Racines d'un polynôme

1. Introduction

Cette page montre comment obtenir numériquement les racines d'un polynôme de degré n, défini par :

$$P(x) = \sum_{k=0}^{n} a_k x^k = a_0 + a_1 x + \dots a_n x^n \tag{1}$$

Les coefficients peuvent être complexes et, dans tous les cas, on recherche les racines complexes. Un polynôme de degré n possède n racines complexes mais il peut y avoir des racines multiples. Par exemple, le polynôme $P(x) = (x-1)^3$ admet trois racines égales.
On peut citer comme application la recherche des pôles d'une fonction de transfert d'un système linéaire et la réduction en éléments simples de cette fonction de transfert, ou plus généralement la réduction en éléments simples d'une fraction rationnelle.

2. Méthodes numériques

2.a. Évaluation du polynôme et de ses dérivées

Il s'agit de calculer $P(x)$ pour une valeur de x donnée (complexe en général) le plus efficacement possible et en minimisant les erreurs d'arrondis. L'évaluation repose sur la factorisation suivante [1] :

$$\begin{aligned} P(x) &= a_0 + x(a_1 + a_2 x + \dots a_n x^{n-1}) \\ &= a_0 + x(a_1 + x(a_2 + a_3 x + \dots a n_n x^{n-2})) \\ &= \dots \\ &= a_0 + x(a_1 + x(a_2 + x(a_3 + x(\dots)))) \end{aligned}$$

qui conduit à l'algorithme récursif suivant :

$$\begin{aligned} p &\leftarrow a_n \\ p &\leftarrow a_{n-1} + px \\ p &\leftarrow a_{n-2} + px \\ &\dots \\ p &\leftarrow a_0 + px \end{aligned}$$

```
import numpy as np
```

La fonction suivante effectue l'évaluation d'un polynôme dont les coefficients sont donnés dans une liste ou un tableau `numpy.ndarray` contenant des nombres complexes :

```
def evalPol(a,x):
    k=len(a)-1
    p = a[k] #a_n
    k -= 1
    while k>=0:
        p = a[k]+p*x
        k -= 1
    return p
```

Voici un exemple avec le polynome $P(x) = 1 + x + x^2$:

```
a=[1+1j*0,1+1j*0,1+1j*0]
p = evalPol(a,1j)

print(p)
--> 1j
```

Considérons la dérivée du polynôme, définie par :

$$P'(x) = a_1 + 2a_2x + 3a_3x^2 + \dots na_nx^{n-1} \tag{2}$$

L'évalution de cette dérivée se fait de manière similaire à l'évaluation du polynôme, en multipliant chaque coefficient a_n par n :

```
def evalDerivPol(a,x):
    k=len(a)-1
    p = k*a[k] #(n-1)na_n
    k -= 1
    while k>=1:
        p = k*a[k]+p*x
        k -= 1
    return p
```

```
a=[1+1j*0,1+1j*0,1+1j*0]
p = evalDerivPol(a,1j)

print(p)
--> (1+2j)
```

Considérons la dérivée seconde du polynôme :

$$P''(x) = 2a_2 + 2(3)a_3x + \dots (n-1)na_nx^{n-2} \tag{3}$$

```
def evalDeriv2Pol(a,x):
    k=len(a)-1
    p = (k-1)*k*a[k] #na_n
    k -= 1
    while k>=2:
        p = (k-1)*k*a[k]+p*x
        k -= 1
    return p



a=[1+1j*0,1+1j*0,1+1j*0]
p = evalDeriv2Pol(a,1j)


print(p)
--> (2+0j)


a=[0+0j,0+0j,0+0j,1+0j]
p = evalDeriv2Pol(a,1)


print(p)
--> (6+0j)
```

2.b. Méthode de Laguerre

La méthode de Laguerre [1] permet d'obtenir la valeur approchée de la racine d'un polynôme la plus proche d'une valeur initiale donnée. Elle repose sur l'utilisation des dérivées première et seconde du polynôme. Notons $x_0, x_1, \ldots x_{n-1}$ les n racines du polynôme puis sa factorisation :

$$P(x) = (x - x_0)(x - x_1)\ldots(x - x_{n-1}) \tag{4}$$

La dérivée du polynôme (définie plus haut) peut se calculer à partir de cette expression (comme si x et les racines étaient réelles). On obtient aisément :

$$\frac{P'(x)}{P(x)} = \frac{1}{x - x_0} + \frac{1}{x - x_1} + \ldots + \frac{1}{x - x_{n-1}} \tag{5}$$

La dérivation de ce rapport conduit à :

$$\left(\frac{P'(x)}{P(x)}\right)^2 - \frac{P''(x)}{P(x)} = \frac{1}{(x - x_0)^2} + \frac{1}{(x - x_1)^2} + \ldots + \frac{1}{(x - x_{n-1})^2} \tag{6}$$

Soit x une valeur (complexe) supposée proche de la racine x_0, constituant une première estimation de cette racine. Si elle en est suffisamment proche, on peut considérer que la différence entre x est toutes les autres racines est pratiquement la même :

$$x - x_0 = d \tag{7}$$

$$x - x_k = e,\ k = 1, 2, \ldots n - 1 \tag{8}$$

D'après la relation (5), on a alors :

$$G(x) = \frac{P'(x)}{P(x)} = \frac{1}{d} + \frac{n-1}{e} \tag{9}$$

et d'après la relation (6) :

$$H(x) = \left(\frac{P'(x)}{P(x)}\right)^2 - \frac{P''(x)}{P(x)} = \frac{1}{d^2} + \frac{n-1}{e^2} \tag{10}$$

L'élimination de e de ces deux dernières équations conduit à une équation de degré 2 pour d, dont les solutions sont :

$$d = \frac{n}{G(x) \pm \sqrt{(n-1)(nH(x) - G(x)^2)}} \tag{11}$$

La solution dont le module est le plus petit est retenue. La nouvelle estimation est $x - d$. On répète itérativement ces opérations jusqu'à obtenir une valeur de d dont le module est inférieur à une certaine tolérance. Les deux fonctions suivantes implémentent cet algorithme :

```
def iteration(a,x):
    n = len(a)-1
    p = evalPol(a,x)
    if p==0: return 0
    pp = evalDerivPol(a,x)
    ppp = evalDeriv2Pol(a,x)
    G = pp/p
    H = G*G-ppp/p
    sq = np.sqrt((n-1)*(n*H-G*G))
    z1 = G+sq
    z2 = G-sq
    if np.absolute(z2) > np.absolute(z1):
        z1 = z2
    d = n/z1
    return d

def laguerre(a,x,tol,maxiter=100):
    d = iteration(a,x)
    if d==0:
        return x
    x = x-d
    iter = 1
    while np.absolute(d)>tol:
        d = iteration(a,x)
        if d==0:
            return x
        x = x-d
        iter += 1
        if iter > maxiter:
            raise Exception("Non convergence")
    return x
```

Afin de tester cette fonction, nous écrivons une fonction qui renvoie les coefficients d'un polynôme de degré 3 dont les trois racines sont données :

```
def polynome(x0,x1,x2):
    return np.array([-x0*x1*x2,x0*x2+x0*x1+x1*x2,-(x0+x1+x2),1],dtype=np.complex128)
```

Voici un test. La valeur initiale de x est nulle.

```
x0 = 1
x1 = 2+3*1j
x2 = 2-3*1j
a = polynome(x0,x1,x2)
try:
    x = laguerre(a,0,1e-4)
except:
    print("Non convergence")
    x = 0


print(x)
--> np.complex128(1.0000000000000036+0j)
```

On obtient la racine la plus proche de la valeur initiale (nulle).

2.c. Factorisation

Une fois la racine x_0 obtenue (la plus proche de 0 si la valeur d'essai initiale est nulle), on factorise le polynome :

$$P(x) = (x - x_0)(b_0 + b_1 x + b_2 x^2 + \dots b_{n-1} x^{n-1}) = (x - x_0)Q(x) \tag{12}$$

puis on cherche la racine la plus proche de zéro de $Q(x)$. On procède ainsi itérativement jusqu'à l'obtention des n racines.
Le développement de la forme factorisée permet d'obtenir :

$$\begin{aligned}
a_n &= b_{n-1} \\
a_{n-1} &= b_{n-2} - x_0 b_{n-1} \\
a_{n-2} &= b_{n-3} - x_0 b_{n-2} \\
&\dots \\
a_2 &= b_1 - x_0 b_2 \\
a_1 &= b_0 - x_0 b_1 \\
a_0 &= -x_0 b_0
\end{aligned}$$

On peut donc calculer les coefficients du polynôme Q de la manière suivante :

$$\begin{aligned}
b_{n-1} &= a_n \\
b_{n-2} &= a_{n-1} + x_0 b_{n-1} \\
b_{n-3} &= a_{n-2} + x_0 b_{n-2} \\
&\dots \\
b_1 &= a_2 + x_0 b_2 \\
b_0 &= a_1 + x_0 b_1
\end{aligned}$$

La fonction suivante effectue la factorisation :

```
def factorisation(a,x0):
    n = len(a)-1
    b = np.zeros(n,dtype=np.complex128)
    k = n-1
    b[k] = a[k+1]
    k -= 1
    while k>=0:
        b[k] = a[k+1]+x0*b[k+1]
        k -= 1
    return b
```

La fonction suivante détermine les n racines d'un polynôme :

```
def racines(a,tol=1e-4):
    n = len(a)-1
    r = []
    for k in range(n-1):
        x = laguerre(a,0,tol)
        r.append(x)
        a = factorisation(a,x)
    r.append(-a[0]/a[1]) # racine d'une polynome de degré 1
    return r
```

Voici un test :

```
x0 = 1
x1 = 2+3*1j
x2 = 2-3*1j
a = polynome(x0,x1,x2)
r = racines(a)


print(r[0])
--> np.complex128(1.0000000000000036+0j)

print(r[1])
--> np.complex128(1.9999999999999982+2.999999999999999j)

print(r[2])
--> np.complex128(1.9999999999999982-2.999999999999999j)
```

En raison des erreurs d'arrondis qui apparaissent lors de la factorisation, [1] recommande de considérer ces premières évaluations des racines et d'appliquer la méthode de Laguerre pour chacune d'elle sur le polynôme de départ afin d'affiner la précision :

```
def racines2(a,tol=1e-4):
    r = racines(a,tol)
    rac = []
    for x in r:
        x = laguerre(a,x,tol)
        rac.append(x)
    return rac
```

Voici un test :

```
x0 = 1
x1 = 2+3*1j
x2 = 2-3*1j
a = polynome(x0,x1,x2)
r = racines2(a)


print(r[0])
--> np.complex128(1+0j)

print(r[1])
--> np.complex128(1.9999999999999998+3j)

print(r[2])
--> np.complex128(1.9999999999999998-3j)
```

voici un autre exemple :

```
x0 = -3
x1 = 1+1j
x2 = 1-1j
a = polynome(x0,x1,x2)
r = racines2(a,tol=1e-6)


print(r[0])
--> np.complex128(1+1j)

print(r[1])
--> np.complex128(1-1j)

print(r[2])
--> np.complex128(-3+0j)
```

Voici un exemple avec des racines multiples :

```
x0 = 1
x1 = 1
x2 = 1
a = polynome(x0,x1,x2)
r = racines2(a,tol=1e-6)


print(r)
--> [np.complex128(1+0j), np.complex128(1+0j), np.complex128(1-0j)]
```

Références

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical recipes, the art of scientific computing*, (Cambridge University Press, 2007)