Résolution numérique des équations polynomiales avec les relations de Viète (relations entre coefficients et racines) et implémentation des calculs en Python

Benoît Pasquet

Résolution numérique des équations polynomiales avec les relations de Viète (relations entre coefficients et racines) et implémentation des calculs en Python

Par Benoît Pasquet

easymaths31@gmail.com

September 8, 2024

Résumé

Cet article présente une méthode de résolution numérique des équations polynomiales qui utilise les relations de Viète (relations entre coefficients et racines). Cette méthode permet de déterminer en même temps toutes les racines d'un polynôme de degré n qu'elles soient réelles ou complexes.

1 Théorème fondamental de l'algèbre (ou Théorème de Gauss-d'Alembert)

Tout polynôme à coefficient complexe de degré n ≥ 1 :

$$P(z) = a_n z^n + a_{n-1} z^{n-1} + ... + a_1 z + a_0 \tag{1}$$

possède exactement n racines complexes : $z_1, z_2, ..., z_n$, distinctes ou confondues, telles que :

$$P(z) = a_n (z - z_1)(z - z_2)...(z - z_n) \tag{2}$$

2 Relations entre coefficients et racines (ou relations de Viète)

2.1 Polynômes symétriques

On définie les polynômes symétriques $\sigma_1, \sigma_2, ..., \sigma_n$ à n indéterminées par :

$$\sigma_1(z_1, z_2, ..., z_n) = \sum_{i=1}^{n} z_i$$

$$\sigma_2(z_1, z_2, ..., z_n) = \sum_{1 \leq i \leq j \leq n} z_i z_j$$

$$\vdots$$

$$\sigma_k(z_1, z_2, ..., z_n) = \sum_{1 \leq i_1 < ... < i_k \leq n} z_{i1} z_{i2} ... z_{ik}$$

$$\vdots$$

$$\sigma_n(z_1, z_2, ..., z_n) = z_1 z_2 ... z_n$$

2.2 Théorème

Pour tout k $\in$ [1,n] :

$$\sigma_k(z_1, z_2, ..., z_n) = (-1)^k \frac{a_{n-k}}{a_n} \tag{3}$$

Ces relations sont appelées relations de Viète [1] et s'obtiennent en développant le produit (2) et en identifiant les coefficients du développement obtenu avec les coefficients de (1).

2.3 Exemples

- Cas n=2 :

$$\begin{aligned} a_2z^2 + a_1z + a_0 &= a_2(z - z_1)(z - z_2) \\ &= a_2(z^2 - z_2z - z_1z + z_1z_2) \\ &= a_2[z^2 - (z_2 + z_1)z + z_1z_2] \\ &= a_2z^2 - a_2(z_2 + z_1)z + a_2z_1z_2 \end{aligned}$$

Par identification des coefficients :

$$\left\{ \begin{array}{rcl} -a_2(z_1 + z_2) &=& a_1 \\ a_2(z_1z_2) &=& a_0 \end{array} \right.$$

On retrouve bien les relations de Viète :

$$\left\{ \begin{array}{rcl} z_1 + z_2 &=& -\frac{a_1}{a_2} \\ z_1z_2 &=& \frac{a_0}{a_2} \end{array} \right.$$

z_1 et z_2 sont des nombres complexes :

$$\begin{aligned} z_1 &= x_1 + iy_1 \\ z_2 &= x_2 + iy_2 \end{aligned}$$

En remplaçant dans les relations de Viète, on a :

$$\left\{ \begin{array}{rcl} (x_1 + iy_1) + (x_2 + iy_2) &=& -\frac{a_1}{a_2} \\ (x_1 + iy_1)(x_2 + iy_2) &=& \frac{a_0}{a_2} \end{array} \right.$$

$$\left\{ \begin{array}{rcl} (x_1 + x_2) + i(y_1 + y_2) &=& -\frac{a_1}{a_2} \\ x_1x_2 + ix_1y_2 + ix_2y_1 - y_1y_2 &=& \frac{a_0}{a_2} \end{array} \right.$$

$$\left\{ \begin{array}{rcl} (x_1 + x_2) + i(y_1 + y_2) &=& -\frac{a_1}{a_2} \\ (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1) &=& \frac{a_0}{a_2} \end{array} \right.$$

Les coefficients a_k sont des nombres complexes. Par identification des parties réelles et imaginaires, on obtient le système à quatre équations et quatre inconnues :

$$\left\{\begin{array}{rcl} x_1 + x_2 & = & \mathrm{Re}(-\frac{a_1}{a_2}) \\ y_1 + y_2 & = & \mathrm{Im}(-\frac{a_1}{a_2}) \\ x_1x_2 - y_1y_2 & = & \mathrm{Re}(\frac{a_0}{a_2}) \\ x_1y_2 + x_2y_1 & = & \mathrm{Im}(\frac{a_0}{a_2}) \end{array}\right. \tag{4}$$

- Cas n=3 :

$$\begin{aligned} a_3z^3+a_2z^2+a_1z+a_0 &= a_3(z-z_1)(z-z_2)(z-z_3) \\ &= a_3[z^2-(z_1+z_2)z+z_1z_2](z-z_3) \\ &= a_3[z^3-(z_1+z_2)z^2+z_1z_2z-z_3z^2+z_3(z_1+z_2)z-z_1z_2z_3] \\ &= a_3[z^3-(z_1+z_2+z_3)z^2+(z_1z_2+z_1z_3+z_2z_3)z-z_1z_2z_3] \\ &= a_3z^3-a_3(z_1+z_2+z_3)z^2+a_3(z_1z_2+z_1z_3+z_2z_3)z-a_3z_1z_2z_3] \end{aligned}$$

Par identification des coefficients :

$$\left\{\begin{array}{rcl} -a_3(z_1+z_2+z_3) &=& a_2 \\ a_3(z_1z_2+z_1z_3+z_2z_3) &=& a_1 \\ -a_3(z_1z_2z_3) &=& a_0 \end{array}\right.$$

On retrouve bien les relations de Viète :

$$\left\{\begin{array}{rcl} z_1+z_2+z_3 &=& -\frac{a_2}{a_3} \\ z_1z_2+z_1z_3+z_2z_3 &=& \frac{a_1}{a_3} \\ z_1z_2z_3 &=& -\frac{a_0}{a_3} \end{array}\right.$$

z_1 , z_2 et z_3 sont des nombres complexes :

$$\begin{aligned} z_1 &= x_1+iy_1 \\ z_2 &= x_2+iy_2 \\ z_3 &= x_3+iy_3 \end{aligned}$$

En remplaçant dans les relations de Viète, on a :

$$\left\{\begin{array}{rcll} (x_1+iy_1)+(x_2+iy_2)+(x_3+iy_3) &=& -\frac{a_2}{a_3} & (a) \\ (x_1+iy_1)(x_2+iy_2)+(x_1+iy_1)(x_3+iy_3)+(x_2+iy_2)(x_3+iy_3) &=& \frac{a_1}{a_3} & (b) \\ (x_1+iy_1)(x_2+iy_2)(x_3+iy_3) &=& -\frac{a_0}{a_3} & (c) \end{array}\right.$$

Calcul équation (a) :

$$(x_1 + x_2 + x_3) + i(y_1 + y_2 + y_3) \quad = \quad -\frac{a_2}{a_3}$$

Calcul équation (b) :

$$\begin{array}{rcl} (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1) & & \\ +(x_1x_3 - y_1y_3) + i(x_1y_3 + x_3y_1) & & \\ +(x_2x_2 - y_2y_3) + i(x_2y_3 + x_3y_2) & = & \frac{a_1}{a_3} \end{array}$$

$$\begin{array}{rcl} (x_1x_2 - y_1y_2) + (x_1x_3 - y_1y_3) + (x_2x_3 - y_2y_3) & & \\ +i[(x_1y_2 + x_2y_1) + (x_1y_3 + x_3y_1) + (x_2y_3 + x_3y_2)] & = & \frac{a_1}{a_3} \end{array}$$

$$\begin{array}{rcl} (x_1x_2 + x_1x_3 + x_2x_3 - y_1y_2 - y_1y_3 - y_2y_3) & & \\ +i(x_1y_2 + x_2y_1 + x_1y_3 + x_3y_1 + x_2y_3 + x_3y_2) & = & \frac{a_1}{a_3} \end{array}$$

Calcul équation (c) :

$$[(x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)](x_3 + iy_3) \quad = \quad -\frac{a_0}{a_3}$$

$$x_3(x_1x_2 - y_1y_2) + ix_3(x_1y_2 + x_2y_1) + iy_3(x_1x_2 - y_1y_2) - y_3(x_1y_2 + x_2y_1) \quad = \quad -\frac{a_0}{a_3}$$

$$(x_1x_2x_3 - x_3y_1y_2 - x_1y_2y_3 - x_2y_1y_3) + i(x_1x_3y_2 + x_2x_3y_1 + x_1x_2y_3 - y_1y_2y_3) \quad = \quad -\frac{a_0}{a_3}$$

Les coefficients a_k sont des nombres complexes. Par identification des parties réelles et imaginaires, on obtient le système à six équations et six inconnues :

$$\left\{ \begin{array}{rcl} x_1 + x_2 + x_3 & = & \mathrm{Re}(-\frac{a_2}{a_3}) \\ y_1 + y_2 + y_3 & = & \mathrm{Im}(-\frac{a_2}{a_3}) \\ x_1x_2 + x_1x_3 + x_2x_3 - y_1y_2 - y_1y_3 - y_2y_3 & = & \mathrm{Re}(\frac{a_1}{a_3}) \\ x_1y_2 + x_2y_1 + x_1y_3 + x_3y_1 + x_2y_2 + x_3y_2 & = & \mathrm{Im}(\frac{a_1}{a_3}) \\ x_1x_2x_3 - x_3y_1y_2 - x_1y_2y_3 - x_2y_1y_3 & = & \mathrm{Re}(-\frac{a_0}{a_3}) \\ x_1x_3y_2 + x_2x_3y_1 + x_1x_2y_3 - y_1y_2y_3 & = & \mathrm{Im}(-\frac{a_0}{a_3}) \end{array} \right. \qquad (5)$$

3 Résolution numérique d'un système non-linéaire

3.1 Formule de Taylor

Les systèmes (4) et (5) obtenus précédemment sont non-linéaires. Pour les résoudre, nous devons les transformer en systèmes linéaires. Pour cela nous allons utiliser la formule de Taylor [2] pour les fonctions à plusieurs variables, à l'ordre 1.

Par exemple, pour une fonction à deux variables :

$$F(x,y) \simeq F(x_0,y_0) + \frac{\partial F}{\partial x}(x_0,y_0)(x-x_0) + \frac{\partial F}{\partial y}(x_0,y_0)(y-y_0)$$

Ou x_0 et y_0 sont des valeurs approchées de x et y.

3.2 Principe de résolution

Considérons le système S à deux équations et deux inconnues:

$$\left\{ \begin{array}{rcl} F1(x,y) & = & l1 \\ F2(x,y) & = & l2 \end{array} \right.$$

D'aprés la formule de Taylor à l'ordre 1 :

$$\begin{array}{rcl} F1(x,y) & \simeq & F1(x_0,y_0) + \frac{\partial F1}{\partial x}(x_0,y_0)(x-x_0) + \frac{\partial F1}{\partial y}(x_0,y_0)(y-y_0) \\ F2(x,y) & \simeq & F2(x_0,y_0) + \frac{\partial F2}{\partial x}(x_0,y_0)(x-x_0) + \frac{\partial F2}{\partial y}(x_0,y_0)(y-y_0) \end{array}$$

On a donc :

$$\left\{ \begin{array}{rcl} F1(x_0,y_0) + \frac{\partial F1}{\partial x}(x_0,y_0)(x-x_0) + \frac{\partial F1}{\partial y}(x_0,y_0)(y-y_0) & = & l1 \\ F2(x_0,y_0) + \frac{\partial F2}{\partial x}(x_0,y_0)(x-x_0) + \frac{\partial F2}{\partial y}(x_0,y_0)(y-y_0) & = & l2 \end{array} \right.$$

Soit :

$$\left\{ \begin{array}{rcl} \frac{\partial F1}{\partial x}(x_0,y_0)(x-x_0) + \frac{\partial F1}{\partial y}(x_0,y_0)(y-y_0) & = & l1 - F1(x_0,y_0) \\ \frac{\partial F2}{\partial x}(x_0,y_0)(x-x_0) + \frac{\partial F2}{\partial y}(x_0,y_0)(y-y_0) & = & l2 - F2(x_0,y_0) \end{array} \right.$$

Ou sous forme matricielle : $J * X = K$, avec :

$$\text{J=} \begin{pmatrix} \frac{\partial F1}{\partial x}(x_0, y_0) & \frac{\partial F1}{\partial y}(x_0, y_0) \\ \frac{\partial F2}{\partial x}(x_0, y_0) & \frac{\partial F2}{\partial y}(x_0, y_0) \end{pmatrix}$$

J est la matrice Jacobienne du système S.

$$\text{X=} \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix}$$

$$\text{K=} \begin{pmatrix} l1 - F1(x_0, y_0) \\ l2 - F2(x_0, y_0) \end{pmatrix}$$

On a alors un système linéaire que l'on peut résoudre matriciellement :

$$X = J^{-1} * K$$

On peut alors calculer x_1 et y_1 :

$$\begin{aligned} x_1 &= x_0 + X[0] \\ y_1 &= y_0 + X[1] \end{aligned}$$

On réitére ensuite le processus pour calculer x_2 et y_2 avec :

$$\text{J=} \begin{pmatrix} \frac{\partial F1}{\partial x}(x_1, y_1) & \frac{\partial F1}{\partial y}(x_1, y_1) \\ \frac{\partial F2}{\partial x}(x_1, y_1)(& \frac{\partial F2}{\partial y}(x_1, y_1) \end{pmatrix}$$

$$\text{K=} \begin{pmatrix} l1 - F1(x_1, y_1) \\ l2 - F2(x_1, y_1) \end{pmatrix}$$

$$X = J^{-1} * K$$

$$\begin{aligned} x_2 &= x_1 + X[0] \\ y_2 &= y_1 + X[1] \end{aligned}$$

On peut ensuite réitérer le processus jusqu'à ce que les valeurs X[0] et X[1] soient inférieures à la précision voulue.

3.3 Exemple

Considérons le système S à deux équations et deux inconnues :

$$\begin{cases} x + y - 2y^2 &= -4 \\ x^2 + y^2 &= 8 \end{cases}$$

- Itération n°1 :

$$\begin{aligned} x_0 &= 1.5 \\ y_0 &= 1.5 \end{aligned}$$

$$\text{J}= \begin{pmatrix} 1 & 1-4y_0 \\ 2x_0 & 2y_0 \end{pmatrix} = \begin{pmatrix} 1 & -5 \\ 3 & 3 \end{pmatrix}$$

$$\text{K}= \begin{pmatrix} -4-(x_0+y_0-2y_0^2) \\ 8-(x_0^2+y_0^2) \end{pmatrix} = \begin{pmatrix} -2.5 \\ 3.5 \end{pmatrix}$$

$$X = J^{-1} * K = \begin{pmatrix} 0.555 \\ 0.611 \end{pmatrix}$$

$$\begin{aligned} x_1 &= 1.5 + 0.555 = 2.055 \\ y_1 &= 1.5 + 0.611 = 2.111 \end{aligned}$$

- Itération n°2 :

$$\text{J}= \begin{pmatrix} 1 & 1-4y_1 \\ 2x_1 & 2y_1 \end{pmatrix} = \begin{pmatrix} 1 & -7.444 \\ 4.11 & 4.222 \end{pmatrix}$$

$$\text{K}= \begin{pmatrix} -4-(x_1+y_1-2y_1^2) \\ 8-(x_1^2+y_1^2) \end{pmatrix} = \begin{pmatrix} 0.746642 \\ -0.679346 \end{pmatrix}$$

$$X = J^{-1} * K = \begin{pmatrix} -0.0547 \\ -0.1077 \end{pmatrix}$$

$$
\begin{aligned}
x_2 &= 2.055 - 0.0547 = 2.0003 \\
y_2 &= 2.111 - 0.1077 = 2.0033
\end{aligned}
$$

- On peut procéder encore à quelques itérations supplémentaires pour mieux approcher les racines x=2.0 et y=2.0.

4 Résolution numérique des équations polynomiales avec Python

Les calculs précédents sont fastidieux et répétitifs. Il est donc nécessaire de les programmer en langage informatique. Pour ce faire, nous allons utiliser le langage Python. Outre sa simplicité, ce langage propose le module "SymPy" qui permet de faire du calcul formel. Ce module permet notamment de développer des polynômes et de calculer des nombres complexes et des matrices Jacobienne.

Pour les valeurs approchées des racines, nous avons utilisé les valeurs préconisées dans la méthode de Durand-Kerner [4] :

$$z_{k0} \quad = \quad (0.4 + 0.9i)^k$$

Nous présentons les listings des programmes pour les cas n=2 et n=3.

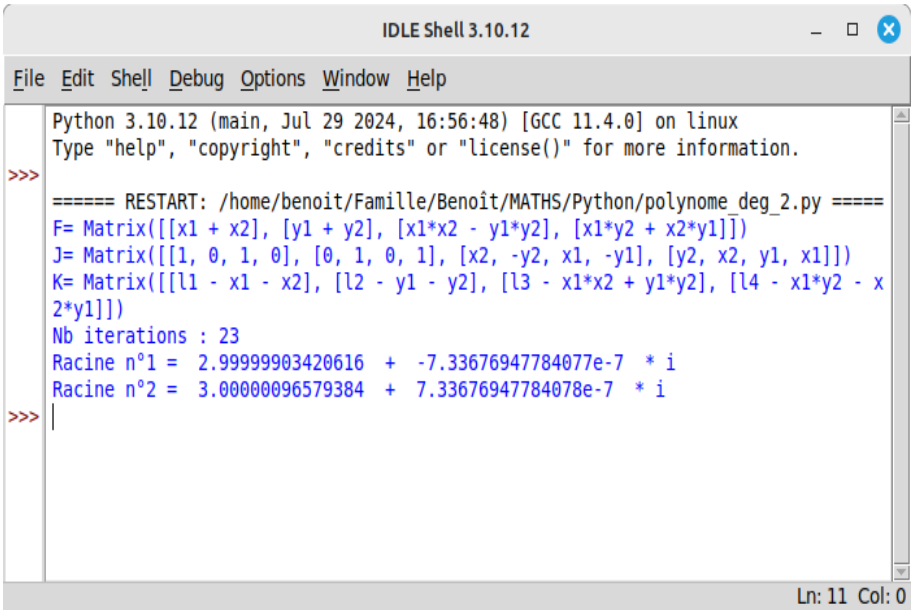
4.1 Polynôme de degré 2

```
# pour le calcul formel
import sympy as sp

# définition des variables utiles dans les calculs et de leur type
a2,a1,a0,x1,y1,x2,y2,l1,l2,l3,l4=sp.symbols("a2 a1 a0 x1 y1 x2 y2 l1 l2 l3 l4",real=True)
k,n,p,l=sp.symbols("k n p l",integer=True)

#### début declaration des fonctions utiles ####

# renvoie l'argument d'un complexe
def argumentz(z1):
    return sp.simplify(sp.arg(z1))

# renvoie le module d'un complexe
def modulez(z1):
    return sp.simplify(sp.abs(z1))

# renvoie la forme cartésienne d'une addition z1+z2
def addiz(z1,z2):
    add=z1+z2
    add=sp.simplify(add.sp.expand(complex=True))
    return add

#renvoie la forme cartésienne de l'addition d'un nombre quelconque de complexes
def addizn(*z):
    add=0
    for k in range(len(z)):
        add=add+z[k]
    add=sp.simplify(add.sp.expand(complex=True))
    return add

# renvoie la forme cartésienne d'un produit z1*z2
def multiz(z1,z2):
    mul=z1*z2
    mul=sp.simplify(mul.sp.expand(complex=True))
    return mul

# renvoie la forme cartésienne du produit d'un nombre quelconque de complexes
def multizn(*z):
```

```
    mul=1
    for k in range(len(z)):
        mul=mul*z[k]
    mul=sp.simplify(mul.sp.expand(complex=True))
    return mul

# renvoie la forme cartésienne d'un quotient z1/z2
def divz(z1,z2):
    if modulez(z2) !=0 :
        quo=z1*sp.conjugate(z2)
        quo=quo.sp.expand(complex=True)
        quo=quo/(modulez(z2)**2)
        quo=sp.simplify(quo.sp.expand(complex=True))
        return quo
    else :
        return "erreur: division  par zero"
def inversez(z1):
    return divz(1,z1)

#### fin déclaration des fonctions utiles ####

# racines du polynome
r1=x1+sp.I*y1
r2=x2+sp.I*y2

# calcul des sigmas (relations de Viète)
s1=r1+r2
s2=r1*r2

# calcul des fonctions
f1=sp.re(s1)
f2=sp.im(s1)
f3=sp.re(s2)
f4=sp.im(s2)

# calcul de la matrice F
F = sp.Matrix([f1, f2, f3, f4])
F.nullspace()
print('F=',F)

# calcul de la matrice J
J=F.jacobian([x1,y1,x2,y2])
J.nullspace()
print('J=',J)

```

```
# calcul de la matrice K
K = sp.Matrix([l1-f1, l2-f2, l3-f3, l4-f4])
K.nullspace()
print('K=',K)

# coefficients du polynome
val_a2=1.0+0.0*sp.I
val_a1=-6.0+0.0*sp.I
val_a0=9.0+0.0*sp.I

# calcul des lx
val_l1=sp.re(-val_a1/val_a2)
val_l2=sp.im(-val_a1/val_a2)
val_l3=sp.re(val_a0/val_a2)
val_l4=sp.im(val_a0/val_a2)

# valeurs initiales des racines
val_x1=1.0
val_y1=0.0
val_x2=0.4
val_y2=0.9

# tolérance et nombre d'itérations max
tolerance=0.000001
nb_iterations=100

for i in range(1, nb_iterations):

    #print('Iteration n° :',i)

    # on substitue les valeurs dans la matrice J
    Jsubs=J.subs({x1: val_x1, y1: val_y1, x2: val_x2, y2: val
   _y2})
    #print(Js)

    # on substitue les valeurs dans la matrice K
    Ksubs=K.subs({x1: val_x1, y1: val_y1, x2: val_x2, y2: val
   _y2, l1: val_l1, l2: val_l2, l3: val_l3, l4: val_l4})
    #print(Ks)

    # calcul de la matrice X
    X= Jsubs.inv()*(Ksubs)
    #print(X)

    val_x1=val_x1+X[0]
```

```
128     #print('x1=',val_x1)
129     val_y1=val_y1+X[1]
130     #print('y1=',val_y1)
131     val_x2=val_x2+X[2]
132     #print('x2=',val_x2)
133     val_y2=val_y2+X[3]
134     #print('y2=',val_y2)
135 
136     # affichage des résultats
137     if abs(X[0])<tolerance and abs(X[1])<tolerance and abs(X
        [2])<tolerance and abs(X[3])<tolerance :
138         print('Nb iterations :',i)
139         print('Racine n°1 = ' , val_x1 , ' + ' , val_y1 , ' *
        i')
140         print('Racine n°2 = ' , val_x2 , ' + ' , val_y2 , ' *
        i')
141         break
142 
143 if i >= (nb_iterations-1) :
144     print('Nombre d itérations dépassé.')
```



La matrice Jacobienne est : $\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ x_2 & -y_2 & x_1 & -y_1 \\ y_2 & x_2 & y1 & x_1 \end{pmatrix}$

Le polynôme $P(z) = z^2 - 6z + 9$ a une racine double : $z_1 = z_2 = 3$.

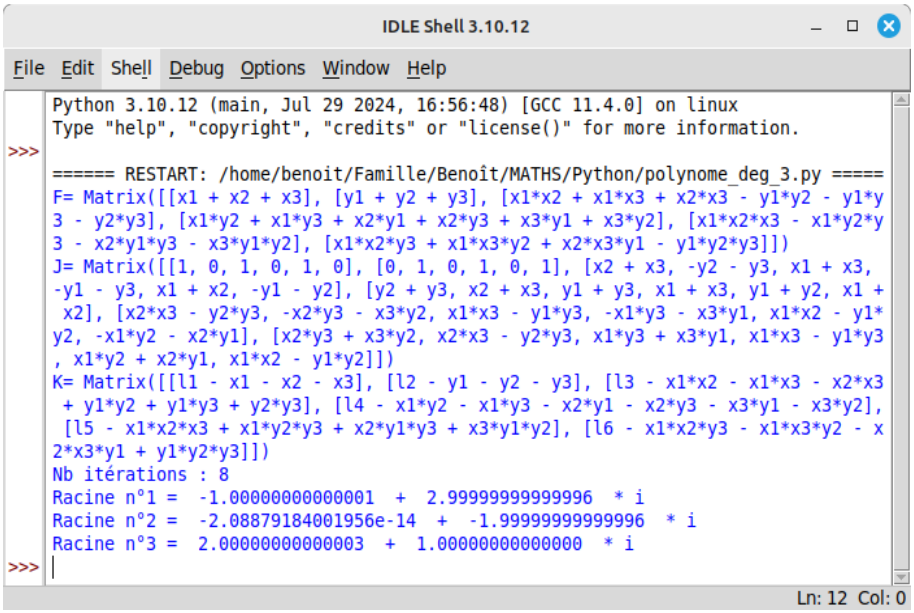
4.2 Polynôme de degré 3

```
# pour le calcul formel
import sympy as sp

# définition des variables utiles dans les calculs et de leur type
a2,a1,a0,x1,y1,x2,y2,x3,y3,l1,l2,l3,l4,l5,l6=sp.symbols("a2 a1 a0 x1 y1 x2 y2 x3 y3 l1 l2 l3 l4 l5 l6",real=True)
k,n,p,l=sp.symbols("k n p l",integer=True)

#### début declaration des fonctions utiles ####

# renvoie l'argument d'un complexe
def argumentz(z1):
    return sp.simplify(sp.arg(z1))

# renvoie le module d'un complexe
def modulez(z1):
    return sp.simplify(sp.abs(z1))

# renvoie la forme cartésienne d'une addition z1+z2
def addiz(z1,z2):
    add=z1+z2
    add=sp.simplify(add.sp.expand(complex=True))
    return add

# renvoie la forme cartésienne de l'addition d'un nombre quelconque de complexes
def addizn(*z):
    add=0
    for k in range(len(z)):
        add=add+z[k]
    add=sp.simplify(add.sp.expand(complex=True))
    return add

# renvoie la forme cartésienne d'un produit z1*z2
def multiz(z1,z2):
    mul=z1*z2
    mul=sp.simplify(mul.sp.expand(complex=True))
    return mul

# renvoie la forme cartésienne du produit d'un nombre quelconque de complexes
def multizn(*z):
```

```
    mul=1
    for k in range(len(z)):
        mul=mul*z[k]
    mul=sp.simplify(mul.sp.expand(complex=True))
    return mul

# renvoie la forme cartésienne d'un quotient z1/z2
def divz(z1,z2):
    if modulez(z2) !=0 :
        quo=z1*sp.conjugate(z2)
        quo=quo.sp.expand(complex=True)
        quo=quo/(modulez(z2)**2)
        quo=sp.simplify(quo.sp.expand(complex=True))
        return quo
    else :
        return "erreur: division  par zéro"
def inversez(z1):
    return divz(1,z1)

# racines du polynome
r1=x1+sp.I*y1
r2=x2+sp.I*y2
r3=x3+sp.I*y3

# calcul des sigmas (relations de Viète)
s1=r1+r2+r3
s2=r1*r2+r1*r3+r2*r3
s3=r1*r2*r3

# calcul des fonctions
f1=sp.re(s1)
f2=sp.im(s1)
f3=sp.re(s2)
f4=sp.im(s2)
f5=sp.re(s3)
f6=sp.im(s3)

# calcul de la matrice F
F = sp.Matrix([f1, f2, f3, f4, f5, f6])
F.nullspace()
print('F=',F)

# calcul de la matrice J
J=F.jacobian([x1,y1,x2,y2,x3,y3])
J.nullspace()
```

```
85 print('J=',J)
86
87 # calcul de la matrice K
88 K = sp.Matrix([l1-f1, l2-f2, l3-f3, l4-f4, l5-f5, l6-f6])
89 K.nullspace()
90 print('K=',K)
91
92 # coefficients du polynome
93 val_a3=1.0+0.0*sp.I
94 val_a2=-1.0-2.0*sp.I
95 val_a1=3.0+3.0*sp.I
96 val_a0=-10.0-10.0*sp.I
97
98 # calcul des lx
99 val_l1=sp.re(-val_a2/val_a3)
100 val_l2=sp.im(-val_a2/val_a3)
101 val_l3=sp.re(val_a1/val_a3)
102 val_l4=sp.im(val_a1/val_a3)
103 val_l5=sp.re(-val_a0/val_a3)
104 val_l6=sp.im(-val_a0/val_a3)
105
106 # valeurs initiales des racines
107 val_x1=1.0
108 val_y1=0.0
109 val_x2=0.4
110 val_y2=0.9
111 val_x3=-0.65
112 val_y3=0.72
113
114 # tolérance et nombre d'itérations max
115 tolerance=0.000001
116 nb_iterations=100
117
118 for i in range(1, nb_iterations):
119
120     #print('Itération n° :',i)
121
122     # on substitue les valeurs dans la matrice J
123     Jsubs=J.subs({x1: val_x1, y1: val_y1, x2: val_x2, y2: val
    _y2, x3: val_x3, y3: val_y3})
124     #print(Js)
125
126     # on substitue les valeurs dans la matrice K
127     Ksubs=K.subs({x1: val_x1, y1: val_y1, x2: val_x2, y2: val
    _y2, x3: val_x3, y3: val_y3, l1: val_l1, l2: val_l2, l3:
```

```
    val_l3, l4: val_l4, l5: val_l5, l6: val_l6})
    #print(Ks)

    # calcul de la matrice X
    X= Jsubs.inv()*(Ksubs)
    #print(X)

    val_x1=val_x1+X[0]
    #print('x1=',val_x1)
    val_y1=val_y1+X[1]
    #print('y1=',val_y1)
    val_x2=val_x2+X[2]
    #print('x2=',val_x2)
    val_y2=val_y2+X[3]
    #print('y2=',val_y2)
    val_x3=val_x3+X[4]
    #print('x3=',val_x3)
    val_y3=val_y3+X[5]
    #print('y3=',val_y3)

    # affichage des résultats
    if abs(X[0])<tolerance and abs(X[1])<tolerance and abs(X
    [2])<tolerance and abs(X[3])<tolerance and abs(X[4])<
    tolerance and abs(X[5])<tolerance:
        print('Nb itérations :',i)
        print('Racine n°1 = ' , val_x1 , ' + ' , val_y1 , ' *
     i')
        print('Racine n°2 = ' , val_x2 , ' + ' , val_y2 , ' *
     i')
        print('Racine n°3 = ' , val_x3 , ' + ' , val_y3 , ' *
     i')
        break

if i >= (nb_iterations-1) :
    print('Nombre d itérations dépassé.')
```



La matrice Jacobienne est :

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ x_2+x_3 & -y_2-y_3 & x_1+x_3 & -y_1-y_3 & x_1+x_2 & -y_1-y_2 \\ y_2+y_3 & x_2+x_3 & y_1+y_3 & x_1+x_3 & y_1+y_2 & x_1+x_2 \\ x_2x_3-y_2y_3 & -x_3y_2-x_2y_3 & x_1x_3-y_1y_3 & -x_3y_1-x_1y_3 & x_1x_2-y_1y_2 & -x_1y_2-x_2y_1 \\ x_3y_2+x_2y_3 & x_2x_3-y_2y_3 & x_3y_1+x_1y_3 & x_1x_3-y_1y_3 & x_3y_1+x_1y_3 & x_1x_2-y_1y_2 \end{pmatrix}$$

Le polynôme $P(z) = z^3 - (1+2i)z^2 + 3(1+i)z - 10(1+i)$ a 3 racines distinctes : $z_1 = -1+3i, z_2 = -2i, z_3 = -1+3i$.

4.3 Polynôme de degré supérieur à 3

Pour résoudre les équations polynomiales de degré ≥ 4, il suffit d'adapter le programme Python précédent en ajoutant les variables nécessaires.

On remarque toutefois que le temps de calcul de la matrice Jacobienne devient de plus en plus important. Mais ce calcul n'a besoin de se faire à chaque exécution du programme. La matrice Jacobienne peut être calculée dans un programme séparé et être ensuite appliquée dans le programme final.

5 Bibliographie

References

[1] Relations entre coefficients et racines. https://fr.wikipedia.org/wiki/Relations_entre_coefficients_et_racines.

[2] Formule de Taylor. https://math.stackexchange.com/questions/221669/derivation-of-multivariable-taylor-series.

[3] Calcul formel avec SymPy. http://formav.eu/2015PYTHON/FORMAV_Complexes_calcul_formel.html.

[4] Méthode de Durand-Kerner. https://en.wikipedia.org/wiki/Durand%E2%80%93Kerner_method.