TD Informatique TSI2

Lagrange polynomials

Solution

PRELIMINARY

Comment faire des opérations sur les polynômes avec Python

Exemple

Soit le polynôme $P = X^3+2X-3$, *écrire les commandes pour calculer successivement* $P(0)$, $[P(1), P(2), P(3)]$, deg P, *les racines de* P, *puis les polynômes* P', P'' *puis le quotient et le reste de la division euclidienne de* P *par* $B = X^2 + 1$ *et enfin* P^n *pour* n *variant de* 0 *à* 8.

```
In [1]: from numpy.polynomial import Polynomial
In [2]: import numpy as np

In [3]: P = Polynomial([-3.,2.,0.,1.])

In [4]: P(0)
Out[4]: -3.0
In [5]: [P(1),P(2),P(3)]
Out[5]: [0.0, 9.0, 30.0]
In [6]: P(np.array([1,2,3]))
Out[6]: array([ 0.,  9., 30.])

In [7]: P.degree()
Out[7]: 3

In [8]: P.roots()
Out[8]: array([-0.5-1.6583124j, -0.5+1.6583124j,  1. +0.j ])
In [9]: P.deriv()
Out[9]:
Polynomial([2., 0., 3.],domain=[-1.,  1.],window=[-1.,  1.])
In [10]: P.deriv(2)
Out[10]:
Polynomial([0., 6.], domain=[-1.,  1.], window=[-1.,  1.])
In [11]: B=Polynomial([1.,0.,1.])
In [12]: P//B
Out[12]:
Polynomial([0., 1.], domain=[-1.,  1.], window=[-1.,  1.])
In [13]: P%B
Out[13]: # J'OBTIENT UN TRUC MOCHE
In [14]:B = Polynomial([1, 0, 1 ]; Q = P // B ; R = P % B
In [15]: Q.coef, R.coef
Out[15]: array([0., 1.]) , array([-3., 1.])
# C'EST MIEUX !

In [16]: P*B
Out[16]:
Polynomial([-3.,2.,-3.,3., 0.,1.],domain=[-1.,  1.],window=[-1.,1.])
```

Au passage, on a bien $BQ + R = (X^2 + 1) * X - 3 + X = X^3 + 2X - 3 = P$

```
In [17]: for n in range(0,9):
    ...:     P**n
    ...:
# IL NE SE PASSE RIEN !
# ON VA CHERCHER PRINT

In [18]: for n in range(0,9):
    ...:     print(P**n)
    ...:

poly([1.])

poly([-3.  2.  0.  1.])

poly([  9. -12.   4.  -6.   4.   0.   1.])

poly([-27.  54. -36.  35. -36.  12.  -9.   6.   0.   1.])

poly([ 81. -216. 216. -204. 232. -144. 86. -72. 24. -12. 8. 0. 1.])

poly([-2.430e+02  8.100e+02 -1.080e+03  1.125e+03 -1.320e+03  1.112e+03
 -7.500e+02  6.200e+02 -3.600e+02  1.700e+02 -1.200e+02  4.000e+01
 -1.500e+01  1.000e+01  0.000e+00  1.000e+00])

poly([ 7.290e+02 -2.916e+03  4.860e+03 -5.778e+03  7.020e+03 -7.056e+03
  5.599e+03 -4.680e+03  3.432e+03 -1.980e+03  1.320e+03 -7.200e+02
  2.950e+02 -1.800e+02  6.000e+01 -1.800e+01  1.200e+01  0.000e+00
  1.000e+00])

poly([-2.1870e+03 1.0206e+04 -2.0412e+04 2.7783e+04 -3.5532e+04 4.0068e+04
 -3.6687e+04  3.2258e+04 -2.6712e+04  1.8403e+04 -1.2600e+04  8.2320e+03
 -4.3050e+03  2.4500e+03 -1.2600e+03  4.6900e+02 -2.5200e+02  8.4000e+01
 -2.1000e+01  1.4000e+01  0.0000e+00  1.0000e+00])

poly([ 6.56100e+03 -3.49920e+04  8.16480e+04 -1.26360e+05  1.72368e+05
 -2.11680e+05  2.17980e+05 -2.05680e+05  1.84720e+05 -1.45320e+05
  1.06864e+05 -7.66080e+04  4.77820e+04 -2.85600e+04  1.69120e+04
 -8.23200e+03  4.14400e+03 -2.01600e+03  7.00000e+02 -3.36000e+02
  1.12000e+02 -2.40000e+01  1.60000e+01  0.00000e+00  1.00000e+00])

poly([ 7.290e+02 -2.916e+03  4.860e+03 -5.778e+03  7.020e+03 -7.056e+03
  5.599e+03 -4.680e+03  3.432e+03 -1.980e+03  1.320e+03 -7.200e+02
  2.950e+02 -1.800e+02  6.000e+01 -1.800e+01  1.200e+01  0.000e+00
  1.000e+00])

poly([-2.1870e+03 1.0206e+04 -2.0412e+04 2.7783e+04 -3.5532e+04 4.0068e+04
 -3.6687e+04  3.2258e+04 -2.6712e+04  1.8403e+04 -1.2600e+04  8.2320e+03
 -4.3050e+03  2.4500e+03 -1.2600e+03  4.6900e+02 -2.5200e+02  8.4000e+01
 -2.1000e+01  1.4000e+01  0.0000e+00  1.0000e+00])

poly([ 6.56100e+03 -3.49920e+04  8.16480e+04 -1.26360e+05  1.72368e+05
 -2.11680e+05  2.17980e+05 -2.05680e+05  1.84720e+05 -1.45320e+05
  1.06864e+05 -7.66080e+04  4.77820e+04 -2.85600e+04  1.69120e+04
 -8.23200e+03  4.14400e+03 -2.01600e+03  7.00000e+02 -3.36000e+02
  1.12000e+02 -2.40000e+01  1.60000e+01  0.00000e+00  1.00000e+00])
```

Construction des polynômes de Lagrange

1. Créer une fonction `BASE_LAGRANGE` d'argument la liste `Xabs` , qui renvoie la liste des polynômes de la base d'interpolation de Lagrange associée aux valeurs de `Xabs`, en utilisant l'algorithme écrit dans le cours.

Indications : *On utilisera la fonction `Polynomial` de `numpy.polynomial` pour construire la liste des polynômes interpolateurs $L_0, ..., L_n$ de Lagrange. Au départ, on initialise avec `L=[]` et on prend `n=len(Xabs)-1`. Puis on construit les polynômes $L_0, L_1, ..., Ln$ dans une boucle double en calquant l'algorithme du cours. Plus précisément, on initialise avec `Pol=Polynomial([1])` qui est $Pol = 1$ juste après le premier `for` et après le second `for`, on met une instruction conditionnelle avec le code `if j!=i` et on fait le produit de `Pol` par un certain polynôme de degré 1*
On rappelle que le code `Polynomial([a,b])` est le polynôme $a + bX$.
Après être sorti de la boucle intérieure, on met `Pol` dans `L` avec `append`
Enfin, après la boucle `for` extérieure, on retourne `L` qui sera une liste de listes polynomiales

```
In [6]:   def BASE_LAGRANGE(Xabs):
   ...:       n = len(Xabs) -1 ; L=[]
   ...:       for i in range(0,n+1):
   ...:           Pol = Polynomial([1])
   ...:           for j in range(0,n+1):
   ...:               if j!=i :
   ...:                   Pol = Pol * Polynomial([-Xabs[j]/(Xabs[i]-Xabs[j]),
                          1/(Xabs[i]-Xabs[j])])
   ...:           L.append(Pol)
   ...:       return L
```

Appliquer au cas où `Xabs= [0,1,2]` et retrouver le résultat de l'exercice 01 du cours.

```
In [7]:  BASE_LAGRANGE([0,1,2])
Out[7]:
[Polynomial([ 1.  , -1.5,  0.5], domain=[-1.,  1.], window=[-1.,  1.]),
 Polynomial([ 0.,  2., -1.], domain=[-1.,  1.], window=[-1.,  1.]),
 Polynomial([ 0.  , -0.5,  0.5], domain=[-1.,  1.], window=[-1.,  1.])]
```

2. En utilisant `BASE_LAGRANGE` comme sous-procédure et la fonction `sum`, créer une fonction `POL_LAG` d'arguments `Xabs` et `Xord` qui renvoie le polynôme d'interpolation de Lagrange passant par les points dont les abscisses sont les valeurs de `Xabs` et les ordonnées sont les valeurs de `Xord`

```
In [10]: def POL_LAG(Xabs,Xord):
   ...:      n = len(Xabs)-1
   ...:      return sum(Xord[k]*BASE_LAGRANGE(Xabs)[k] for k in range(0,n+1))
```

Appliquer au cas où `Xabs=[0,1,2]` et `Xord=[1,-3,2]` et retrouver le résultat de l'exercice 05 du cours.

```
In [11]: POL_LAG([0,1,2],[1,-3,2])
Out[11]: x -> 1 - 8.5 x + 4.5 x^2
# AUTRE ECRITURE
In [12]: POL_LAG([0,1,2],[1,-3,2]).coef
Out[12]: array([ 1.  , -8.5,  4.5])
```

Une utilisation des polynômes de Lagrange

L'idée est de pouvoir faire une approximation du graphe d'une fonction en remplaçant ce graphe par celui d'un polynôme d'interpolation qui passe par un certain nombre de points de ce graphe. Il est clair que plus le nombre de points est grand, plus les deux graphes vont se coller (hors éventuellement des effets de bord que l'on laissera de côté dans ce TP, pour ceux que cela intéresse, on utilise pour éviter des effets de bords comme `Xabs` les racines de polynômes de Tchebychev).

1. Modifier la procédure `POL_LAG` en remplaçant l'argument `Xord` par `f`, où `f` est une fonction donnée et la procèdure (qui s'appelle maintenant `NEW_POL_LAG`) renvoie le polynôme d'interpolation de Lagrange qui passe par les points $(x_0, f(x_0)), ..., (x_n, f(x_n))$. Ainsi `Xord[k]` est remplaçé par `f(Xabs[k])`

```
In [16]: def NEW_POL_LAG(Xabs,f):
    ...:      n = len(Xabs)-1
    ...:      return sum(f(Xabs[k])*BASE_LAGRANGE(Xabs)[k]for k in range(0,n+1))
```

Appliquer `NEW_POL_LAG` avec $Xabs = [0, 1, 2]$ et la fonction $x \mapsto x^2$. Que remarque t-on ?

```
# RAPPEL. Code pour rentrer la fonction f
In [17]: def f(x): return x**2
```

```
In [18]: NEW_POL_LAG([0,1,2],f)
Out[18]: Polynomial([0., 0., 1.], domain=[-1., 1.], window=[-1., 1.])
# On remarque que on tombe sur f
# normal car P est un pol de degre 2 comme f donc P=f
```

2. Appliquer `NEW_POL_LAG` avec `Xabs = [-1,0,1,2,3]` et $f = \arctan$. On appellera dans la suite P_1 le dernier polynôme trouvé.

Indication *: on tapera* `import numpy as np` *et donc la fonction* arctan *est* `np.arctan`

```
In [21]: import numpy as np
In [22]: NEW_POL_LAG([-1,0,1,2,3],np.arctan)
Out[22]:
Polynomial([ 0.        , 0.92495957, -0.0311434 , -0.13956141, 0.0311434 ],
    domain=[-1., 1.], window=[-1., 1.])
In [23]: P1 = NEW_POL_LAG([-1,0,1,2,3],np.arctan)
```

Comparer $P(1.5)$ et $\arctan(1.5)$.

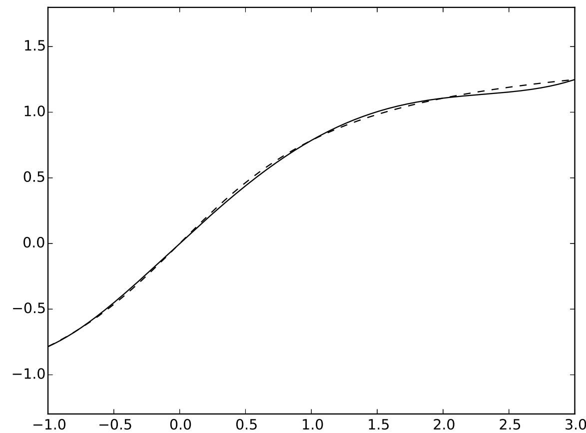
```
In [24]: P1(1.5), np.arctan(1.5)
Out[24]: (1.0040104251658764, 0.982793723247329)
```

3. On veut tracer ensemble P_1 et la fonction arctan sur $[-1, 3]$.

On tapera (les quatre instructions de la ligne 27 sont à mettre sur la même ligne, séparés par des ;) :

```
In [25]: import matplotlib.pyplot as plt
In [26]: XX = np.linspace(-1,3,500)
In [27]: plt.plot(XX,np.arctan(XX),linestyle="--");
plt.plot(XX,P1(XX),color='0'); plt.axis('equal');plt.show()
```

Attention, entre les deux `- -` et entre un `"` et un `-` dans `linestyle`, il ne doit pas y avoir d'espace.



4. On cherche des valeurs approchées de $I = \int_{-1}^{3} \arctan x \, dx$.

4-a Méthode 1. Trouver une valeur approchée de I en important `scipy.integrate` avec l'alias `integr` et la fonction `integr.quad`

```
In [28]: import scipy.integrate as integr
In [29]: integr.quad(np.arctan,-1,3)
Out[29]: (2.1570201975802648, 3.3650176253999675e-14)
```

4-b Méthode 2. Trouver une valeur approchée de I en utilisant `P2 =P1.integ()`

```
In [30]: P2 =P1.integ()
In [31]: P2
Out[31]:
Polynomial([ 0.        ,  0.        ,  0.46247979, -0.01038113, -0.03489035,
        0.00622868], domain=[-1.,  1.], window=[-1.,  1.])
In [32]: P2(3)-P2(-1)
Out[32]: 2.137736453030486
```

4-c Méthode 3. En remarquant qu'une primitive de $\arctan x$ est $x \arctan x - \frac{1}{2}\ln(x^2+1)$, calculer I.

```
In [34]: def F(x) :
    ...:     return x*np.arctan(x) - 0.5*np.log(x**2+1)
    ...:
In [35]: F(3)-F(-1)
Out[35]: 2.1570201975802648
```