Rapport de Recherche

Optimisation du tracé d'une fractale

VIENNE Mathys

Sommaire

Introduction

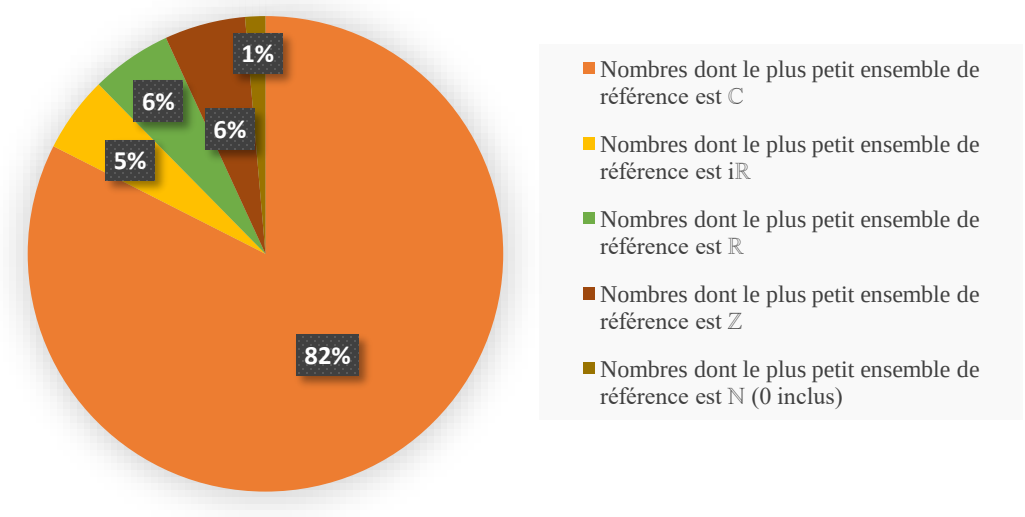
Notre étude portera ici sur l'analyse d'une fractale obtenue grâce à la base numérique $(i-1)$. Cette base numérique théorisée par S. Khmelnik[1] et Walter F. Penney[2] nous offre une écriture composée uniquement de 0 et de 1, à l'image de la base numérique 2. On dira alors que les mots résultants de la base $(i-1)$ sont binaires, même s'il ne s'agit pas de la base numérique 2. En effet, le seul fait qu'ils ne puissent être composés uniquement qu'avec deux digits différents suffit à leur attribuer cette dénomination. On peut alors, grâce à ce système, écrire des nombres complexes sous une forme binaire qu'un ordinateur pourrait tout à fait utiliser.

Nos ordinateurs, et plus généralement nos systèmes de stockage de données informatiques, fonctionnent sur 8 bits, ou un octet. Cela signifie que nos données sont codées en mots binaires composées de 8 caractères. Pour évaluer le potentiel de la base numérique $(i-1)$, une piste serait l'analyse de la répartition des nombres que l'on peut obtenir avec un codage sur 8 bits. Comme pour la base numérique 2, dans la mesure où nous ne possédons que deux digits pour composer nos mots binaires, le nombre total de nombres que l'on peut coder avec la base $(i-1)$ sur 8 bits sera de 2^8 nombres, soit 256 nombres au total. Notons E_n l'ensemble des nombres que l'on peut coder sur n bits en base numérique $(i-1)$. On aura donc $card(E_n) = 2^n$.

$$\forall\, z \in E_n, z = \sum_{m=0}^{(n-1)} x(i-1)^m \text{ , avec } x \in \{0; 1\}$$

Grâce à cette notation, il est possible de générer informatiquement E_8. Nous utilisons ici le langage de programmation Python dans sa version 3 pour réaliser cette tâche[3]. La génération nous permet alors de réaliser le diagramme circulaire suivant :

[1] Khmelnik, S.I. (1964). "Specialized digital computer for operations with complex numbers". Questions of Radio Electronics (En Russe). XII (2).
[2] W. Penney, A "binary" system for complex numbers, JACM 12 (1965) 247-248.
[3] Les programmes sont disponibles en annexe.



Ce diagramme nous montre la répartition des nombres codables dans la base numérique $(i–1)$. On peut observer que le nombre d'entiers naturels est petit comparé aux complexes. En fait, $E_8 \cap \mathbb{N} = \{0; 1; 2; 3\}$. Par exemple, le nombre 4 nécéssite 9 bits dans son écriture en base numérique $(i–1)$ car :

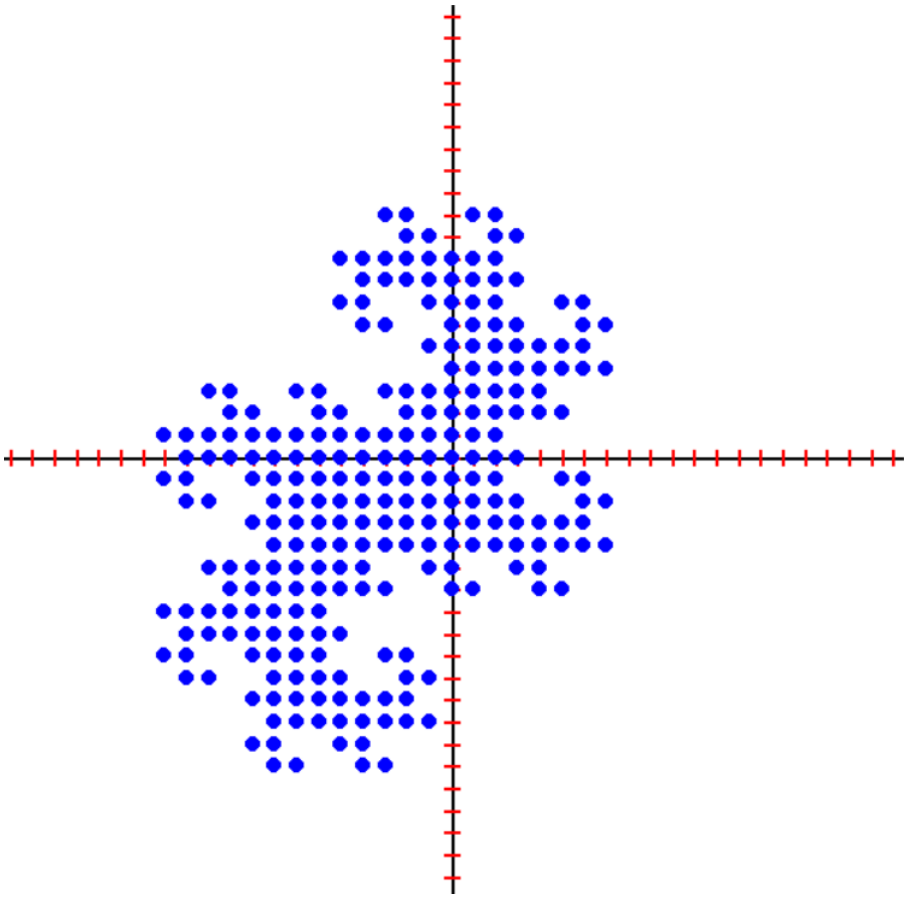
$$4 = 1 \times (i-1)^4 + 1 \times (i-1)^6 + 1 \times (i-1)^7 + 1 \times (i-1)^8$$

$$(4)_{10} = (1\ 1101\ 0000)_{(i-1)}$$

Donc $4 \notin E_8$ mais $4 \in E_9$.

On rappelle qu'un nombre complexe peut s'écrire sous la forme $a + ib, (a; b) \in \mathbb{R}^2$, et peut être placé dans un plan, le plan complexe.

Cette répartition inhabituelle des nombres que montre le diagramme, nous invite à nous demander quelle est la forme de E_8 lorsque son contenu représenté dans le plan complexe. Plaçons E_8 dans le plan complexe.



Nuage de points obtenu en plaçant les éléments de E_8 dans le plan complexe

La figure obtenue semble être une fractale. Elle est en réalite la fractale nommée « Twindragon », aussi appelée le « Davis-Knuth Dragon ». Comme l'avait montré D. Knuth[4], le placement des nombres de la base numérique $(i-1)$ nous permet d'obtenir cette figure. Cependant, l'algorithme utilisé ici pour générer E_8 a une complexité algorithmique $O(2^n)$, autrement dit, une complexité exponentielle. Cette méthode n'est donc pas optimale pour générer la figure. La questions à laquelle nous tenterons donc de répondre ici est la suivante : est-il possible de parvenir à optimiser l'algorithme générant cette fractale, jusqu'à une complexité algorithmique linéaire ? Pour répondre à cette problématique, nous allons analyser plus profondément E_n ainsi que son tracé dans le plan.

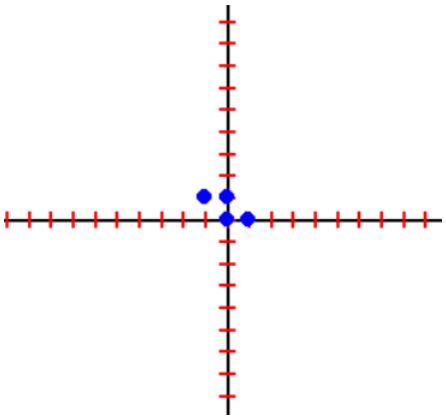
[4] Knuth, Donald (1998). "Positional Number Systems". The art of computer programming. Vol. 2 (3rd ed.). Boston: Addison-Wesley. p. 206

I – Analyse de l'ensemble de point

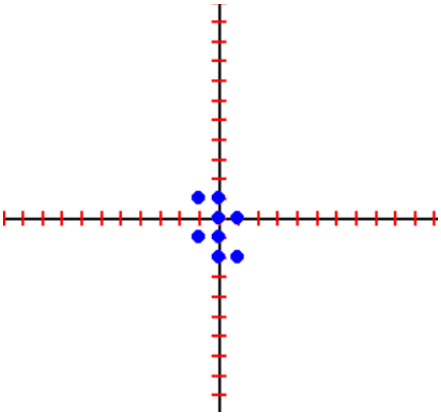
A – Modification du nombre de bits

Nous avons précédemment tracé E_8 dans un plan complexe et obtenu de ce qui semble être une fractale. La question que nous nous poserons maintenant est de savoir si la figure obtenue pour d'autres valeurs de n est toujours la même.

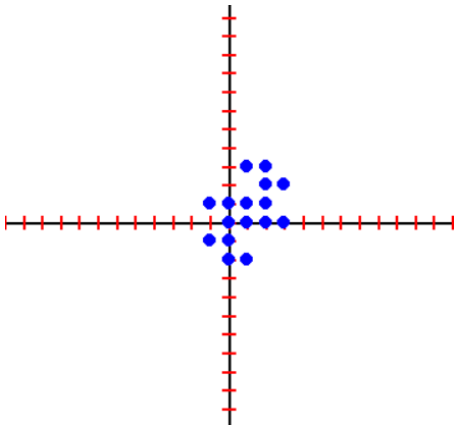
Pour commencer, on tracera pour des $n < 8$. On s'abstiendra de tracer E_0 car $card(E_0) = 0$.



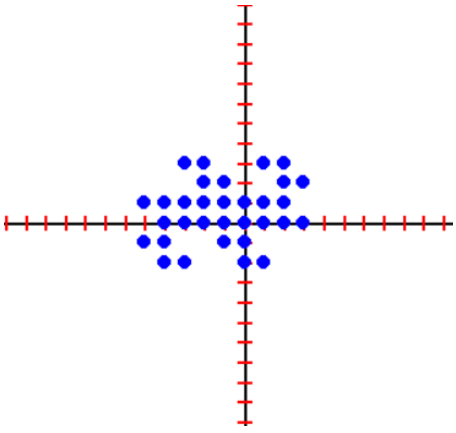
Représentation de E_2 dans le plan complexe



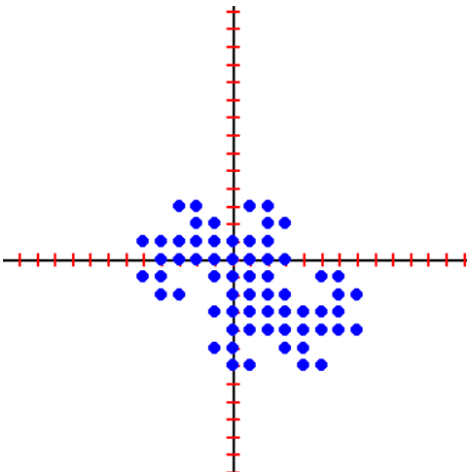
Représentation de E_3 dans le plan complexe



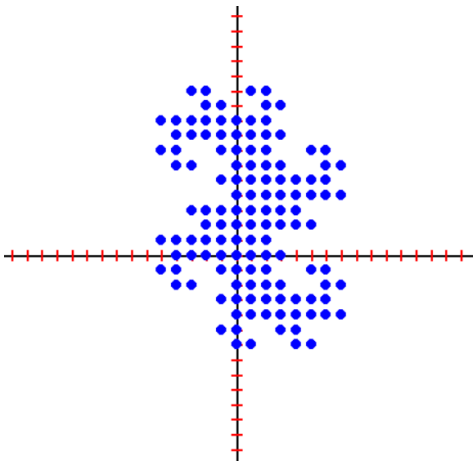
Représentation de E_4 dans le plan complexe



Représentation de E_5 dans le plan complexe



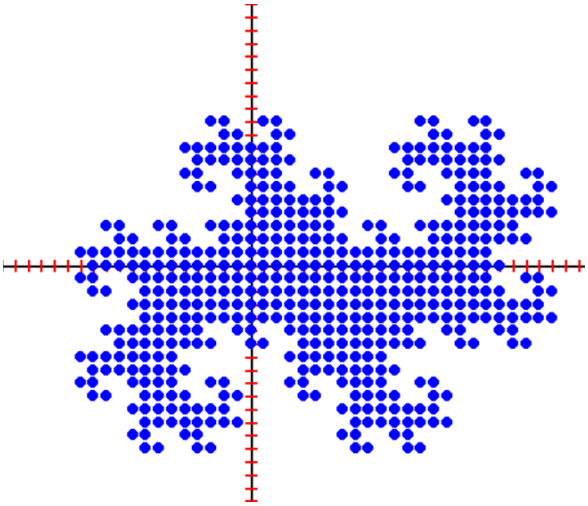
Représentation de E_6 dans le plan complexe



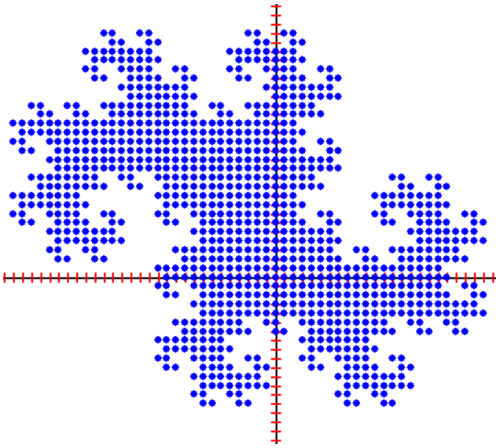
Représentation de E_7 dans le plan complexe

On voit sur ces représentations que la figure semble rester de la même forme. On peut aussi observer que la figure obtenue en représentant le contenu de E_{n+1} dans le plan semble être composée de deux fois la figure obtenue en représentant le contenu de E_n dans le plan, bout à bout. On ne démontre pas cette conjecture graphique ici.

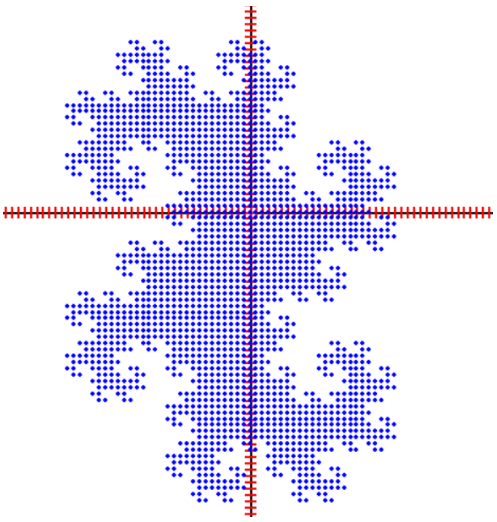
Procédons à une analyse similaire en représentant E_n pour des valeurs de n strictement supérieures à 8. L'algorithme actuel permettant de générer E_n possède une complexité algorithmique exponentielle. Par conséquent, la génération pour de grandes valeurs de n demande plus de ressources, des ressources dont je ne dispose pas. On montrera ici uniquement l'affichage de E_9, E_{10}, E_{11} et E_{12}.



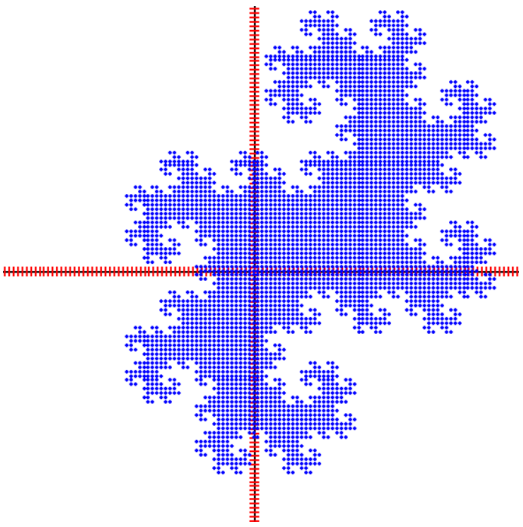
Représentation de E_9 dans le plan complexe



Représentation de E_{10} dans le plan complexe



Représentation de E_{11} dans le plan complexe



Représentation de E_{12} dans le plan complexe

Les conjectures faites sur des valeurs de n plus petites que 8 semblent aussi vraies avec des valeurs de n plus grandes que 8. On retrouve le même motif qui semble se détailler au fur et à mesure que n augmente. La figure E_{n+1} semble toujours être formée de deux figures E_n mises bout à bout. On peut donc constater le phénomène sur de plus grandes valeurs de n et ainsi conjecturer qu'il se produira $\forall \in \mathbb{N}^*$. Cependant, on ne proposera ici aucune démonstration.

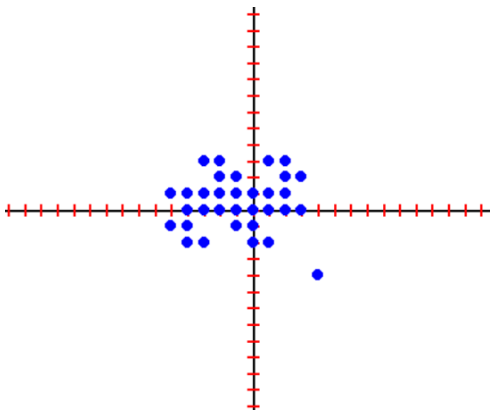
B – Unité de E_n

Nous avons jusqu'à maintenant uniquement représenté l'ensemble E_n. On rappelle que E_n est l'ensemble des nombres que l'on peut coder sur n bits en base numérique $(i-1)$. Lorsque l'on représentera les points de cet ensemble dans le plan complexe, nous obtiendrons toujours $card(E_n) = 2^n$ points. On a alors des figures dont les points sont tous réunis ensemble, de manière régulière, et sans laisser de trou. Nos analyses précédentes nous permettent de conjecturer que cela se produit pour la représentation des points de $E_n, \forall\, n \in \mathbb{N}^*$. On ne proposera pas de démonstration ici. On admet que ces propriétés sont vérifiées $\forall\, n \in \mathbb{N}^*$. La question que l'on peut se poser est la suivante : ces propriétés sont-elles vérifiées par la figure obtenue en représentant les $2^n + 1$ premiers nombres que l'on peut écrire en base numérique $(i-1)$, $n \in \mathbb{N}$. On se posera la même question pour réciproquement les $2^n - 1$ premiers nombres que l'on peut écrire en base numérique $(i-1), n \in \mathbb{N}$. Autrement dit, que se passe-t-il lorsque l'on affiche un point en plus ou un point en moins par rapport à E_n ?

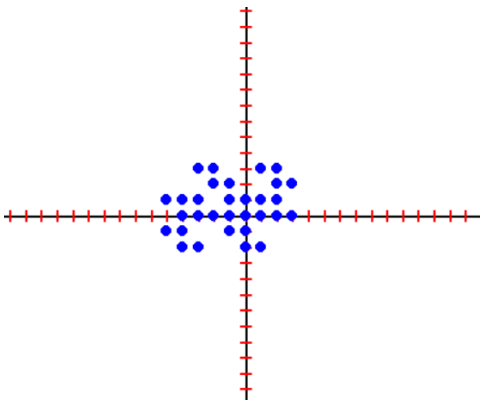
Afin d'effectuer des tests, on prend pour référence E_5 dont le cardinal est

$$card(E_5) = 2^5 = 32$$

Pour avoir la représentation de référence, voir la partie précédente (*I.A : Modification de taille*).



Représentation des 33 premiers nombres que l'on peut écrire en base numérique $(i-1)$ dans le plan complexe



Représentation des 31 premiers nombres que l'on peut écrire en base numérique $(i-1)$ dans le plan complexe

Concentrons-nous dans un premier temps sur le point en plus. On peut voir sur la première image que le point ajouté par rapport à la représentation de référence est situé au loin, brisant la structure régulière de la figure que nous donnait le placement des éléments de E_5. On peut, grâce à un l'affichage progressif des points, observer que lorsque l'on place l'ensemble des points E_6 dans l'ordre de la représentation binaire en base numérique $(i-1)$ de leurs affixes, on trace d'abord E_5 puis on vient tracer une seconde figure semblable qui viendra se coller à la première lorsqu'elle sera achevée à son tour. Il faudrait donc considérer ce point à l'écart comme le commencement d'un autre motif similaire au premier.

En ce qui concerne le point en moins, on voit un trou se former au sein de la figure de référence. Si on fait le lien entre les observations faites sur le point en plus et cette situation, on peut déduire que nous avons retiré le point qui permettait de passer définitivement de E_4 à E_5, et que nous avons par la même occasion ramené la figure dans un stade intermédiaire qui ne vérifie aucune des propriétés que nous souhaitions tester.

On peut donc en conjecturer que nos propriétés ne se vérifient que pour E_n, les autres laissant des représentations graphiques intermédiaires entre deux états plus « stables ».

II – Prévoir le contenu de l'ensemble

Dans cette partie, nous allons tenter de trouver des liens mathématiques nous permettant de prévoir le contenu de l'ensemble E_n. Pour cela, il nous faut comprendre le fonctionnement de la figure fractale ainsi que sa périodicité.

A – Motif élémentaire

Nous avons précédemment observé que, graphiquement, E_{n+1} était un regroupement de deux E_n. On peut donc remonter les valeurs de n jusqu'à trouver le plus petit motif. On met toujours de côté E_0 en raison de son cardinal nul. On trouve alors E_2 avec $card(E_2) = 4$ qui est représenté graphiquement par ce qui est un parallélogramme[5].

Démonstration : Rappelons que $E_2 = \{0;\ 1;\ (i-1);\ i\}$

Soient Z_n les points d'affixe z_n n-ième élément de E_2, $z_{\overrightarrow{Z_1Z_2}}$ l'affixe du vecteur $\overrightarrow{Z_1Z_2}$ et $z_{\overrightarrow{Z_3Z_4}}$ l'affixe du vecteur $\overrightarrow{Z_3Z_4}$.

$z_{\overrightarrow{Z_1Z_2}} = z_2 - z_1 = 1 - 0 = 1$

$z_{\overrightarrow{Z_3Z_4}} = z_4 - z_3 = i - (i-1) = i - i + 1 = 1$

On a donc $z_{\overrightarrow{Z_1Z_2}} = z_{\overrightarrow{Z_3Z_4}}$

Donc E_2 est un parallélogramme.

Lorsque n augmente, on remarque que le parallélogramme que l'on observe en E_2 semble se répéter à différents endroits du plan, ce qui formerait la figure E_n. On le voit dans les représentations générées pour la partie précédente, le parallélogramme est présent partout. L'utilisation d'un affichage ralenti point par point dans l'ordre des représentations binaires des nombres en base $(i-1)$ nous apprend que la figure se trace parallélogramme par parallélogramme. On admet que E_n est formé de parallélogrammes de la forme de E_2 situés à différents endroits du plan.

[5] On ne prend pas E_1 car la structure en parallélogramme qui semble se répéter n'apparait qu'à partir de E_2.

On a donc $\forall\, k \in \mathbb{N}, P_k = \{z_{4k}\,;\, z_{4k+1}\,; z_{4k+2}\,; z_{4k+3}\}$ avec $E'_n = (z_0;\, z_1;\, z_2; \ldots; z_n)$, le k-ième parallélogramme formant E_n.

On a alors les propriétés suivantes :

- Pour tout entier naturel $n \geq 2$, on peut écrire E_n comme suit :

$$E_n = \bigcup_{i=1}^{\frac{2^n}{4}} P_i$$

- $\forall\, k \in \mathbb{N}, P_k = \{z;\, z+1; z+(i-1); z+i\}$ avec z l'affixe du premier point d'un parallélogramme.

Démonstrations :

- $\forall\, k \in \mathbb{N}, P_k$ est le k-ième parallélogramme formant E_n.
  On a donc $card(P_k) = 4, \forall\, k \in \mathbb{N}$.

  Or $card(E_n) = 2^n$. $\forall\, n \geq 2, E_n$ est formé de groupes P_k de 4 éléments, donc il y a $\frac{2^n}{4}$ parallélogrammes dans E_n. On peut donc écrire

$$\forall\, n \geq 2, E_n = \bigcup_{i=1}^{\frac{2^n}{4}} P_i$$

- On a identifié $E_2 = \{0;\, 1;\, (i-1);\, i\} \Leftrightarrow \{z; z+1; z+(i-1); z+i\}$ avec z = 0

  P_k est par définition de la même forme que E_2 dans la mesure où les points des affixes contenus dans P_k sont disposés entre eux de la même manière que les points des affixes contenus dans E_2.

Donc avec z l'affixe du premier (dans l'ordre croissant de leurs représentations binaires en base numérique $(i-1)$) point d'un parallélogramme de P_k,

$$P_k = \{z;\, z+1; z+(i-1); z+i\}$$

Pour la suite, on admettra que toutes les propriétés conjecturées jusqu'ici sont vraies. On a donc identifié un motif élémentaire se répétant à différents endroits du plan. Il nous faut maintenant déterminer où commence chaque parallélogramme.

B – Placement des parallélogrammes

On a z_m le m-ième élément de E'_n en partant de m = 0. On a donc Δ_0 l'ensemble des premiers (dans l'ordre croissant de leur représentation binaire en base numérique $(i-1)$) points de parallélogramme P_k.

$$\Delta_0 = \{z_{4m}, m \in \mathbb{N}\}$$

On peut, grâce à cette expression, introduire la notion de « période ». La période j est l'ensemble Δ_j tel que

$$\Delta_j = \{z_{4^j m}, m \in \mathbb{N}\}, \forall\, j \in \mathbb{N}$$

Par exemple :

$\Delta_0 = \{0\ ;\ 1\ ;\ (i-1)\ ;\ i\ ;\ \dots\} = E_n$

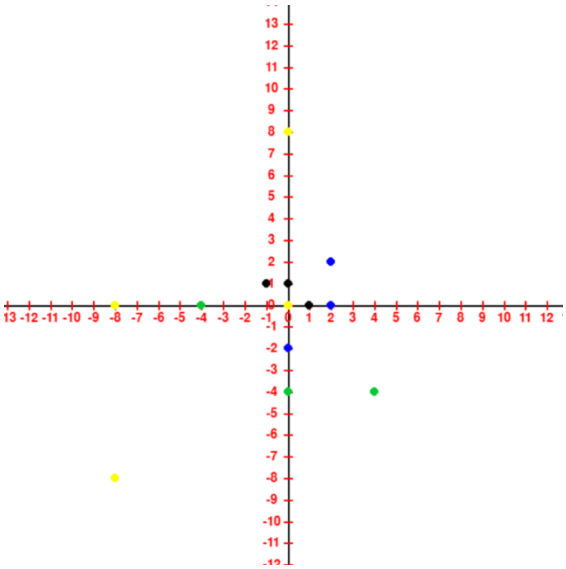
$\Delta_1 = \{0\ ;\ -2i\ ;\ (2+2i)\ ;\ 2\ ;\ \dots\}$

$\Delta_2 = \{0;\ -4;\ (4-4i);\ -4i;\ \dots\}$

$\Delta_3 = \{0; 8i; (-8-8i); -8; \dots\}$

Plaçons dans le plan les 4 plus petits éléments[6] de Δ_0, Δ_1, Δ_2 et Δ_3 dans l'ordre de leurs représentations binaires en base numérique $(i-1)$.

[6] Si on affiche tous les éléments des ensembles, il sera plus difficile d'observer les parallélogrammes. Autrement, les ensembles semblent contenir des parallélogrammes qui semblent se répéter de la même manière que P_k. On admet que phénomène se produit pour tous les Δ_j sans proposer de démonstration.



Δ_0 en noir, Δ_1 en bleu, Δ_2 en vert, Δ_3 en jaune

On peut observer graphiquement que les parallélogrammes formés par les éléments Δ_1 sont les mêmes que ceux de Δ_0 après avoir subi une rotation de $-\frac{\pi}{2}$ radians ainsi qu'une homothétie de rapport 2. On admet que $\forall\, j \in \mathbb{N}$, Δ_{j+1} est Δ_j qui a subi une homothétie de rapport 2 et une rotation de $-\frac{\pi}{2}$ radians sans proposer de démonstration.

On obtient alors la formule de z_{4m}

$$\forall\, m \in \mathbb{N}, z_{4m} = -2iz_m$$

Démonstration :

Δ_0 est l'ensemble des points z_m.

On peut représenter z_m sous la forme d'une matrice.

$$\forall\, m \in \mathbb{N}, z_m = \begin{pmatrix} Re(z_m) \\ Im(z_m) \end{pmatrix}$$

On vient appliquer la rotation de $-\frac{\pi}{2}$ radians et l'homothétie de rapport 2 à z_m pour passer de Δ_0 à Δ_1 et ainsi obtenir l'ensemble des points z_{4m}

$$\forall\, m \in \mathbb{N}, z_{4m} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \times \begin{pmatrix} Re(z_m) \\ Im(z_m) \end{pmatrix}$$

$$z_{4m} = \begin{pmatrix} 0 & 2 \\ -2 & 0 \end{pmatrix} \times \begin{pmatrix} Re(z_m) \\ Im(z_m) \end{pmatrix}$$

$$z_{4m} = \begin{pmatrix} 2Im(z_m) \\ -2Re(z_m) \end{pmatrix}$$

$$z_{4m} = 2Im(z_m) - 2iRe(z_m)$$
$$z_{4m} = -2i\left(\frac{-Im(z_m)}{i} + Re(z_m)\right)$$
$$z_{4m} = -2i\left(-Im(z_m) \times \frac{1}{i} + Re(z_m)\right)$$
$$z_{4m} = -2i\big(-Im(z_m) \times (-i) + Re(z_m)\big)$$
$$z_{4m} = -2i\big(Re(z_m) + i\ Im(z_m)\big)$$
$$z_{4m} = -2iz_m$$

On a donc une expression de z_{4m} en fonction de z_m. À partir d'un nombre $z_{4m} \in \Delta_1$, on peut générer quatre nombres $z_m \in E_n$, le parallélogramme correspondant. Par conséquent, nous avons suffisamment de données pour tracer la fractale avec un algorithme de complexité O(n)

Conclusion

Construisons l'algorithme permettant de générer E_n.

Le principe de l'algorithme est simple. On sait que $z_0 = 0$. On peut donc calculer P_1 et ainsi obtenir les quatre premiers points de l'ensemble. On va ensuite prendre le z_1 pour calculer $z_{4\times1} = z_4$ pour ensuite calculer P_2 à partir de ce nombre. On répète le même principe jusqu'à obtenir le nombre d'éléments souhaités.

Entrée : t la taille en point de la fractale

Sortie : E un tableau contenant tous les points de la fractale

E ← générer_parallélogramme(0)

n ← 1

Tant que nombre d'éléments de E < t faire

z4m ← E[n] × (-2i)

E = E + générer_parallélogramme(z4m)

n = n + 1

Fin Tant que

Si nombre d'éléments de E ≠ t alors

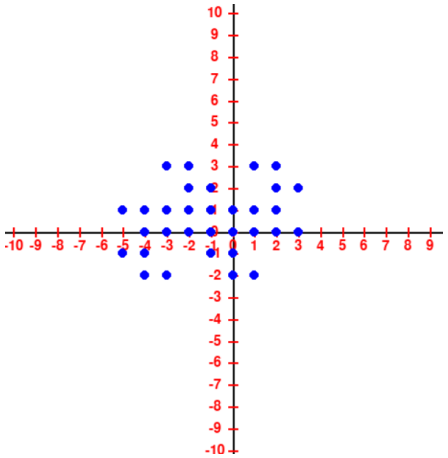
Renvoyer E[0 à t]

Cet algorithme peut se coder de la façon suivante en Python 3 :

```
def generer_parallelogramme(point: Nombre) -> list:
      """
      Fonction qui retourne le parallélogramme commençant par un point dans
      la fractale
      Args:
      point: <Nombre>, le point de départ du parallélogramme
      Returns: <list>, le parallélogramme
      """
      return [point, point+Nombre(1, 0), point+Nombre(-1, 1),
      point+Nombre(0, 1)]


def creer_fractale(taille: int) -> list:
      """
      Fonction permettant la génération d'un ensemble de points fractal
      Args:
      taille: <int>, le nombre de point qui va être généré
      Returns: <list>, la fractale composée du nombre de points taille
      """
      E = generer_parallelogramme(Nombre(0, 0))
      n = 1
      while len(E) < taille:
            z4m = Nombre(0, -2)*E[n]
            E += generer_parallelogramme(z4m)
            n += 1
      return E[:taille]
```

Nous avons donc un programme fonctionnel de complexité O(n) qui va nous permettre de générer les points de notre fractale. Voilà le résultat obtenu en affichant `creer_fractale(32)` dans un plan.



Nous avons bien obtenu E_5 donc le programme, ainsi que tous les raisonnements effectués, semblent corrects.

En conclusion, c'est en analysant et tentant de comprendre en profondeur la figure que nous sommes parvenus à saisir les liens et la périodicité propres celle-ci. C'est de cette façon que nous avons pu optimiser la génération de la fractale. Néanmoins, c'est une conclusion qui ouvre aussi de nombreuses portes, que ce soit au niveau du rôle du premier point de la fractale lors du tracé, mais aussi au niveau de toutes les propriétés qui ont été admises restant ici sans démonstration.

Annexe

- Classe Nombre utilisée afin de manipuler informatiquement des nombres complexes.

```
from math import sqrt, acos


class Nombre:
    """
    Classe représentant des nombres complexes de la forme a+ib où a et b
sont des nombres réels.
    """
    def __init__(self, r: float, im: float):
        """
        :param r: Partie réelle du nombre complexe, un <float>
        :param im: Partie Imaginaire du nombre complexe, un <float>
        """
        self.R = r # Re(a+ib)
        self.IM = im # Im(a+ib)

        self.module = sqrt(self.R**2 + self.IM**2)
        if self.IM > 0:
            self.argument = acos(self.R/self.module)
        elif self.IM > 0:
            self.argument = -1*(acos(self.R/self.module))
        else:
            self.argument = 0

    def __eq__(self, other):
        """
        Méthode traitant l'égalité de deux nombres.
        Deux complexes sont égaux lorsque leurs parties réelles et
        imaginaires sont égales respectivement.
        :param other: <Nombre>
        :return: <bool>
        """
        if self.R == other.R and self.IM == other.IM:
            return True
        else:
            return False
```

```
def __add__(self, other):
        """
        Méthode traitant l'addition de deux nombres entre-eux.
        Pour additionner deux nombres, il faut additionner les parties
        réelles entre-elles et faire de même avec les
        parties imaginaires.
        :param other: <Nombre>
        :return: <Nombre>
        """
        a = self.R + other.R
        b = self.IM + other.IM
        return Nombre(a, b)

    def __sub__(self, other):
        """
        Méthode traitant la soustraction de nombres entre-eux.
        Pour soustraire deux nombres, il faut soustraire les parties réelles
        entre-elles et faire de même avec les
        parties imaginaires.
        :param other: <Nombre>
        :return: <Nombre>
        """
        a = self.R - other.R
        b = self.IM - other.IM
        return Nombre(a, b)

    def __mul__(self, other):
        """
        Méthode qui traite la multiplication de nombres entre-eux.
        Pour multiplier, on utilise les règles de la double distributivité
        ainsi que la propriété [i² = -1].
        :param other: <Nombre>
        :return: <Nombre>
        """
        a = (self.R * other.R) - (self.IM * other.IM)
        b = (self.R * other.IM) + (self.IM * other.R)
        return Nombre(a, b)

    def __truediv__(self, other):
        """
        Méthode qui traite la division d'un nombre par un autre.
        :param other: <Nombre>
        :return: <Nombre>
        """
        a = ((self.R*other.R)+(self.IM*other.IM))/(other.R**2 + other.IM**2)
        b = (-1*(self.R*other.IM)+(other.R*self.IM))/(other.R**2 +
other.IM**2)
        return Nombre(a, b)
```

```
def __pow__(self, power):
        """
        Méthode qui traite les puissances d'un nombre.
        Pour effectuer cela, on procède par multiplications successives.
        :param power: <int>
        :return: <Nombre>
        """
        if power == 0:
            return Nombre(1, 0)

        nombre = Nombre(self.R, self.IM)
        puissance_pos = Nombre(self.R, self.IM)
        compteur = 1
        while compteur < abs(power):
            puissance_pos = puissance_pos * nombre
            compteur += 1

        if power > 0:
            return puissance_pos

        else:
            retour = Nombre(1, 0)/(puissance_pos**abs(power))
            return retour

    def __str__(self):
        """
        Méthode qui traite l'affichage du nombre en <str>
        :return: <str>
        """
        if self.IM != 0 and self.R != 0:
            if self.IM >= 0:
                return f"{self.R}+{self.IM}i"

            else:
                return f"{self.R}{self.IM}i"

        elif self.IM != 0 or self.R != 0:
            if self.R == 0:
                return f"{self.IM}i"
            else:
                return f"{self.R}"

        else:
            return "0"
```

- Classes Plan et Set qui ont servi pour les représentations graphiques

```
import pygame


class Set:
    """Classe modélisant un esemble de points"""
    def __init__(self, points, couleur, point_rayon, ralenti, nom,
complexe):
        self.name = nom
        self.points = points
        self.couleur = couleur
        self.point_rayon = point_rayon
        self.ralenti = ralenti
        self.complexe = complexe


class Plan:
    """Classe modélisant un plan"""
    def __init__(self, longueur: int, hauteur: int, x_min: int, x_max: int,
y_min: int, y_max: int, obj=[], graduation=False):
        """
        Initialisation du Plan
        Args:
            longueur: <int>, la longueur de la fenêtre
            hauteur: <int>, la hauteur de la fenêtre
            x_min: <int>, la graduation minimum sur l'axe horizontal
            x_max: <int>, la graduation maximum sur l'axe horizontal
            y_min: <int>, la graduation minimum sur l'axe vertical
            y_max: <int>, la graduation maximum sur l'axe vertical
            obj: <list>, la liste des objets à dessiner sur le plan
            graduation: <bool>, True si on souhaite afficher les valeurs des
             gratuations, False sinon
        """
        pygame.init()
        self.longueur = longueur
        self.hauteur = hauteur
        self.x_min = x_min
        self.x_max = x_max
        self.y_min = y_min
        self.y_max = y_max
        self.obj = obj
        self.graduation = graduation
        print("Repère créé avec succès.")
```

```
    def draw(self):
        """
        Méthode dessinant le plan
        Sortie: Aucune
        """

        # Création d'une fenêtre Pygame
        self.fenetre = pygame.display.set_mode((self.longueur,
self.hauteur))
        pygame.display.set_caption("Plan")
        self.fenetre.fill((255, 255, 255))

        # On dessine les axes
        pygame.draw.line(self.fenetre, (0, 0, 0), (0, self.hauteur / 2),
(self.longueur, self.hauteur / 2), 2)
        pygame.draw.line(self.fenetre, (0, 0, 0), (self.longueur / 2, 0),
(self.longueur / 2, self.hauteur), 2)

        # On dessine les graduations pour l'axe horizontal
        for i in range(self.x_min, self.x_max + 1):
            x = self.longueur / 2 + i * (self.longueur / (self.x_max -
self.x_min + 1))
            pygame.draw.line(self.fenetre, (255, 0, 0), (x, self.hauteur / 2
- 5), (x, self.hauteur / 2 + 5), 2)

            # Si on souhaite afficher les valeurs des graduations
            if self.graduation:
                font_size = self.longueur // (self.x_max - self.x_min)
                if font_size > 15:
                    font_size = 15

                font = pygame.font.Font('freesansbold.ttf', font_size)
                text = font.render(str(i), True, (255, 0, 0))
                text_rect = text.get_rect()
                text_rect.center = (x, self.hauteur / 2 + 15)
                self.fenetre.blit(text, text_rect)

        # On dessine les graduations pour l'axe vertical
        for j in range(self.y_min, self.y_max + 1):
            y = self.hauteur / 2 - j * (self.hauteur / (self.y_max -
self.y_min + 1))
            pygame.draw.line(self.fenetre, (255, 0, 0), (self.longueur / 2 -
5, y), (self.longueur / 2 + 5, y), 2)
```

```
            # Si on souhaite afficher les valeurs des graduations
            if self.graduation:
                font_size = self.hauteur // (self.y_max - self.y_min)
                if font_size > 15:
                    font_size = 15
                font = pygame.font.Font('freesansbold.ttf', font_size)
                text = font.render(str(j), True, (255, 0, 0))
                text_rect = text.get_rect()
                text_rect.center = (self.longueur / 2 - 20, y)
                self.fenetre.blit(text, text_rect)

        # On dessine les ensembles de points
        for sets in self.obj:
            for point in sets.points:
                if sets.complexe:
                    x = self.longueur / 2 + point.R * (self.longueur /
(self.x_max - self.x_min + 1))
                    y = self.hauteur / 2 - point.IM * (self.hauteur /
(self.y_max - self.y_min + 1))
                else:
                    x = self.longueur / 2 + int(point[0]) * (self.longueur /
(self.x_max - self.x_min + 1))
                    y = self.hauteur / 2 - int(point[1]) * (self.hauteur /
(self.y_max - self.y_min + 1))

                pygame.draw.circle(self.fenetre, sets.couleur, (int(x),
int(y)), sets.point_rayon)

                # Si on souhaite un ralenti/un affichage point par point
                if sets.ralenti:
                    pygame.time.wait(1000)
                    pygame.display.update()

        # On actualise la fenêtre
        pygame.display.flip()
        print('Repère affiché avec succès.')

        # Boucle d'événement
        running = True
        while running:
            for event in pygame.event.get():
                # Si on tente de fermer la fenêtre
                if event.type == pygame.QUIT:
                    running = False

        # On quitte pygame
        pygame.quit()
```

```
    def add_points(self, ensemble_point: list, point_rayon=5, couleur=(0, 0,
255), ralenti=False, nom="", complexe=False):
        """
        Méthode permettant l'ajout d'ensembles de points <Set> au plan.
        Args:
            ensemble_point: <list>, l'ensemble de point à ajouter
            point_rayon: <int>, le rayon des points à afficher
            couleur: <tuple> de la forme (x, y, z) avec x, y et z trois
            <int> compris entre 0 et 255 inclus
            ralenti: <bool>, active ou non le ralenti sur l'affichage des
            points
            nom: <str>, nom de l'ensemble de points
            complexe: <bool>, True si le contenu de ensemble_point est de
            type <Nombre>, False si ce sont des <tuple>

        Sortie: Aucune

        """
        # On ajoute le nouvel ensemble à la liste d'objets à afficher
        self.obj.append(Set(ensemble_point, couleur=couleur,
point_rayon=point_rayon, ralenti=ralenti, nom=nom, complexe=complexe))
        print(f"Points du set {nom} ajoutés avec succès.")
```

- Programme utilisé lors de l'introduction pour générer E_n

```
from nombre import Nombre
from plan import Plan


def conversion(mot: str) -> Nombre:
    """
    Fonction permettant la conversion de la base numérique (i-1) vers la
    base numérique 10
    Args:
        mot: <str>, mot binaire représentant un nombre en base (i-1)

    Returns: <Nombre>, la représentation en base numérique 10 du nombre
    """
    # Initialisation d'un compteur
    r = Nombre(0, 0)

    # On retourne le mot binaire pour utiliser l'écriture positionnelle
    mot_retourne = mot[::-1]
```

```
# On vient analyser chaque chiffre du mot binaire retourné
    for i in range(len(mot)):
        if mot_retourne[i] == "1":
            # On ajoute au compteur la puissance de (i-1) correspondante
            r += Nombre(-1, 1)**i
    return r


# On crée l'ensemble de points E_8
E_8 = [conversion(bin(i)[2:]) for i in range(2**8)]

# On crée un plan
plan_complexe = Plan(600, 600, -20, 20, -20, 20)

# On ajoute l'ensemble de points E_8 au plan
plan_complexe.add_points(ensemble_point=E_8, nom="E_8", complexe=True,
point_rayon=5)

# On dessine le plan
plan_complexe.draw()
```