Compte Rendu du Projet Langage C

Lienard Simon, **Pecoraro** Gabriel
Professeur Encadrant : **Bourmaud** Guillaume
Groupe F

Mai 2023

Table Des Matières

1 Introduction

Ce projet est la finalité de tout ce que nous avons appris dans le module IF112 en ce qui concerne le traitement d'image ce semestre. Pour cet exercice final, nous allons réaliser une fractale qui s'appelle "La Fougère de Barnsley". Une image de la fougère s'affiche ci-contre:

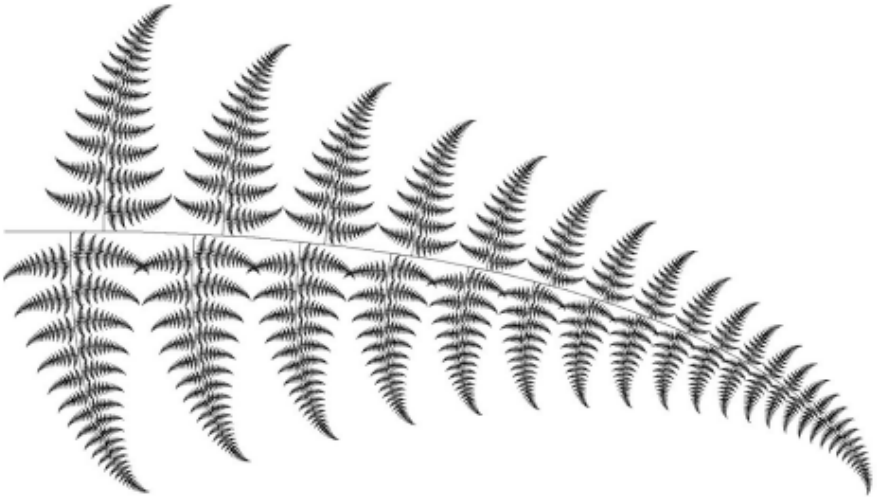

Figure 1: Fougère de Barnsley

Nous allons donc concevoir un algorithme pour tracer cette fougère et nous allons expliciter comment nous avons étayé notre pensée et les chemins qui nous ont mené à l'algorithme final. Dans la suite, pour chaque code on utilisera les modules suivants que nous importons ci-contre:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#define taillex  1600
#define tailley  1000
#define scale 1500 //echelle de l'image
#define pi M_PI
```

On cherche à afficher une fougère de taille 1600×900. Pour ce faire, les modules usuels sont importés mais on définit aussi des variables globales qui correspondent à la taille de l'image totale qui sera ici $taillex \times tailley = 1600 \times 1000$. De plus, on définit un facteur $scale = 1500$ qui est ici l'échelle de l'image. Enfin, pour nous faciliter la programmation, on redéfinit en variable globale π avec une syntaxe plus simple et plus intuitive à utiliser. Pour travailler, on se place dans un repère $(x, -y)$ donc si on monte sur l'image, la coordonnées en y décroît.

2 Premiers Pas Dans Le Codage

Nous avons passer la première séance à peaufiner une stratégie qui nous permettrait de déterminer un algorithme viable et qui marche par la même occasion mais aussi qui utilise uniquement les outils vus pendant les Travaux Pratiques. Nos pensées ont divergé sur un point, non-négligeable, itératif ou récursif, tel était la question. L'itératif semblait contenir des programmes plus simples mais fastidieux à assembler tandis que le récursif lui semblait plus rapide car il pouvait s'écrire en une fonction mais difficile à mettre en place. Nous avons opter d'abord sur le récursif.

Avant l'implanter un algorithme, nous avons réutiliser les fonctions et structures des travaux précédents.

```
struct color{//importation structure couleur
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};
```

Cette première structure permet de définir la couleur d'un pixel en utilisant les composantes Red,Green,Blue (RGB) en les définissant avec des `unsigned char`. Ce type est défini sur $[0, 255]$ ce qui nous offre une large palette de couleur.

```
    struct picture{
    int width; //largeur
```

```
    int height; //hauteur
    struct color *pixel;
};
```

Cette structure défini une image en tant que telle avec sa largeur qui est ici `width` de type `int` et de même sa hauteur. De plus, pour l'image on définit un tableau de pointeurs de pixel qui permet de caractériser chaque pixel de l'image en utilisant les structures précédentes.

```
    struct point{
    double x;
    double y;
};
```

Une dernière structure qui définit dans un plan les coordonnées d'un point avec abscisse et ordonnées qui sont des doubles car avec des calculs de normes par exemple, on peut avoir des nombres non-entiers donc il faut anticiper.

Nous allons désormais parler des fonctions

```
    struct picture new_pic(double largeur, double hauteur){//creation d'une
    image vierge
    struct picture image;
    image.width = largeur;
    image.height = hauteur;
    double taille = largeur*hauteur;
    image.pixel = malloc(taille*sizeof(struct color));//on alloue la memoire
    for(int i = 0; i< taille; i++){//boucle qui initialise chaque pixel
        image.pixel[i].blue = 0;
        image.pixel[i].green = 0;
        image.pixel[i].red = 0;
    }
    return image;
}

void save_pic(struct picture image, char * fichier){//sauvegarde

    FILE * im = fopen(fichier, "w");
    if(im == NULL){
        printf("erreur de creation fichier\n");
    }
    else{
        fprintf(im, "P6");
        fprintf(im, " ");
        fprintf(im, "%d", image.width);//on converti en const char*
        fprintf(im, " ");
        fprintf(im, "%d", image.height);//on converti en const char*
        fprintf(im, " ");
        fprintf(im, "255");
        fprintf(im, " ");
        printf("%s en fichier cree\n",fichier);
    }
    for(double i = 0; i<image.height; i++){
        for(double j = 0; j<image.width; j++){
            int position = i*image.width + j;
            fputc(image.pixel[position].red, im);
            fputc(image.pixel[position].green, im);
            fputc(image.pixel[position].blue, im);
        }
    }

}
```

Nous avons directement ici introduit deux fonctions. Lorsque on les fusionne elle crée une image. Chaque fonction est une étape dans cette création. La première intitulée `new_pic` prend en argument la largeur et la hauteur d'une image et nous renvoie une image de type `struct picture`. Le but de cette fonction est de fixer la largeur et la hauteur dans les champs de structures de `image` qui a été initialisée au préalable. Par la suite, en calculant la taille de l'image (`width` × `height`). On initialise le tableau de pixels avec un `malloc(taille*sizeof(struct color))`. On utilise par la suite la syntaxe des tableaux unidimensionnelles avec une boucle qui remplit un à un les pixels de notre tableau en leurs assignant les composantes RGB. En ce qui concerne sa complexité, elle est dominée par la boucle `for` et l'allocation de mémoire, qui ont tous les deux une complexité linéaire. Par conséquent, la complexité globale du code est $O(n)$, où n est le nombre total de pixels dans l'image.

La fonction `save_pic` enregistre l'image dans nos fichiers et nous permet d'aller la regarder. Elle ne renvoie rien car elle est de type `void` et elle prend en argument une image de type `struct picture` qui est l'image à sauvegarder et une chaîne de caractère qui est le nom du fichier où sera stocker l'image. Au début de la fonction, on créer un pointeur de type `FILE` appelé `im` qui, en utilisant la fonction `fopen`, ouvre un nouveau document appelé "fichier" et qui avec `"w"` nous donne les droits d'écriture dans le fichier. Si le fichier créé pointe sur rien, on renvoie un texte qui indique que le fichier n'a pas été créé. Sinon, on utilise `fprintf` qui est utilisée pour écrire des données formatées dans un flux de sortie spécifié. Cela nous permet de paramétrer la taille et la largeur de l'image mais aussi de fixer les couleurs comme le fait la double boucle dans le programme qui utilise les tableaux unidimensionnels. La complexité de ce code est linéaire, c'est-à-dire $O(n)$, où n est le nombre total de pixels dans l'image (largeur * hauteur). À l'intérieur des deux boucles imbriquées, les valeurs de chaque pixel sont écrites dans le fichier à l'aide de la fonction `fputc`. Cette opération est de temps constant. La complexité de la boucle externe dépend de image.height, et la complexité de la boucle interne dépend de image.width. Ainsi, la complexité totale du code est proportionnelle à la taille de l'image, soit $O(n)$, où n est le nombre total de pixels.

Pour finir sur ces deux fonctions, il faut préciser que ces deux fonctions encadrent la création de l'image. De fait, `new_pic` crée l'image et `save_pic` l'enregistre, ce qui indique que toutes modifications apportées à l'image via des fonctions ou dans le `main` devront être faite entre ces deux fonctions. Ci-contre la structure du `main` type:

```
int main(){
    image = new_pic(longueur,largeur);
    //
    //
    //modifications sur l'image
    //
    //
    save_pic(image,"nom_image.ppm");
    //le ppm est essentiel pour preciser que c'est une image
    free(image.pixel);//liberation de l'espace memoire occupe par le
    tableau de pixel
    return 0;
}
```

Et enfin, pour conclure sur les fonctions élémentaires de notre futur algorithme, nous détaillons ici les fonctions pour tracer des traits, dessiner des points et d'aide au calcul. La première permet de fixer la couleur d'un pixel:

```
void set_pixel(struct picture image, double x, double y, struct color
color_c){//fonction qui modifie pixel
    int position = y*(image.width) + x;
    image.pixel[position].red = color_c.red;
    image.pixel[position].green = color_c.green;
    image.pixel[position].blue = color_c.blue;
}
```

Cette fonction est de type `void`. Elle prend en argument une `struct picture` qui est une image, l'abscisse et ordonnées de la case à modifier définit par `(x,-y)` et la couleur qu'on veut donner à notre pixel de type `struct color`. On calcul au préalable la position du pixel à modifier avec une fonction extraite qui localise dans le tableau en fonction de x et y, on stocke sa valeur dans un entier `position`. Enfin, on localise dans le tableau de pixel la valeur de `position` concernée et on modifie ses couleurs. Ici, il y a aucun processus qui amènerait une complexité linéaire ou autre. On en conclut que cette fonction est de complexité constante $O(1)$.

La seconde permet de tracer une ligne entre deux points:

```
void draw_line(struct picture image, double x1, double y1, double  x2,
```

```
    double y2, struct color color_c){/*il faut
    une nouvelle fonction qui trace toutes les droites peu importe le
    coefficients directeur*/
    double nb_pixels = fmax(fabs(x1-x2),fabs(y1-y2)) + 1;
    if((x1 == x2) && (y1 < y2)){
        for(int i = 0; i<nb_pixels; i++){
            set_pixel(image, x2, y2-i,color_c);
        }
        return;
    }
    else if((x1 == x2) && (y1 > y2)){
        for(int i = 0; i<nb_pixels; i++){
            set_pixel(image, x1, y1 - i,color_c);
        }
        return;
    }
    else if((y1 == y2) && (x1<x2)){
        for(int i = 0; i<nb_pixels; i++){
            set_pixel(image, x1 + i, y1, color_c);
        }
        return;
    }
    else if((x1 > x2) && (y1 == y2)){
        for(int i = 0; i<nb_pixels; i++){
            set_pixel(image, x2 + i, y1,color_c);
        }
        return;
    }
    else{
        for(int i = 0; i<nb_pixels; i++){
            float t = (float) i/ (float) nb_pixels;
            double x = round(x1+ t*(x2 - x1));
            double y = round(y1 + t*(y2 - y1));
            set_pixel(image, x, y, color_c);
        }
        return;
    }
}
```

Cette fonction en argument une image de type `struct picture`, deux points avec respectivement leurs abscisses et coordonnées et une couleur de type `struct color`. L'objectif est de tracer un trait entre les deux points sur la couleur. Tout d'abord, on calcule le nombre de pixels à tracer en utilisant une norme vectorielle. Par la suite, on fait une disjonction de cas pour les configurations faciles et une régression linéaire pour des traits en diagonale. Cette algorithme se base sur la méthode de tracé de ligne de Bresenham mais a été largement modifié. Elle ne renvoie rien car elle est de type `void` mais elle dessine le trait sur notre image. La complexité de ce code est $O(n)$ car elle dépend de la taille des fonctions `for` qui sont linéaire, d'où une complexité en $O(n)$

On introduit enfin une dernière fonction qui permet de calculer les coordonnées d'un point d'arriver si on connaît les coordonnées de départ et la distance entre les deux points. Ci-contre le code:

```
struct point coordonnes(struct point p1, double angle/*radian*/, double
echelle, double distance){
//calcul des coordonees d'un point pour une distance d'un point connu a ce
point
    while (angle < 0.0) {
            angle += 2 * M_PI;
            }
    while (angle >= (2 * M_PI)) {
            angle -= 2 * M_PI;
```

```
        }
    struct point p2 ={0,0};
    p2.x = abs(distance*echelle*cos(angle) + p1.x);
    p2.y = abs(-distance*echelle*sin(angle) + p1.y);
    return p2;
}
```

On doit renvoyer une structure de type point `struct point`. On prend en argument, le point de départ `p1`, l'angle par rapport à l'origine en radians de type `double`, l'échelle de la figure de type `double` et la distance de type `double`. Tout d'abord sachant qu'on travaille dans le cercle trigonométrique $[0; 2\pi[$, on doit normaliser l'angle. Cela nous permet de réduire considérablement le nombre de cas. Par la suite, on utilise les projections usuels entre les angles. La valeur absolue ici, nous permet de ne pas avoir de coordonnées négatives qui pourrait mener une erreur de segmentation ou a des résultats inattendus. La complexité de ce code est constante $O(1)$ car il n'y a aucune fonction linéaire qui dépendrait d'un nombre de terme.

3 Recherche d'algorithme

3.1 Récursion ratée

Au tout début, nous n'avions pas la fonction `coordonnes`, on a néanmoins tenté une récursion pour essayer de tout afficher du premier coup. Le code suivant s'est soldé avec une erreur de segmentation:

```
void tige(struct picture image,struct color color_c,int n){
int x1;
int y1;
if(n == 10){
    return;
}
x1=image.xstart+image.scale*cos(image.angle);
y1=image.ystart-image.scale*sin(image.angle);
image.angle=-0.05+image.angle;
image.scale=image.scale*0.86;
draw_line(image,x1,y1,image.xstart,image.ystart,color_c);
image.xstart=x1;
image.ystart=y1;
n++;
tige(image, color_c,n);

}
```

Ce qui faisait défaut sur cette fonction était sûrement le fait que l'on calculait de manière erronées les coordonnées et on sortait de l'image. Bien que la complexité de code était optimale, on aurait eu des longs temps de compilation à cause des calculs fastidieux. Cela nous donnait une complexité en $O(n)$. Après cela, on pensait que le récursif était très difficile à mettre en place, on a donc essayé une approche itérative en se penchant plus sur la théorie.

3.2 Code Itératifs Raté

L'idée de cette approche itérative était de faire un code pour chaque composant de la fougère soit la tige principale et les deux feuilles. On a donc commencé par le plus simple, la tige. Grâce à la fonction `coordonees`, on a calculé à l'aide de l'énoncé (cf. tableau suivant) la coordonnées de convergence de la tige et on a encadré notre fougère grâce à une approximation grossière avec aussi ce que stipulait l'énoncé:

Longueurs	OH =	1 (taille de référence)
	OA / OH =	0.335
	OB / OH =	0.877
	OC / OH =	1.086
	OD / OH =	0.765
	OE / OH =	0.339
Angles (radians)	HOA =	0.9421
	HOB =	-0.0647
	HOC =	-0.3
	HOD =	-0.263
	HOE =	-1.175

Figure 2: Le tableau des distances

```
void iteratif_devant(struct picture image,
struct color color_c,int n){
//initialisation
struct point Og = {0,0};
//point O de départ
struct point O = {10,424};
//initialisation point O_G
struct point O_ginit = {240,424};
//première droite
draw_line(image,O.x,O.y,O_ginit.x,O_ginit.y,color_c);
int a = 1; //compteur itérations
while((a != n)){
    Og.x = 0.16*pow(0.86,a)
    *scale*cos(-0.05*a) + O_ginit.x;
    Og.y = -0.16*pow(0.86,a)
    *scale*sin(-0.05*a) + O_ginit.y;
    //Og = coordonnes(O_ginit,-0.05*a,scale,
    pow(0.86,a)); mène à segmentation fault
    draw_line(image,Og.x,Og.y,O_ginit.x,
    O_ginit.y,color_c);
    //on copie les coordonées
    O_ginit.x = Og.x;
    O_ginit.y = Og.y;
    a++;
}
}
```

Positions	O Og / OH	0,16
	O Or / OH	0,12
	O Ob / OH	0,08
Angles (radians)	H Og Hg	-0,05
	H Or Hr	π/2
	H Ob Hb	-π/2
Tailles	OgHg / OH	0,86
	OrHr / OH	0,27
	ObHb / OH	0,30

Figure 3: Tableau les points des feuilles et tailles tiges

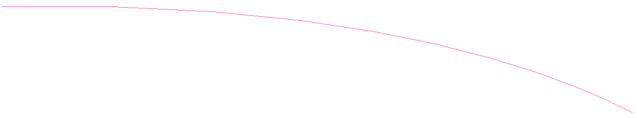

Figure 4: tige pour 10 itérations

Dans ce programme, nous avons exhiber une suite qui converge vers le point C en utilisant n itérations. À chaque tour de boucle, on calcul les nouvelles coordonnées du prochain point et on trace depuis le point présent jusqu'au prochain. Après le tracé, le point futur devient le point présent. La boucle `while` apporte une complexité linéaire en $O(n)$ sur le programme. Ainsi, ce dernier est-il viable et à des temps d'exécutions corrects. Néanmoins, tracer une tige était facile car on devait prendre en paramètre que un seul point. Pour les feuilles, ça commençait à se corser car il fallait prendre en compte trois paramètres respectivement le prochain point de la tige 0_G, le prochain point de la feuille du haut 0_R et le prochain point de la feuille du bas O_B. La récursivité était donc la solution au problème.

4 Programme Final avec la Récursivité

4.1 Première Solution

On a donc transposé le programme itératif précèdent en récursif et cela nous a permis de généraliser la fonction pour pouvoir tracer les feuilles de la fougère. Le raisonnement est le suivant.

Comme énoncé dans le sujet, chaque élément de la figure se trace à partir de 3 données. Ces données sont le point d'origine, l'angle et la taille. Nous allons donc utiliser des coordonnées polaires pour tracer notre figure. Avant de tracer, il est nécessaire d'utiliser la fonction `coordonnees` pour convertir les coordonnées polaires en coordonnées cartésiennes, afin de pouvoir tracer les droites à l'aide des fonctions `drawline` et `set_pixel`.

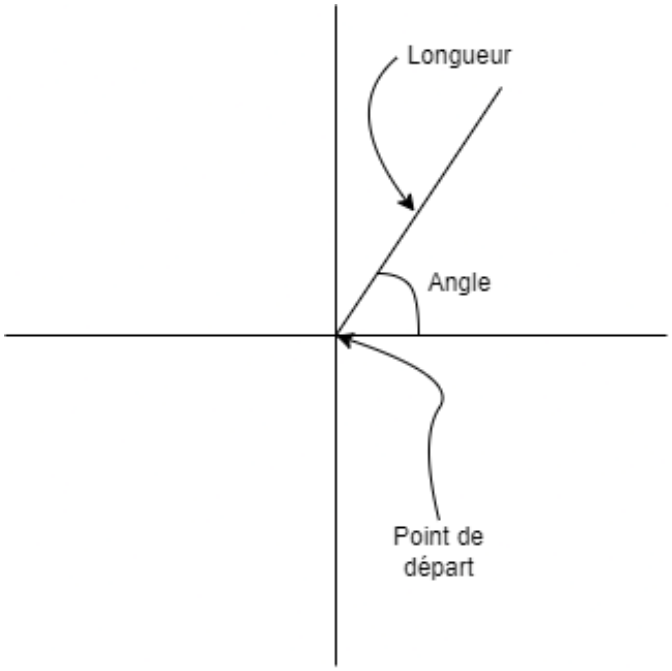

Figure 5: Diagramme représentant les 3 données de base

Ensuite, notre code récursif pourrait être formulé ainsi :

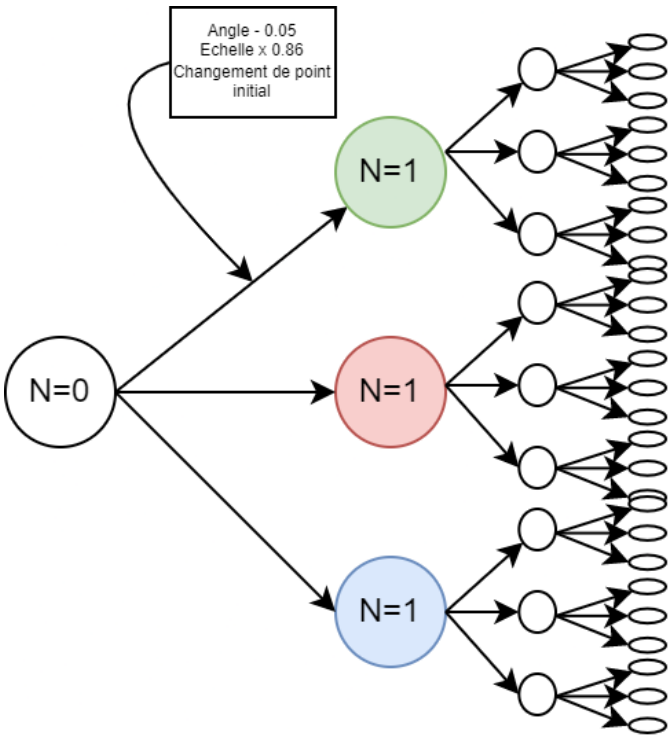

Figure 6: Diagramme représentant notre code récursif

Dans ce diagramme, les bulles représentent la fonction de traçage d'une ligne. Cependant, comme on peut le voir, cette fonction est appelée plusieurs fois. Ainsi, à l'étape 0, lorsque $N = 0$, le programme trace OO_G (conformément aux notations de l'énoncé). Ensuite, après avoir tracé cette ligne, le programme rappelle la même fonction avec un point de départ, un angle et une longueur différents. Ici, dans le cercle vert, nous traçons la ligne qui prolonge la tige centrale. On va donc avoir :

$$Angle_{n+1} = Angle_n - 0.05$$
$$Echelle_{n+1} = Echelle_n * 0.86$$

Et le point O va devenir le point O_G.
Pour la bulle en rouge, on va pouvoir tracer la première tige de la première feuille qui monte. Pour cela, on réalise les mêmes opérations que pour la bulle verte, mais en adaptant les valeurs. Par exemple, on a :

$$Angle_{n+1} = Angle_n + \frac{\pi}{2}$$

ou pour l'échelle :

$$Echelle_{n+1} = Echelle_n * 0.27$$

Et de la même manière, on remplace O parO_R.

Et on répète l'opération pour la bulle bleue.

Mais avant d'avoir réalisé la bulle rouge, la bulle verte va elle-même s'appeler 3 fois avec à chaque fois de nouveaux arguments.

Cependant, lorsque nous avons lancé ce programme avec seulement ces quelques considérations, nous avons remarqué que les tiges du bas ne se refermaient pas vers la droite mais vers la gauche. Cela s'explique par le fait que lorsqu'on retire $0.05, \text{rad}$ à un angle, ce dernier tourne toujours dans le sens horaire. Afin de résoudre ce problème, nous avons introduit la variable `power` de type `int`, qui nous permet de contrôler le signe de la valeur $-0.05, \text{rad}$. Ainsi, lorsque la tige continue dans sa lancée et reste droite, afin d'éviter toute aberration, le paramètre `power` reste le même. Lorsque nous traçons une tige qui remonte, nous conservons le signe "-" en attribuant `power=1`. Finalement, lorsque nous souhaitons tracer une tige partant vers le bas, nous devons mettre `power=0` afin d'obtenir la branche qui se replie sur elle-même dans le sens trigonométrique. Ci-contre le code de la première solution :

```
    void recursivite_devant(struct picture fougere, struct point depart,
    double angle, double echelle,struct color color_c, int n,int power){
    n++;
    if (n>15){
        return;
    }
    struct point next=coordonnes(depart, angle, echelle, 0.16);
    double angle1=angle+0.05*pow(-1,power);
    double echelle1=echelle*0.86;
    draw_line(fougere,depart.x,depart.y,next.x,next.y,color_c);
    recursivite_devant(fougere, next, angle1, echelle1,color_c,n,power);
    struct point next1=coordonnes(depart, angle, echelle, 0.14);
    double echelle2=echelle*0.27;
    double angle2=angle + pi/2;
    recursivite_devant(fougere, next1, angle2, echelle2,color_c, n,1);
    struct point next2=coordonnes(depart, angle, echelle, 0.1);
    double echelle3=echelle*0.30;
    double angle3=angle - pi/2;
    recursivite_devant(fougere, next2, angle3, echelle3,color_c, n,0);
}
```

Cette fonction nous donne l'image suivant:

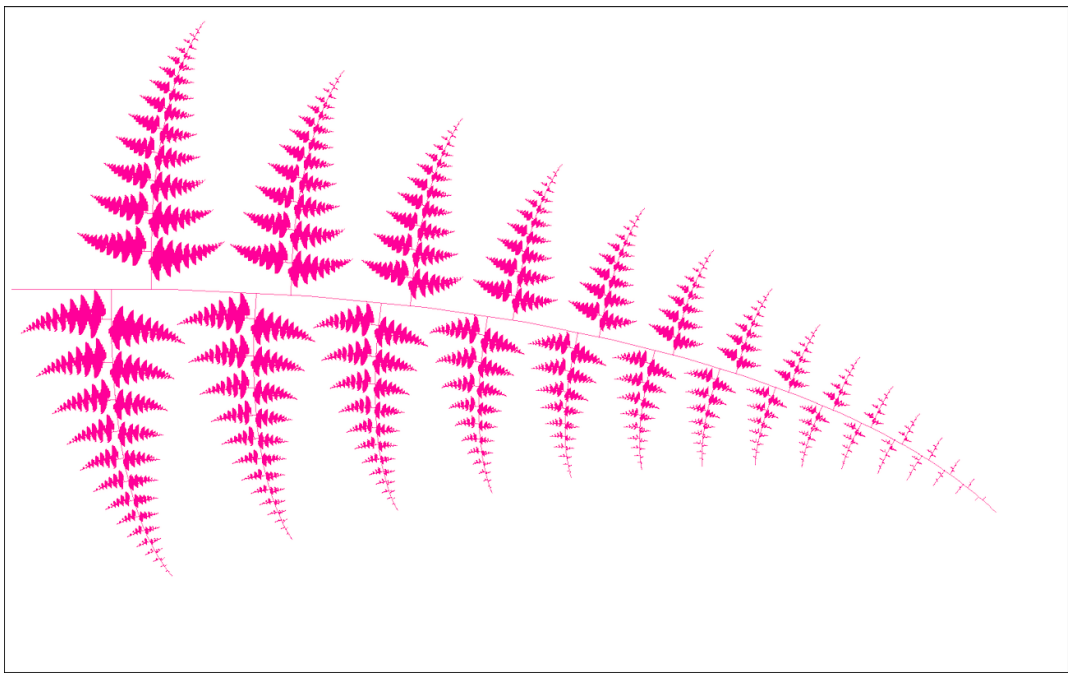

Figure 7: Première Fougère de Barnsley

On remarque que c'est presque ce qu'on vise mais le résultat n'est pas celui attendu. En effet, les feuilles sont trop grandes la fin de la tige n'est pas très feuillus. De plus, ce programme à une complexité de $O(2^n)$, cela vient du fait que a fonction `recursivite_devant` est récursive et s'appelle elle-même plusieurs fois avec des valeurs différentes pour les arguments. Dans chaque appel récursif, le nombre total d'appels double car il y a trois appels récursifs (pour `next`, `next1` et textnext2). Par conséquent, le nombre total d'appels récursifs est de l'ordre de 2^n d'où la complexité. Cela veut dire que plus la condition d'arrêt dans le `if` est grande plus le temps de compilation est long. Ci-contre, on a mesuré le temps de compilation pour différentes valeurs de n et nous avons exhiber un graphe pour chacun :

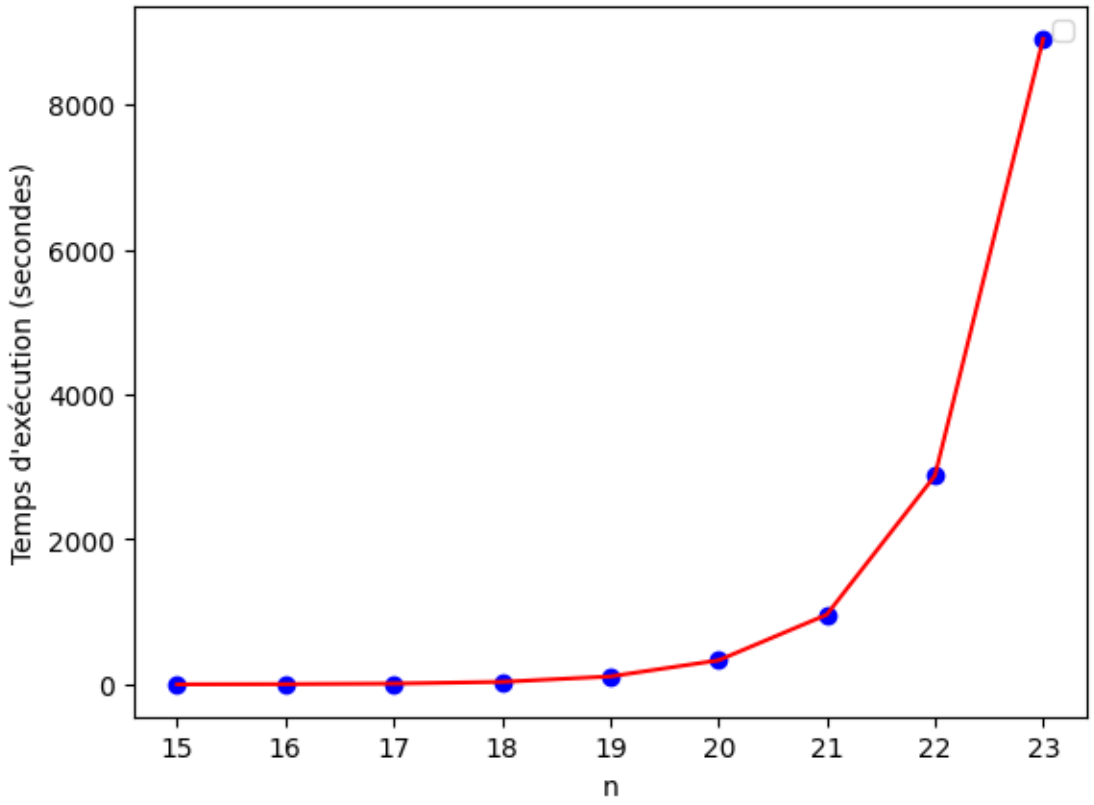

Figure 8: Temps d'éxécution en fonction de la condition d'arrêt

On montre le `main` qui permet de mesurer les temps d'éxécution

```
int main(){
clock_t start, end;
double cpu_time_used;
// Obtenir le temps de depart
start = clock();
int a = rand()%255;
int b = rand()%255;
int c = rand()%255;
struct color couleur = {a,b,c};
struct color white = {255,255,255};
int n = 0;
struct point O = {10, 424};//point depart
struct picture fougere = new_pic(taillex,tailley);
for(int i = 0; i<fougere.height; i++){
    for(int j = 0; j<fougere.width; j++){
        set_pixel(fougere,j,i,white);
    }
}rendu
recursivite_devant(fougere,O,0,scale,couleur,n,1);
save_pic(fougere,"fougere.ppm");
free(fougere.pixel);
// Obtenir le temps d arret
end = clock();

// Calculer le temps ecoule en secondes
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

```
    // Afficher le temps d execution
    printf("Temps d execution : %.6f secondes\n", cpu_time_used);
    return 0;
```

On remarque que les temps d'éxécution deviennent très long à partir d'un certain n ce qui montre qu'il faudrait modifier quelque chose dans notre code pour le rendre plus rapide. À noter que ces valeurs peuvent fluctuer car certaines mesures temporelles ont était réalisé avec l'ordinateur portable (MAC OS) branché à l'alimentation et d'autres non. Nous avons donc essayé d'optimiser notre algorithme récursif.

4.2 Solution finale

Dans cette partie nous nous proposons d'améliorer notre code pour pouvoir obtenir une image propre de la fougère de Barnsley et aussi de réduire le temps d'exécution du programme. L'idée est la suivante :

On remarque dans notre programme, que l'on réalise Voici le programme ci-contre résultant de cette idée qui est sensiblement le même que celui de la partie précèdente:

```
    void recursivite_devantopti(struct picture fougere, struct point
    depart, double angle, double echelle,struct color color_c, int n,int power){
    int a = rand()%30;
    color_c.green = 255-a;
    color_c.red = 255-a*3+20;
    n++;
    if (n>40){
        return;
    }
    struct point next=coordonnes(depart, angle, echelle, 0.16);
    double angle1=angle+0.05*pow(-1,power);
    double echelle1=echelle*0.86;
    draw_line(fougere,depart.x,depart.y,next.x,next.y,color_c);
    recursivite_devantopti(fougere, next, angle1, echelle1,color_c,n,power);
    struct point next1=coordonnes(depart, angle, echelle, 0.14);
    double echelle2=echelle*0.27;
    double angle2=angle + pi/2;
    recursivite_devantopti(fougere, next1, angle2, echelle2,color_c, n+pas,1);
    struct point next2=coordonnes(depart, angle, echelle, 0.1);
    double echelle3=echelle*0.30;
    double angle3=angle - pi/2;
    recursivite_devantopti(fougere, next2, angle3, echelle3,color_c, n+pas,0);
}
```

Ce programme à une complexité qui fluctue suivant le n et le pas qu'on donne pour avancer plus vite dans les itérations des feuilles. La partie récursive de la fonction `recursivite_devantopti` est appelée deux fois à chaque niveau de récursion, et la valeur de n est incrémentée de `pas` à chaque appel récursif. Ainsi, le nombre total d'appels récursifs est de l'ordre de $2^{(\frac{n}{pas})}$, et chaque appel récursif effectue des opérations de complexité $O(1)$ et $O(n)$ qui sont celles de `coordonnees` et `draw_line`. On peut en conclure que la complexité totale du programme `recursivute_devantopti` est de $O(n2^{\frac{n}{pas}})$. On remarque donc que la complexité totale du programme à été amélioré ce qui nous permet de tracer une image nettement plus belle comme celle ci-contre:

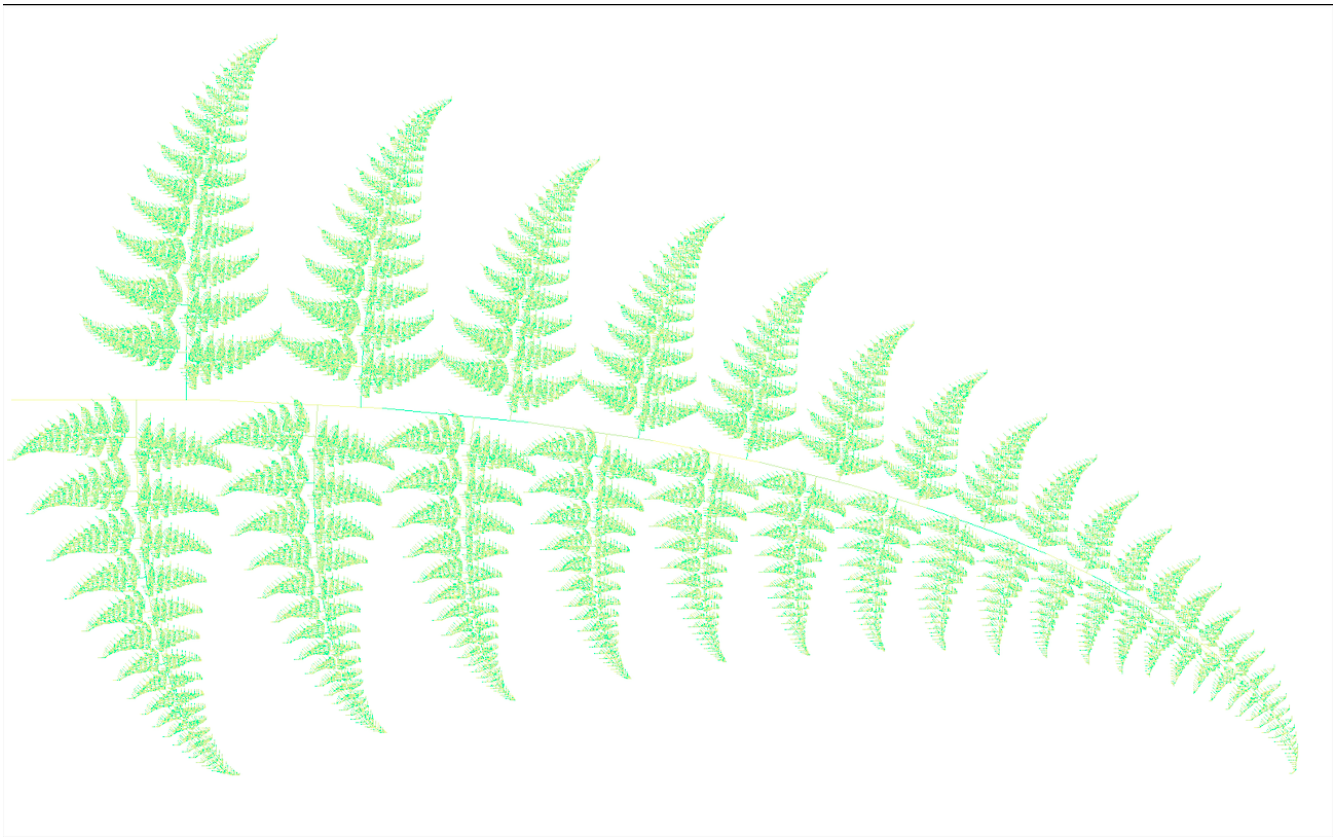

Figure 9: Fougère de Barnsley

On a donc une fougère beaucoup plus belle et réaliste par rapport à celle qu'on voulait. On peut donc conclure que ce programme est viable. On propose néanmoins de faire une analyse temporelle du code. On prend en compte le `pas` et `n` la condition d'arrêt. On a le graphe suivant pour les résultats calculé avec le `main` situé en annexe:

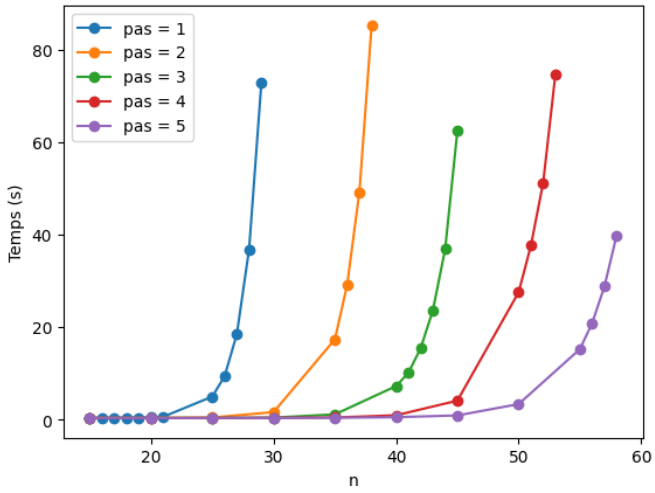

Figure 10: Graphe temporel pour les différents pas

Tout d'abord, on remarque par rapport au graphe du programme précèdent que l'ajout de ce pas a permis d'avoir un gain temporelle lors de l'exécution plutôt concluant. En effet, alors que pour $n = 20$, on avait un temps de compilation de l'ordre de $332s$, ici même avec un `pas` $= 1$, on est aux alentours de $0.4s$. De plus, cette analyse temporelle permet de mettre en relief le fait que avec un pas plus grand, on peut effectuer de plus en plus d'itérations dans la fonction récursives. Ceci permet

d'offrir plus de détails dans les feuilles de la fougères. Néanmoins, cela amène aussi à une discussion. En effet, si il y a trop d'itérations, la fougère risque d'être trop feuillu et donc on risque de diverger du résultat final voulu. Ainsi, voit-on avec le graphe qu'il faut choisir un nombre d'itérations et un pas suffisant pour que la fougère soit concluante mais qui non plus ne prends pas trop de temps de d'exécution. On lit sur le graphe que pour un pas de 4, on a $n = 40$, un nombre suffisant d'itérations et un temps d'exécution plutôt raisonnable. Ce résultat peut-être parmi d'autres, une configuration optimale pour un bon affichage de la fougère, c'est d'ailleurs celle-là qu'on a choisi pour la fougère affiché précédemment(cf figure 7).

5 Conclusion

5.1 Possibilité d'Amélioration de l'Algorithme

Notre programme est viable et il fonctionne très bien. De plus, il est rapide bien qu'il est une complexité en $O(n2^n)$. Néanmoins, si on souhaite agrandir cette image par exemple en la mettant au format 10000×10000 avec une échelle à 9000, on a un temps de compilation autour de $20s$ mais un résultat beaucoup plus net (cf Annexe). On remarque que l'on perd en vitesse d'exécution et que notre complexité nous met en défaut. En somme, le programme doit être optimisé pour répondre au besoin de l'utilisateur qui souhaiterait en changer la taille sans trop attendre non plus que la compilation s'effectue. Ainsi, pour implémenter une triple récursivité on pourrait avoir recours à une structure contenant une liste chaînée qui permettrait d'accélérer la récursivité.

```
struct pointchaine{
    double x;
    double y;
    strcut pointchaine* next;
};
```

Cela permettrait de réduire de manière drastique la complexité car tout les éléments pointeraient vers le prochain point en utilisant les calculs de coordonnées introduit précédemment. Néanmoins, nous avons pas fais cela ici, car cette configuration reviendrait à utiliser une triple liste chaînées et à créer ce qu'on appelle, un arbre binaire dont la racine serait le point O de l'énoncé. Or, cela dépasse largement nos compétences pour l'instant et de long calculs de dénombrement serait nécessaire pour en créer un viable qui répondrait à nos besoins.

5.2 Remerciements

Ce projet était très formateur et reprenait de manière habile toutes les compétences acquises au cours de ce semestre et toutes les notions vu en traitement d'image. Bien qu'il soit clair et limpide, le problème laisse de la place pour mûrir sa réflexion informatique et octroie plusieurs méthodes pour être résolu. Il faisait, de plus, appelle à certains instruments de calcul vectoriel et de dénombrement ce qui permet un mélange très intéressant de différentes matières. Merci à Mr. Bourmaud de nous avoir surveillé et conseillé tout au long du projet.

6 Annexe

```
int main(){
clock_t start, end;
double cpu_time_used;
// Obtenir le temps de départ
start = clock();
int a = rand()%255;
int b = rand()%255;
int c = rand()%255;
struct color couleur = {a,b,c};
struct color white = {255,255,255};
int n = 0;
struct point O = {10, 474};
//struct point
struct picture fougere = new_pic(taillex,tailley);
for(int i = 0; i<fougere.height; i++){
```

```
    for(int j = 0; j<fougere.width; j++){
        set_pixel(fougere,j,i,white);
    }
}
recursivite_devantopti(fougere,0,0,scale,couleur,n,1);
save_pic(fougere,"fougere.ppm");
free(fougere.pixel);
// Obtenir le temps d'arrêt
end = clock();

// Calculer le temps écoulé en secondes
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

// Afficher le temps d'exécution
printf("Temps d'execution : %.6f secondes\n", cpu_time_used);
//Version initiale
//Pour n = 15, on a  1.616672 secondes, 1.687434
//Pour n = 16, on a  4.489257 secondes, 4.439246
//Pour n = 17, on a 12.201812 secondes
//Pour n = 18, on a 35.730829 secondes
//Pour n = 19, on a 110.629473 secondes
//Pour n = 20, on a 332.843410 secondes
//Pour n = 21, on a 962.208853 secondes
//Pour n = 22, on a 2880 seconds
//Pour n = 23, 8903 seconds
//Version optimisée
////////////////////////////////////////////////////////////
//pas = 1
//n=15,  0.331147 s
//n=16, 0.344836 s
//n=17,0.349569 s
//n = 18, 0.404237 s
//n=19, 0.401690 s
//n= 20,0.457686 s
//n=21, 0.597474 s
//n=25,4.962793 s
//n=26,9.364234 s
//n=27, 18.483976 s
//n=28, 36.836083 s
//n=29, 72.791340 s
////////////////////////////////////////////////////////////
//pas = 2
//n=15,0.319623 s
//n=20, 0.328752 s
//n=25,0.415717 s
//n=30,1.603619 s
//n=35,17.341674 s
//n=36,29.096037 s
//n=37,49.130844 s
//n=38, 85.315097 s
//////////////////////////////
//pas = 3
//n=15,  0.325388 s
//n=20,  0.309948 s
//n=25,  0.325650 s
//n=30,  0.405966 s
```

```
//n=35,  1.102068 s
//n=40,  7.230840 s
//n=41,  10.098332 s
//n=42,  15.405791 s
//n=43,  23.635064 s
//n=44,  36.849929 s
//n=45,  62.443762 s
//////////////////
//pas = 4
//n=15, 0.322302 s
//n=20, 0.304270 s
//n=25, 0.319111 s
//n=30, 0.363637 s
//n=35, 0.422020 s
//n=40,  0.913839 s
//n=45, 4.076704 s
//n=50, 27.533862 s
//n=51, 37.631931 s
//n=52, 51.229920 s
//n=53, 74.701445 s
///////////////////
//pas = 5
//n=15, 0.331595 s
//n=20, 0.325551 s
//n=25, 0.331904 s
//n=30, 0.329586 s
//n=35, 0.370755 s
//n=40, 0.499660 s
//n=45, 0.875873 s
//n=50, 3.330401 s
//n=55, 15.230208 s
//n=56, 20.854270 s
//n=57, 28.793322 s
//n=58, 39.861761 s
```

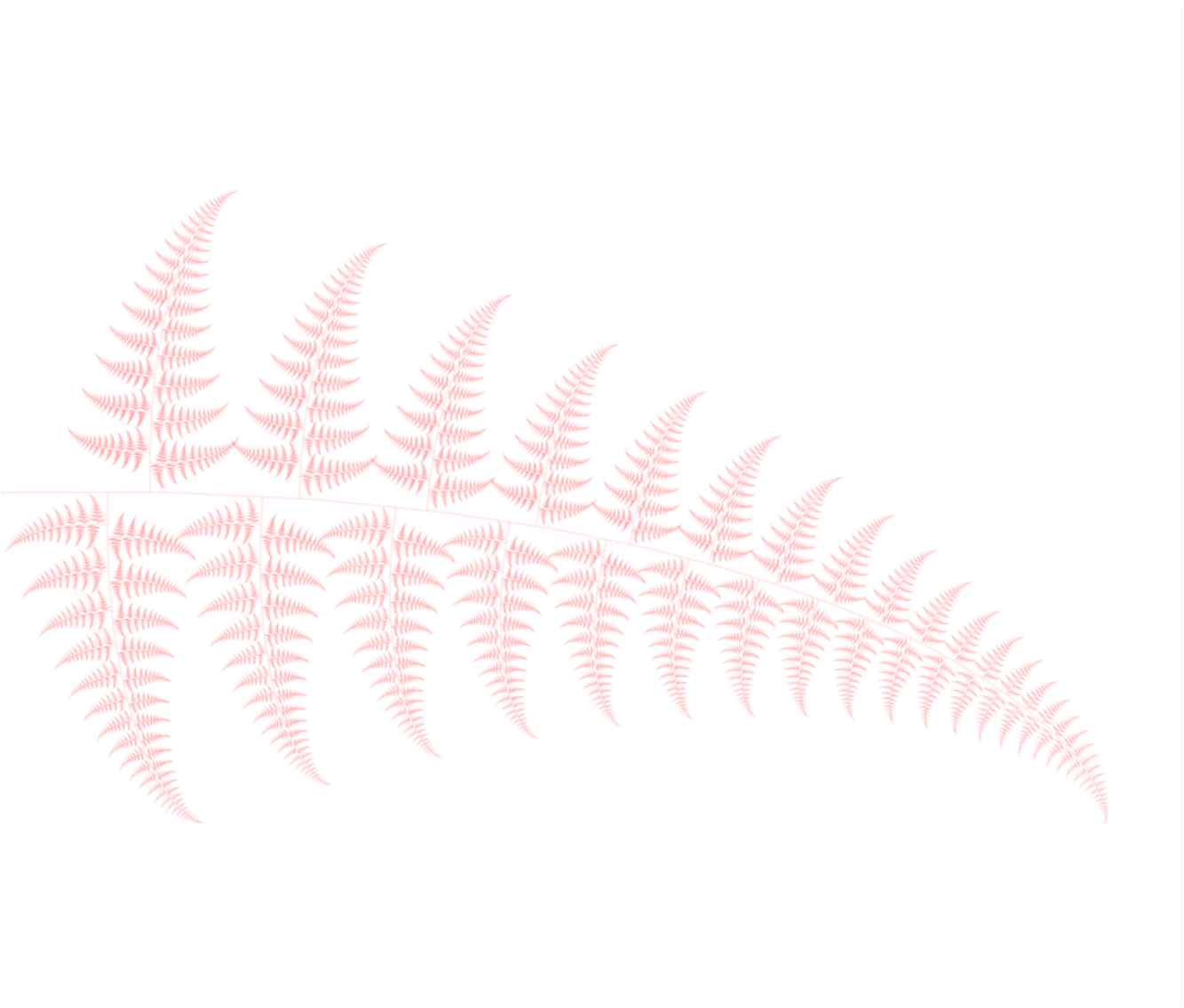

Figure 11: Fougère de taille 10000×10000 avec une échelle de 9000