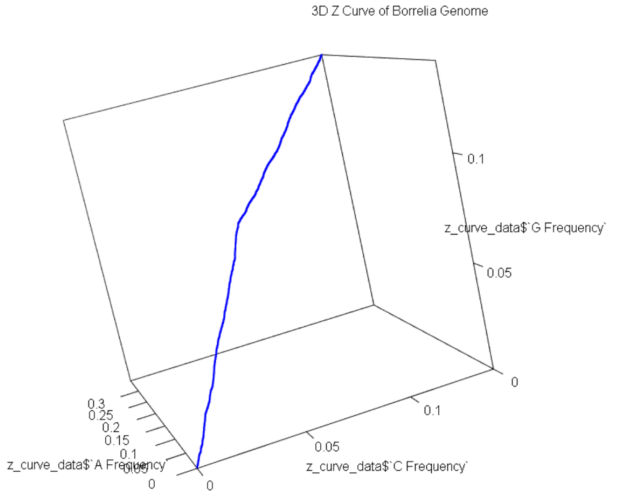

Figure 4. Courbe en Z - vue 2

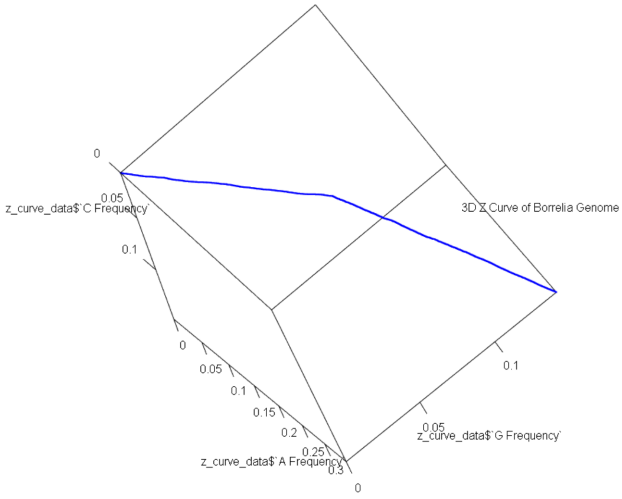

Figure 5. Courbe en Z - vue 3

On remarque un point d'inflexion clair vers le centre de la séquence ce qui coincide avec nos résultats de taux de GC et de biais de GC (à environ 470000 pb).

3.4 Marche sur l'ADN

La marche sur l'ADN peut être définie par le système d'équation suivant:

$$DNA_{walk} = \begin{cases} \frac{A_n - T_n}{len} = x_n \\ \frac{G_n - C_n}{len} = y_n \end{cases}$$

Avec X_n la quantité du nucléotide X à la position n et len la longueur de la séquence.

Table 4
Extrait des résultats sur une section très courte

x	y
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128
-0.0172	0.0128

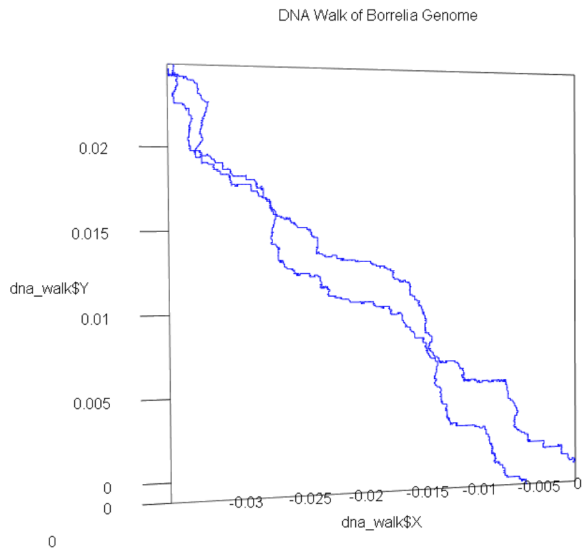

Figure 6. Marche sur l'ADN - vue 1

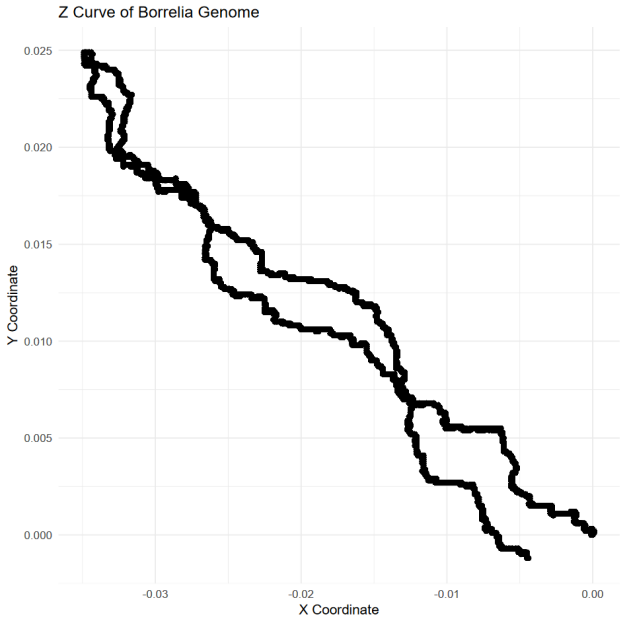

Figure 7. Marche sur l'ADN - vue 2 (voir dna walk badly labeled.pdf dans les documents bonus)

On remarque un changement de direction unique de la marche sur l'ADN en haut à gauche du graphe ce qui coincide avec notre point d'inflexion unique de la courbe en Z et notre biais et taux de GC. Nous pouvons emettre l'hypothèse qu'il s'agisse de l'emplacement de l'oriC de Borrelia.

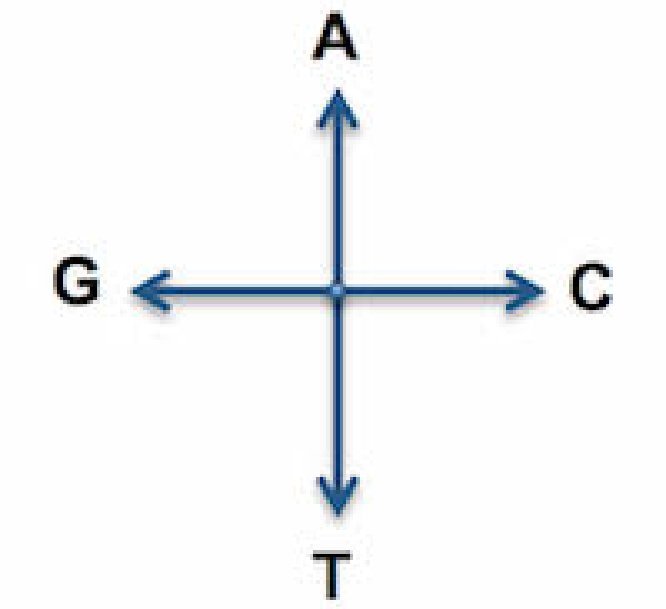

Figure 8. Marche sur l'ADN - règle

4 Discussion

Grace au travail préliminaire réalisé sur les chromosomes de poulet (où nous avions pu observer une tendance décroissante du taux de GC au plus la taille du chromosome est grande en vertu du biais d'usage du code qui favorise l'usage de certains codons plus stables) nous avons pu nous familiariser avec le calcul d'un taux de gc et l'usage de R.

Pour le cas de Borrelia, en combinant ces 4 méthodes nous avons pu nous faire une idée de la localisation de son origine de réplication grace aux propriétés énoncées en 3.1 et à la modélisation statistique.

Afin de m'assurer de la validité de ces résultats j'ai appliqué chacunes de ces méthodes à C.elegans (dont je me suis procuré le génome sur le site du NCBI) sans que les résultats ne soient facilement interprétables, la variation de taille de génome nécessite de totalement recalibrer la fenêtre glissante et les résultats graphiques ne sont pas aussi facilement interprétables (bien que la courbe en Z nous donne une idée plus claire que les autres méthodes).

Le taux de gc peut vouloir dire plusieures choses concernant un organisme, une hypothèse dépassée aujourd'hui voulait que la stabilité thermique soit l'influence principale du taux de gc dans un organisme. Dans les faits la selection naturelle à elle seule ne peut expliquer de telles variations car elle produirait des SNPs qui sont souvent silentieuses.
Beaucoup d'oiseaux ont un taux de GC très élevé, l'hypothèse actuelle est que ces changement sont dus à la gBGC (gc biased gene conversion) limitée au eucaryotes où le taux de gc augmente à cause de conversions de genes durant la recombination de génome (à cause de mécanismes spécifiques comme la déamination methyl-cytosine... etc).

Ce qui peut questionner sur l'évolution des organismes eucaryotes vers un taux de gc plus élevé comme conséquence directe de leur machinerie cellulaire, peut-être la plus grande stabilité de la liaison C-G tend naturellement à être favorisée pour conserver plus facilement les caractères qui ont survécu au "Lindy effect" ? Pour l'instant nous n'en savons rien.

Nos résultats sont à remettre en perspective, ici notre étude s'est concentrée sur le k-mère = 1 mais il existes d'autres manières d'évaluer le taux de gc pour k=2, 3 ou 4.

Aussi il semble ne pas y avoir d'uniformité d'évaluation inter-espèces du taux de gc et de ses conséquences. Chaque organismes à ses spécificités et son/ses origine(s) de réplication lui sont propres.

5 Annexes et programmes

5.1 Taux de GC programme

```c
#include <stdio.h>    // Includes the necessary libraries for input/output
#include <stdlib.h>   // Includes the standard library for memory allocation functions
#include <string.h>   // Includes the library for manipulating strings
#include <ctype.h>    // Includes the library for character manipulation functions

// Function to calculate the percentage of GC content in a sequence portion
double gc_content(char *seq, int start, int end) {
    int gc_count = 0; // Initializes the counter for G and C bases to 0
    // Traverses the sequence from the starting position to the ending position
    for (int i = start; i <= end; i++) {
        if (tolower(seq[i]) == 'g' || seq[i] == 'c') { // Checks if the base is G or C
            gc_count++;    // Increments the counter for G and C bases
        }
    }
    // Calculates the percentage of GC content and returns it
    return (double)gc_count / (end - start + 1) * 100;
}

// Main function of the program
int main(void) {
    char *seq;        // Pointer to store the DNA sequence
    int width, step;  // Variables for the window width and step size
    FILE *fp_in, *fp_out; // File pointers for input and output files

    // Opens the input file in read mode
    fp_in = fopen("Borrelia_burgdorferi_B31_complete_genome.txt", "r");
    // Opens the output file in write mode
    fp_out = fopen("GC_output.txt", "w");

    // Determines the size of the sequence by counting the total number of characters in
        the input file
    fseek(fp_in, 0, SEEK_END);
    long file_size = ftell(fp_in);
    fseek(fp_in, 0, SEEK_SET);

    // Allocates memory to store the DNA sequence based on the file size
    seq = (char *)malloc((file_size + 1) * sizeof(char));
    // Reads the DNA sequence from the input file and stores it in the allocated memory
    fread(seq, sizeof(char), file_size, fp_in);
    seq[file_size] = '\0'; // Adds a null character at the end of the sequence to
        terminate it

    // Reverses the DNA sequence by swapping characters from each end
```

```
    int len = strlen(seq);
    for (int i = 0; i < len / 2; i++) {
        char temp = seq[i];
        seq[i] = seq[len - i - 1];
        seq[len - i - 1] = temp;
    }

    // Asks the user to enter the window width and step size
    printf("Enter the window width: ");
    scanf("%d", &width);
    printf("Enter the step size: ");
    scanf("%d", &step);

    // Traverses the DNA sequence with a sliding window and calculates the GC content
    for (int i = 0; i < len; i += step) {
        int end = i + width - 1;
        if (end >= len) {
            break; // Exits the loop if the end of the sequence is reached
        }
        double avg = gc_content(seq, i, end); // Calls the gc_content function to calculate
            the GC content
        // Displays the moving average at the current position
        printf("Moving average at position %d: %.2f%% GC\n", i, avg);
        // Writes the position and GC content to the output file
        fprintf(fp_out, "%d \t %.2f%% \n", i, avg);
    }

    // Frees dynamically allocated memory for the DNA sequence
    free(seq);
    // Closes the input and output files
    fclose(fp_in);
    fclose(fp_out);
    // Returns 0 to indicate successful execution of the program
    return 0;
}
/* COMMANDES EN R
my_data <- read.table("GC_output.txt", header = FALSE)
print(my_data)

my_data$V2 <- as.numeric(sub("%", "", my_data$V2))
library(ggplot2)

ggplot(my_data, aes(x = V1, y = V2)) +
  geom_line(color = "black") +
  labs(title = "GC content in function of the position in the sequence",
       x = "Position in the sequence",
       y = "GC content") +
  theme_minimal() +
```

```
  theme(panel.border = element_rect(color = "black", fill = NA, size = 1),
        panel.grid.major = element_line(color = "gray", linetype = "dashed"),
        panel.grid.minor = element_blank(),
        plot.margin = margin(1, 1, 1, 1, "cm"),
        axis.title = element_text(size = 12),
        axis.text = element_text(size = 10),
        plot.title = element_text(size = 14))
*/
```

5.2 Biais de GC programme

```
   #include <stdio.h>    // Includes the necessary libraries for input/output
#include <stdlib.h>   // Includes the standard library for memory allocation functions
#include <string.h>   // Includes the library for manipulating strings
#include <ctype.h>    // Includes the library for character manipulation functions

// Function to calculate the GC skew in a sequence portion
double gc_skew(char *seq, int start, int end) {
    int g_count = 0, c_count = 0;
    // Counts the occurrences of G and C bases in the sequence portion
    for (int i = start; i <= end; i++) {
        if (tolower(seq[i]) == 'g') {
            g_count++;
        } else if (tolower(seq[i]) == 'c') {
            c_count++;
        }
    }
    // Calculates the GC skew and returns it
    return (double)(g_count - c_count) / (g_count + c_count);
}

// Main function of the program
int main(void) {
    char *seq;        // Dynamic allocation for the DNA sequence
    int width, step;  // Variables for the window width and step size
    FILE *fp_in, *fp_out; // File pointers for input and output files

    // Opens the input file in read mode
    fp_in = fopen("Borrelia_burgdorferi_B31_complete_genome.txt", "r");
    // Opens the output file in write mode
    fp_out = fopen("GC_skew_output.txt", "w");
    // Checks if opening files failed
    if (fp_in == NULL || fp_out == NULL) {
        printf("Error: Unable to open file(s).\n");
        return 1;
    }
```

```
// Determines the size of the sequence file
fseek(fp_in, 0, SEEK_END);
long file_size = ftell(fp_in);
fseek(fp_in, 0, SEEK_SET);

// Allocates memory for the DNA sequence
seq = (char *)malloc((file_size + 1) * sizeof(char));
// Checks if memory allocation failed
if (seq == NULL) {
    printf("Memory allocation failed.\n");
    return 1;
}

// Reads the DNA sequence from the file
fread(seq, sizeof(char), file_size, fp_in);
seq[file_size] = '\0'; // Null-terminate the sequence

// Reverses the DNA sequence by swapping characters from each end
int len = strlen(seq);
for (int i = 0; i < len / 2; i++) {
    char temp = seq[i];
    seq[i] = seq[len - i - 1];
    seq[len - i - 1] = temp;
}

// Asks the user to enter the window width and step size
printf("Enter the window width: ");
scanf("%d", &width);
printf("Enter the step size: ");
scanf("%d", &step);

// Warns if the sequence length is not divisible by the window width
if (len % width != 0) {
    printf("The last %d nucleotides will be ignored.\n", len % width);
}

// Traverses the DNA sequence with a sliding window and calculates the GC skew
for (int i = 0; i < len; i += step) {
    int end = i + width - 1;
    if (end >= len) {
        break;
    }
    double skew = gc_skew(seq, i, end);
    // Prints the GC skew in the current window
    printf("GC skew in window starting at position %d: %.4f\n", i, skew);
    // Writes the position and GC skew to the output file
    fprintf(fp_out, "%d \t %.4f\n", i, skew);
}
```

```
    // Frees dynamically allocated memory for the DNA sequence
    free(seq);
    // Closes the input and output files
    fclose(fp_in);
    fclose(fp_out);
    // Returns 0 to indicate successful execution of the program
    return 0;
}
/* COMMANDES EN R
my_data <- read.table("GC_skew_output.txt", header = FALSE)
print(my_data)

ggplot(my_data, aes(x = V1, y = V2)) +
  geom_line(color = "black") +
  labs(title = "GC skew in function of the position in the sequence",
       x = "Position in the sequence",
       y = "GC skew") +
  scale_x_continuous(breaks = pretty_breaks(n = 10), labels = comma) +
  scale_y_continuous(breaks = pretty_breaks(n = 10)) +
  theme_minimal() +
  theme(panel.border = element_rect(color = "black", fill = NA, size = 1),
        panel.grid.major = element_line(color = "gray", linetype = "dashed"),
        panel.grid.minor = element_blank(),
        plot.margin = margin(1, 1, 1, 1, "cm"),
        axis.title = element_text(size = 12),
        axis.text = element_text(size = 10),
        plot.title = element_text(size = 14))
*/
```

5.3 Courbe en Z programme

```
    #include <stdio.h>    // Includes the necessary libraries for input/output
#include <stdlib.h>   // Includes the standard library for memory allocation functions
#include <string.h>   // Includes the library for manipulating strings
#include <ctype.h>    // Includes the library for character manipulation functions

// Function to calculate and output the Z curve data
void calculate_z_curve(char *seq, int len, FILE *fp_out) {
    int a_count = 0, g_count = 0, c_count = 0;

    // Counts the occurrences of A, G, and C nucleotides in the sequence
    for (int i = 0; i < len; i++) {
        char nucleotide = toupper(seq[i]); // Convert to uppercase for consistency
        switch (nucleotide) {
            case 'A':
                a_count++;
```

```
                break;
            case 'G':
                g_count++;
                break;
            case 'C':
                c_count++;
                break;
            default:
                break;
        }
    }

    // Outputs the numerical values for plotting the Z curve
    for (int i = 0; i < len; i++) {
        double x = (double)a_count / len;
        double y = (double)g_count / len;
        double z = (double)c_count / len;
        fprintf(fp_out, "%.4f\t%.4f\t%.4f\n", x, y, z);

        char nucleotide = toupper(seq[i]);
        switch (nucleotide) {
            case 'A':
                a_count--;
                break;
            case 'G':
                g_count--;
                break;
            case 'C':
                c_count--;
                break;
            default:
                break;
        }
    }
}

// Main function of the program
int main(void) {
    char *seq; // Dynamic allocation for the DNA sequence
    FILE *fp_in, *fp_out; // File pointers for input and output files

    // Opens the input file in read mode
    fp_in = fopen("Borrelia_burgdorferi_B31_complete_genome.txt", "r");
    // Opens the output file in write mode
    fp_out = fopen("Z_curve_output.txt", "w");
    // Checks if opening files failed
    if (fp_in == NULL || fp_out == NULL) {
        printf("Error: Unable to open file(s).\n");
```

```
        return 1;
    }

    // Determines the size of the sequence file
    fseek(fp_in, 0, SEEK_END);
    long file_size = ftell(fp_in);
    fseek(fp_in, 0, SEEK_SET);

    // Allocates memory for the DNA sequence
    seq = (char *)malloc((file_size + 1) * sizeof(char));
    // Checks if memory allocation failed
    if (seq == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Reads the DNA sequence from the file
    fread(seq, sizeof(char), file_size, fp_in);
    seq[file_size] = '\0'; // Null-terminate the sequence

    // Calculates and outputs the Z curve data
    calculate_z_curve(seq, file_size, fp_out);

    // Frees dynamically allocated memory for the DNA sequence
    free(seq);
    // Closes the input and output files
    fclose(fp_in);
    fclose(fp_out);
    // Returns 0 to indicate successful execution of the program
    return 0;
}
```

5.4 Marche sur l'ADN programme

```
    #include <stdio.h>    // Includes the necessary libraries for input/output
#include <stdlib.h>   // Includes the standard library for memory allocation functions
#include <string.h>   // Includes the library for manipulating strings
#include <ctype.h>    // Includes the library for character manipulation functions

// Function to calculate the DNA walk and produce the Z curve data
void calculate_dna_walk(char *seq, int len, FILE *fp_out) {
    // Initialize counters for each nucleotide
    int a_count = 0, t_count = 0, g_count = 0, c_count = 0;

    // Traverse the sequence to count nucleotides
    for (int i = 0; i < len; i++) {
        char nucleotide = toupper(seq[i]); // Convert to uppercase for consistency
```

```
        // Increment the respective counter based on the type of nucleotide
        switch (nucleotide) {
            case 'A':
                a_count++;
                break;
            case 'T':
                t_count++;
                break;
            case 'G':
                g_count++;
                break;
            case 'C':
                c_count++;
                break;
            default:
                continue; // Ignore unknown characters
        }
    }

    // Traverse the sequence again to calculate and produce the Z curve data
    for (int i = 0; i < len; i++) {
        // Calculate the x and y values for the Z curve
        double x = (double)(a_count - t_count) / len;
        double y = (double)(g_count - c_count) / len;
        // Produce the x and y values in the specified file
        fprintf(fp_out, "%.4f\t%.4f\n", x, y);

        // Update counters based on the current nucleotide for the next iteration
        char nucleotide = toupper(seq[i]);
        switch (nucleotide) {
            case 'A':
                a_count--;
                break;
            case 'T':
                t_count--;
                break;
            case 'G':
                g_count--;
                break;
            case 'C':
                c_count--;
                break;
            default:
                continue; // Ignore unknown characters
        }
    }
}
```

```c
// Main function
int main(void) {
    char *seq; // Pointer to dynamically allocated sequence
    FILE *fp_in, *fp_out; // File pointers for input and output files

    // Open the input and output files
    fp_in = fopen("Borrelia_burgdorferi_B31_complete_genome.txt", "r");
    fp_out = fopen("dna_walk_output.txt", "w");
    // Check if opening files failed
    if (fp_in == NULL || fp_out == NULL) {
        printf("Error: Unable to open file(s).\n");
        return 1;
    }

    // Determine the size of the sequence file
    fseek(fp_in, 0, SEEK_END);
    long file_size = ftell(fp_in);
    fseek(fp_in, 0, SEEK_SET);

    // Allocate memory for the sequence
    seq = (char *)malloc((file_size + 1) * sizeof(char));
    // Check if memory allocation failed
    if (seq == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Read the DNA sequence from the file
    fread(seq, sizeof(char), file_size, fp_in);
    seq[file_size] = '\0'; // Terminate the sequence with a null character

    // Call the function to calculate the DNA walk and produce the Z curve data
    calculate_dna_walk(seq, file_size, fp_out);

    // Free dynamically allocated memory and close file pointers
    free(seq);
    fclose(fp_in);
    fclose(fp_out);
    return 0; // Return 0 to indicate successful execution
}
```