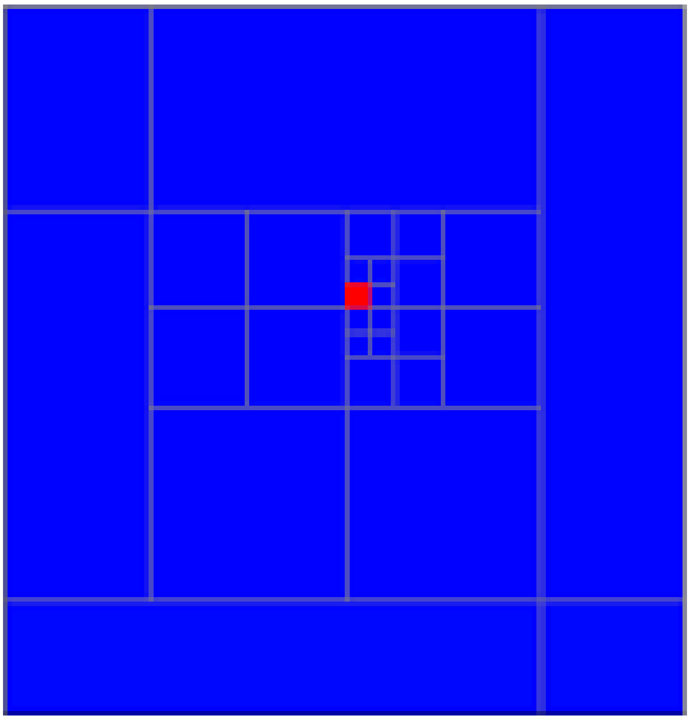
ZILLIOX Théo / MASSON Basile
GE3

PROJET ANALYSE : Algorithme de Weyl



Part 1 : Résultats théoriques

1) On définit : P = $a_nX^n + a_{n-1}X^{n-1} + \ldots + a_1X + a_0$, en supposant que $a_n \neq 0$ et $a_0 \neq 0$.

Soit Z_0 une racine de P, on veut montrer que $\left|Z_0\right| \leq 1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$

On a : $P(Z_0) = 0$

$$\Rightarrow \sum_{k=0}^{n} a_k z_0^k = 0 \Rightarrow a_n z_0^n + a_{n-1} z_0^{n-1} + \ldots + a_1 z_0 + a_0$$

$$\Rightarrow -a_n z_0^n = a_{n-1} z_0^{n-1} + \ldots + a_1 z_0 + a_0$$

$$\Rightarrow \left|a_n z_0^n\right| = \left|a_{n-1} z_0^{n-1} + \ldots + a_1 z_0 + a_0\right|$$

$$\Rightarrow \left|a_n z_0^n\right| \leq \left|a_{n-1} z_0^{n-1}\right| + \ldots + \left|a_1 z_0\right| + \left|a_0\right| \text{ (inégalité triangulaire sur les modules)}$$

$$\Rightarrow \left|a_n\right|\left|z_0\right|^n \leq \sum_{i=0}^{n-1}\left(\left|a_i\right| * \left|z_0\right|^i\right)$$

$$\Rightarrow \left|a_n\right|\left|z_0\right|^n \leq \sum_{i=0}^{n-1}\left(max_{i<n}\left|a_i\right| * \left|z_0\right|^i\right)$$

$$\Rightarrow \left|a_n\right|\left|z_0\right|^n \leq max_{i<n}\left|a_i\right| \sum_{i=0}^{n-1}\left|z_0\right|^i$$

$$\Rightarrow \left|a_n\right|\left|z_0\right|^n \leq max_{i<n}\left|a_i\right| * \frac{1-\left|z_0\right|^n}{1-\left|z_0\right|}$$

- <u>Soit $|z_0|>1$: $(1-|z_0|<0)$</u>

$$\Rightarrow \left|a_n\right|(1 - \left|z_0\right|)\left|z_0\right|^n \geq max_{i<n}\left|a_i\right| * (1 - \left|z_0\right|^n)$$

$$\Rightarrow \left|a_n\right|(1 - \left|z_0\right|) \geq max_{i<n}\left|a_i\right| * \left(\frac{1}{\left|z_0\right|^n} - 1\right)$$

$$\Rightarrow (1 - \left|z_0\right|) \geq \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|} * \left(\frac{1}{\left|z_0\right|^n} - 1\right)$$

Par ailleurs $\frac{1}{\left|z_0\right|^n} - 1 > -1$,

$$\Rightarrow -\left|z_0\right| \geq -\frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|} - 1$$

Finalement : $\left|z_0\right| \leq 1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$

- Soit $|z_0|\leq 1$: donc trivialement,

$$\left|z_0\right| \leq 1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$$

On a pu montrer que si z_0 est une racine de P alors l'inégalité $\left|z_0\right|\leq 1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$ est vérifiée. On a $\left|z_0\right| = \sqrt{a^2 + b^2}$, le module du nombre complexe $z_0 = a + ib$.

On peut donc réécrire, $\sqrt{a^2 + b^2}\leq 1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$

Ou encore, $\sqrt{(a - 0)^2 + (b - 0)^2}\leq 1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$

On reconnaît l'équation du disque de centre 0 et de rayon $1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$ dans le plan complexe. On en déduit que les racines du polynôme évaluées en z_0 appartiennent à ce disque de centre 0 et de rayon $1 + \frac{max_{i<n}\left|a_i\right|}{\left|a_n\right|}$ sur le plan complexe.

2) Soit $z_1 \in C$,

On peut écrire :

$$(\Delta) = P\left(z_1, Y\right) = P\left(z_1 + Y\right) = b_0 + b_1 Y + b_2 Y^2 + ... + b_n Y^n$$

Soit r > 0.

On dispose $\forall a, b \in C$ de :

$$|a| + |b| > |a + b| > |a| - |b|$$

On note la première inégalité (1) et la seconde (2).

$$(\Delta) \Rightarrow \left|P\left(z_1 + Y\right)\right| = \left|b_0 + b_1 Y + b_2 Y^2 + ... + b_n Y^n\right|$$

$\left|P\left(z_1 + Y\right)\right| \geq \left|b_0\right| - \left|b_1 Y + b_2 Y^2 + ... + b_n Y^n\right|$ d'après (2)

$\left|P\left(z_1 + Y\right)\right| \geq \left|b_0\right| - \left[\left|b_1 Y\right| + \left|b_2 Y^2\right| + ... + \left|b_n Y^n\right|\right]$ d'après (1)

$\left|P\left(z_1 + Y\right)\right| \geq \left|b_0\right| - \left[\left|b_1\right||Y| + \left|b_2\right|\left|Y^2\right| + ... + \left|b_n\right|\left|Y^n\right|\right]$ par prop de |.|

Si $|Y| \leq r \Leftrightarrow \left|Y^k\right| \leq r^k$ on obtient,

$$(\emptyset) = \left|P\left(z_1 + Y\right)\right| \geq \left|b_0\right| - \left[\left|b_1\right|r + \left|b_2\right|r^2 + \ldots + \left|b_n\right|r^n\right]$$

Si (d'après notre hypothèse) :

$$\left|b_0\right| > \left|b_1\right|r + \left|b_2\right|r^2 + \ldots + \left|b_n\right|r^n$$

$$\Rightarrow \left|b_0\right| - (\left|b_1\right|r + \left|b_2\right|r^2 + \ldots + \left|b_n\right|r^n) > 0$$

Si $|Y| \leq r$:

$$(\emptyset) \Rightarrow \left|P\left(z_1 + Y\right)\right| \geq \left|b_0\right| - (\left|b_1\right|r + \left|b_2\right|r^2 + \ldots + \left|b_n\right|r^n) > 0$$

$$\Rightarrow \left|P\left(z_1 + Y\right)\right| > 0$$

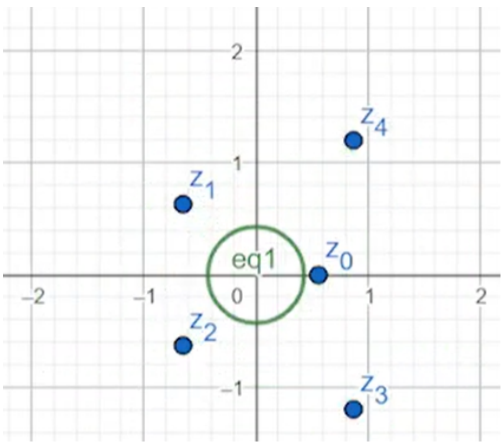
$$\Rightarrow \left|P\left(z_1 + Y\right)\right| \neq 0$$

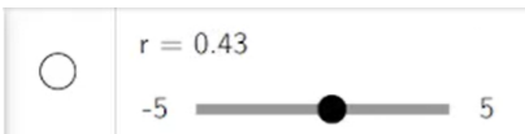
Ainsi $\forall |Y| \leq r$, P n'a pas de racines dans le disque de centre z_1 et de rayon r.

3) Pour le polynôme $Q = X^5 - X^4 + X^3 - X^2 - 1$ On souhaite trouver une valeur de r tel que l'inégalité (1) est vérifiée pour un cercle de centre O.

On résout donc l'équation $1 < r + r^2 + r^3 + r^4 + r^5$ sur WolframAlpha et l'inégalité est vérifiée pour r>0.50866. Donc il faut prendre une valeur de r inférieure.

Ici, r = 0,43 convient.





4) On peut développer l'expression de $P\left(z_1 + Y\right)$ par la formule de Taylor :

$$P\left(z_1 + Y\right) = P\left(z_1\right) + P'\left(z_1\right)Y + \frac{Y^2}{2!}P''\left(z_1\right)\frac{Y^2}{2!} + \ldots + \frac{Y^n}{n!}P^{(n)}\left(z_1\right) + o(Y^n)$$

Or pour les polynômes, $o(Y^n) = 0$.

$$P\left(z_1 + Y\right) = \sum_{k=0}^{n} \frac{Y^k}{k!}P^{(k)}\left(z_1\right)$$

$$P(z_1 + Y) = b_0 + b_1Y + b_2Y^2 + \ldots + b_nY^n$$

On pourra plus spécifiquement écrire les coefficients b_i :

$$\Rightarrow b_i = \frac{P^{(i)}(z_1)}{i!}$$

$$\Rightarrow \left|b_i\right| = \left|\frac{P^{(i)}(z_1)}{i!}\right|$$

$$\Rightarrow \left|b_i\right| = \frac{\left|P^{(i)}(z_1)\right|}{i!}$$

Part 2 : Complexes et polynômes en python

Nous allons dans cette partie créer les programmes pythons qui réalisent différents opérateurs mathématiques de base sur les nombres complexes afin d'être ensuite utilisés dans la 3ème partie.

On pose $z = a + ib \in C$, on le définit en python par le couple de nombre réels z=(a,b).

1) Addition et multiplication de nombres complexes :

```python
def Addition(z1, z2):
    a1, b1 = z1 #Extraction des valeurs des couples
    a2, b2 = z2

    resultat = (a1+a2, b1+b2)
    return resultat

def Multiplication(z1, z2):
    a1, b1 = z1 #Extraction des valeurs des couples
    a2, b2 = z2

    resultat = (a1*a2-b1*b2, a1*b2+b1*a2)
    return resultat
```

2) Puissance nème d'un nombre complexe :

```python
def Puissance(z, n):
    a, b = z
    resultat = (1, 0)
    #On commence par l'élement neutre pour la mutliplication, 1

    for _ in range(n):
        resultat = (
            resultat[0] * a - resultat[1] * b,
            resultat[0] * b + resultat[1] * a)
    # On multiplie le resultat (initialement égal à 1, puis celui obtenu) par z

    return resultat
```

3) Module d'un nombre complexe :

```python
def Module(z):
    a, b = z
    module = (a **2 + b ** 2) ** 0.5
    return module
```

4) Evaluation d’un polynôme P en z :

```
def Eval(P, z):
    #P = [5,1,2] -> 2x^2 + 1x + 5
    longueur = len(P)
    resultat = (0, 0)
    #On commence par l'élement neutre pour l'addition, 0

    for i in range(longueur):
        MonomeSansCoeff = Puissance(z, i)
        a, b = MonomeSansCoeff
        Monome = (a*P[i] ,b *P[i])
        resultat = Addition(resultat, Monome)

    return resultat
```

5) Dérivée $n^{ème}$ du polynôme P :

```
def deriv(P, n):
    longueur = len(P)
    DériP = [0] * longueur
    if n == 0:
        return P
    if n >= len(P):
        return DériP
    else :
        for i in range(0 , len(P)-n):
            DériP[i] = (P[i+n]*math.factorial(i+n))/math.factorial(i)
        for i in range(len(P)-n+1 ,len(P)):
            DériP[i] = 0
        return DériP
```

Part 3 : Algorithme

Nous allons dans cette partie créer l'algorithme de Weyl en utilisant les opérateurs et fonctions créées dans la partie précédente.

1) Test de l'inégalité (1) pour un polynôme P, un complexe z et un réel r :

```
def Test(P, r, z):
    longueur = len(P)
    Somme = 0
    for i in range(1 ,longueur):
        Pdéri = deriv(P, i)
        PdériEvalué = Eval(Pdéri, z)
        RePdériEvalué , ImPdériEvalué = PdériEvalué
        Module = (RePdériEvalué**2 + ImPdériEvalué**2)**0.5
        x = Module*(r**i)/math.factorial(i)
        Somme = Somme + x
    Pz1 = Eval(P, z)
    RePz1, ImPz1 = Pz1
    NormePz1 = (RePz1**2 + ImPz1**2)**0.5
    if NormePz1 > Somme :
        #print("Pas de racine dans le disque de centre z et de rayon r")
        return 0
    else :
        #print("Il y a peut être une racine")
        return 1
```

2) Découpe le carré initial en quatre sous-carrés et test de l'inégalité (1) dans chacun de ces sous-carrés, il ne retient que les non-exclus :

```
def TestCarres(P,c,a):
    ReC, ImC = c
    ListeDesCarrés = []
    Point0 = (ReC - a/4 , ImC + a/4)
    Point1 = (ReC + a/4 , ImC + a/4)
    Point2 = (ReC - a/4 , ImC - a/4)
    Point3 = (ReC + a/4 , ImC - a/4)
    r = (2**0.5)*a/4

    if Test(P, r, Point0):
        ListeDesCarrés.append([Point0 ,a/2])

    if Test(P, r, Point1):
        ListeDesCarrés.append([Point1 ,a/2])

    if Test(P, r, Point2):
        ListeDesCarrés.append([Point2 ,a/2])

    if Test(P, r, Point3):
        ListeDesCarrés.append([Point3 ,a/2])

    return ListeDesCarrés
```

3) Algorithme de Weyl : n étapes de découpage et de test successives sur chaque carrés retenus à chaque itérations, pour renvoyer la liste des petits carrés restant à la fin.

```python
def Weyl(P,n):
    Amax = abs(max ((P[:-1]), key=abs))
    An = abs(P[len(P)-1])
    r = (1+Amax/An)
    coté = 2*r
    if Test(P, r , (0,0)) == 0:
        print("Pas de racines !")
        return 0
    else :
        L = TestCarres(P,(0,0) ,coté)
        M=[]
        for i in range(0, n-1):
            for j in range(len(L)):
                Carréj = L[j]
                CentreCarréj = Carréj[0]
                CotéCarréj = Carréj[1]
                K = TestCarres(P,CentreCarréj,CotéCarréj)
                for k in range(len(K)):
                    M.append(K[k])
            L = M
            M = []
        return L
```

4) Représente le carré de côté a et centre c (nombre complexe) :

```python
def Carre(c, a):
        ReC, ImC = c
        CoinsCarrés = np.array([
            [ReC - a/2, ImC - a/2],
            [ReC + a/2, ImC - a/2],
            [ReC + a/2, ImC + a/2],
            [ReC - a/2, ImC + a/2],
            [ReC - a/2, ImC - a/2] ])
        plt.plot(CoinsCarrés[:, 0], CoinsCarrés[:, 1], 'g-')
        #Le "g-" indique la couleur et
        #que l'on souhaite utiliser un trait rempli pour le contour
        plt.axis('equal')
        plt.title('Carré de Centre {} et Côté {}'.format(c, a))
        plt.xlabel('Axe Réel')
        plt.ylabel('Axe Imaginaire')
        plt.grid(True)
        plt.show()
```

5) Visualisation du programme de Weyl étapes par étapes et représente à la fin les carrés non-exclus.

```
def CarreWeyl(P, n):
        Amax = abs(max ((P[:-1]), key=abs))
        An = abs(P[len(P)-1])
        r = (1+Amax/An)
        coté = 2*r
        L =  Weyl(P,n)

        for i in range(len(L)):
            Carréi = L[i]
            Xcentre, Ycentre = Carréi[0]
            l = Carréi[1]/2
            CoinsCarrés = [
                (Xcentre - l, Ycentre - l),
                (Xcentre + l, Ycentre - l),
                (Xcentre + l, Ycentre + l),
                (Xcentre - l, Ycentre + l),
                (Xcentre - l, Ycentre - l)
                ]
            CooX = [point[0] for point in CoinsCarrés]
            CooY = [point[1] for point in CoinsCarrés]
            plt.fill(CooX, CooY, 'red' ,0.5)
        Carre((0,0), coté)

def FilmWeyl(P,n):
    for i in range(n):
        CarreWeyl(P, i)
        plt.pause(0.1)
```

CarreWeyl rend les carrés finaux dans le grand carré initial au bout de n itérations. Avec FilmWeyl on peut visualiser l'évolution graphique de l'algorithme.

6) Nous nous sommes donnés plusieurs polynômes de différents degrés, allant de 2 à 6. Nous comparons les résultats avec les racines données par Wolfram Alpha, avec un n choisis qui permet une précision d'au moins 10^{-2}.

- **$P(z) = 5 + 4z + 8z^2 + 9z^3 + z^4$:** (P = [5,4,8,9,1] à rentrer sur python)

Racines données par Wolfram alpha :

Real roots

$z \approx -8.0594$ z1

$z \approx -1.0826$ z2

Complex roots

$z \approx 0.07100 - 0.75367\, i$ z3

$z \approx 0.07100 + 0.75367\, i$ z4

On lance pour 10 itérations, Weyl(P,10) :

```
[[(-8.056640625, 0.009765625), 0.01953125], z1
 [(-1.083984375, 0.009765625), 0.01953125],z2
 [(0.068359375, 0.751953125), 0.01953125],z4
 [(-8.056640625, -0.009765625), 0.01953125], z1
 [(-1.083984375, -0.009765625), 0.01953125],z2
 [(0.068359375, -0.751953125), 0.01953125]]z3
```

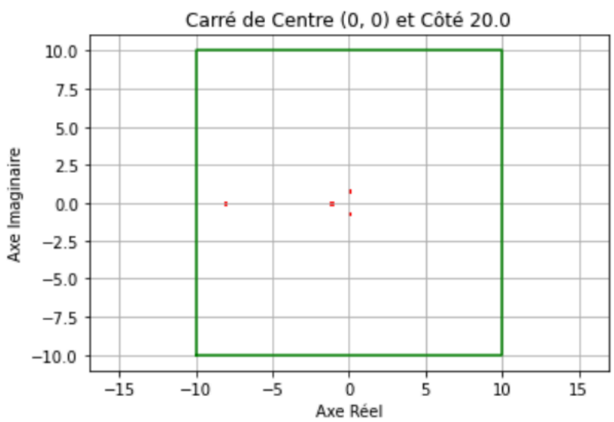
40 itérations, Weyl(P,40) :

```
[[(-8.05940123776054, 9.094947017729282e-12), 1.8189894035458565e-11],z1
 [(-1.0825949326772388, 9.094947017729282e-12), 1.8189894035458565e-11],z2
 [(0.07099808522070816, 0.7536715317746712), 1.8189894035458565e-11], z4
 [(0.07099808522070816, 0.7536715317564813), 1.8189894035458565e-11], z4
 [(-8.05940123776054, -9.094947017729282e-12), 1.8189894035458565e-11],z1
 [(-1.0825949326772388, -9.094947017729282e-12), 1.8189894035458565e-11],z2
 [(0.07099808522070816, -0.7536715317564813), 1.8189894035458565e-11],z3
 [(0.07099808522070816, -0.7536715317746712), 1.8189894035458565e-11]] z3
```

56 itérations, Weyl(P,56) :

```
[[(0.07099808521985565, 0.7536715317673762), 2.7755575615628914e-16], z4
 [(0.07099808521985565, -0.7536715317673762), 2.7755575615628914e-16]] z3
```

Et on obtient pour 8 itérations la représentation graphique suivante (FilmWeyl(P,8)):



Nous remarquons avec ce premier exemple que certaines racines sont sélectionnées plusieurs fois. De plus, la précision augmente quand on augmente le nombre d'étapes n. Par contre, quand ce dernier est trop important (ici n=56), on peut perdre certaines racines. Nous développerons ce point dans la partie 4.

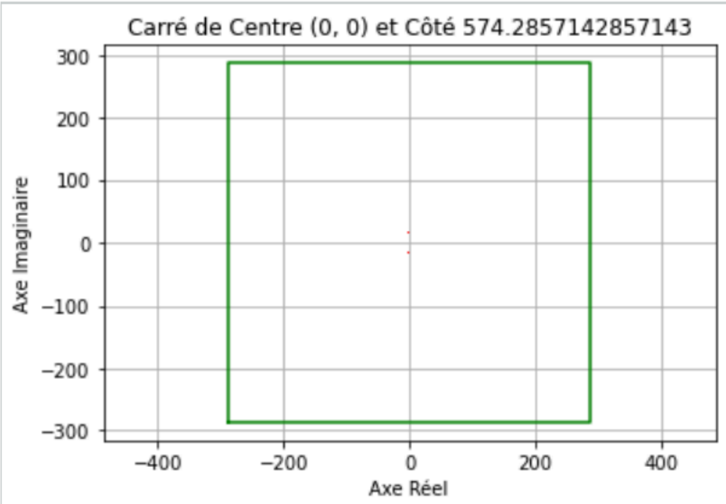
- **$P(z) = 2003 + 12z + 7z^2$:** (P = [2003, 12, 7])

Racines WolframAlpha : − 0.857142857142857142857142857142857142857142857142857142857142857142 85714285… −/+ 16.894027443605805935519948326546100487711004326582759091037453… i

Weyl(P,25) :

```
[[(-0.8571472764015198, 16.89403282744544), 1.7115047999790738e-05],
 [(-0.8571472764015198, -16.89403282744544), 1.7115047999790738e-05]]
```

FilmWeyl(P,9) :



Avec des gros coefficients numériques, comme par exemple ici 2023, la zone de recherche initiale est très grande, rendant parfois la visualisation un peu difficile. De la même manière la visualisation peut s'avérer être compliquée pour un nombre d'itération trop grand rendant les carrés finaux trop petits, pratiquement invisibles.

- **$P(z) = -13 -4z + 7z^2 + 5z^3 -2z^4 + 9z^5 -8z^6$:** (P = [-13, -4, 7, 5, -2, 9, -8)

Racines WolframAlpha

$z \approx -0.74406 - 0.43028\,i$ z1

$z \approx -0.74406 + 0.43028\,i$ z2

$z \approx 0.08859 - 1.19217\,i$ z3

$z \approx 0.08859 + 1.19217\,i$ z4

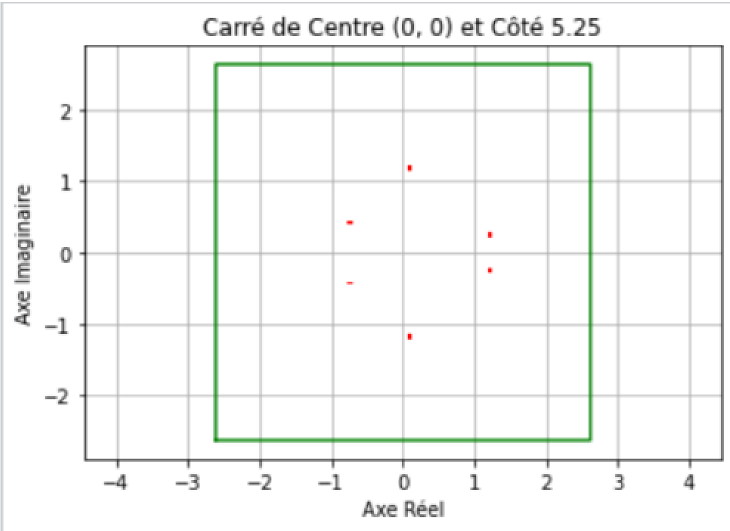
$z \approx 1.21797 - 0.23601\,i$ z5

$z \approx 1.21797 + 0.23601\,i$ z6

Weyl(P,50) :

```
[[(-0.7440610287153683, 0.43027904084104585), 4.6629367034256575e-15],  z2
 [(0.08859444718176601, 1.1921681480615702), 4.6629367034256575e-15],  z4
 [(0.08859444718177067, 1.1921681480615702), 4.6629367034256575e-15],  z4
 [(1.2179665815336027, 0.2360104616694384), 4.6629367034256575e-15], z6
 [(1.2179665815336027, 0.23601046166943374), 4.6629367034256575e-15],  z6
 [(-0.7440610287153683, -0.43027904084104585), 4.6629367034256575e-15], z1
 [(1.2179665815336027, -0.23601046166943374), 4.6629367034256575e-15],  z5
 [(1.2179665815336027, -0.2360104616694384), 4.6629367034256575e-15],  z5
 [(0.08859444718176601, -1.1921681480615702), 4.6629367034256575e-15],  z3
 [(0.08859444718177067, -1.1921681480615702), 4.6629367034256575e-15]] z3
```

FilmWeyl(P,8) :



- **<u>P(z) = 24 -15z + z³ :</u>** (P = [24, -15, 0, 1])

Racines WolframAlpha

Real root

$x \approx -4.5082$ z1

Complex roots

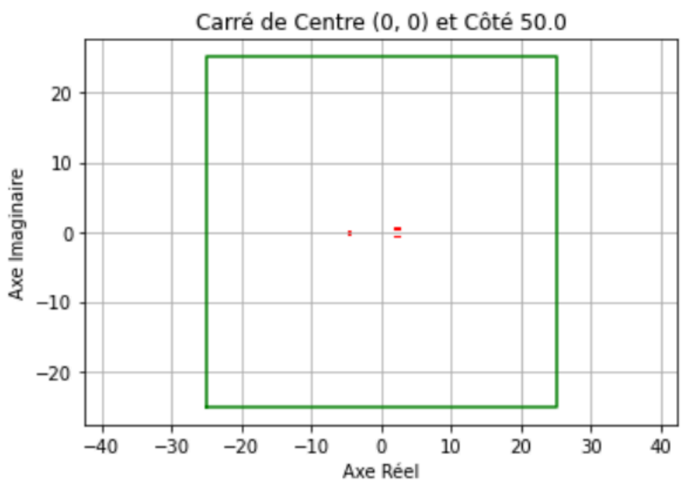
$x \approx 2.2541 - 0.49269\, i$ z2

$x \approx 2.2541 + 0.49269\, i$ z3

Weyl(P,20) :

```
[[(-4.508185386657715, 2.384185791015625e-05), 4.76837158203125e-05], z1
 [(2.254080772399023, 0.4926919937133789), 4.76837158203125e-05],  z3
 [(-4.508185386657715, -2.384185791015625e-05), 4.76837158203125e-05], z1
 [(2.254080772399023, -0.4926919937133789), 4.76837158203125e-05]] z2
```

FilmWeyl(P,8) :



Part 4 : Commentaires

1) La précision ε est donnée par la largeur des carrés. A chaque étape, ceux-ci sont divisés par 2. On en déduit que $\varepsilon = \frac{a}{2^n}$, avec a la taille du carré initial, déterminée dans la fonction Weyl.

On souhaite donc résoudre pour n : $\varepsilon \geq \frac{a}{2^n}$

On trouve $n \geq log_2(\frac{a}{\varepsilon})$

En moyenne, d'après de nombreux tests, la précision de 10-2 est obtenue après environ 10 itérations, ce qui est une vitesse de convergence assez pertinente, car les calculs sont très rapides à faire.

2) La méthode de Newton est une méthode efficace mais dépend grandement de l'initialisation choisie. Il faut l'adapter selon les signes des dérivées premières et secondes (sur R). La méthode de Newton peut alors réaliser une boucle infinie si les conditions initiales sont mal sélectionnées.

De plus, si on ne connaît pas la dérivée première (même si pour les polynômes elle se calculent aisément) mais que l'on dispose seulement d'une approximation, l'algorithme est moins pertinent.

Mais elle reste, si bien programmée, plus efficace que la méthode de Weyl, mais elle ne permet de localiser qu'une seule racine à la fois, contrairement à notre programme.

Son avantage réside dans le fait que les conditions initiales sont directement calculées à partir des coefficients du polynômes, générant un carré de base, qui se réduit rapidement.

Il est ainsi facile, avec peu d'itérations, d'obtenir de bonnes approximations des racines.

Cependant cette méthode n'est pas sans défauts : elle sélectionne plusieurs fois les mêmes racines, on peut donc s'assurer de la précision en regardant les chiffres après la virgule en commun entre les différentes solutions. Plus on augmente le nombre d'étapes, plus la précision augmente, mais plus on risque de perdre certaines racines, comme montré précédemment avec le polynôme P(z) = 5 + 4z + 8z2+ 9z3+z4.

Pour s'assurer d'avoir toutes les racines, on peut penser au corollaire du Théorème Fondamental de l'Algèbre, qui nous assure que tout polynômes à k coefficients admet k racines. Il faut alors pouvoir comptabiliser k racines distinctes pour un polynôme de degré k, et au augmente n pour augmenter la précision, jusqu'à ce que les racines commencent à disparaître. Cependant, l'existence des racines multiples met à mal cette méthode. Il convient cependant de rappeler que le nombre d'étapes nécessaires pour perdre des racines donne une précision très rarement nécessaire. Donc en pratique, le risque de louper des solutions est moindre.

Conclusion : La méthode de Weyl est une méthode efficace pour trouver les racines complexes d'un polynôme à coefficients réels. Elle est rapide (disparition rapide des carrés) et présente l'avantage d'être programmable avec les outils et imports /simples de Python. La visualisation du processus est facile, et doit tourner aux alentours des 8/9 itérations pour observer graphiquement les solutions. La précision est également excellente.