Polynomes

March 1, 2019

1 Opérations sur les polynômes

Marc Lorenzi
1er mars 2019

```
In [1]: import matplotlib.pyplot as plt
        from sympy import S
        import math
        import random
```

```
In [2]: plt.rcParams['figure.figsize'] = (10, 5)
```

Voici deux compteurs, ils nous serviront à compter les additions et les multiplications effectuées par les fonctions que nous allons écrire. Si vous voyez apparaître dans le notebook une ligne du type `cnta += 1`, c'est qu'une somme (ou une différence) de coefficients de polynômes a été faite à la ligne précédente. Les fonctions incrémentant ces compteurs sont clairement signalées par une première ligne du type `global cnta`.

```
In [3]: cnta = 0
        cntm = 0
```

1.1 1. Introduction

1.1.1 1.1 Représentation des polynômes en Python

Dans ce notebook, nous nous intéressons aux diverses "opérations" (au sens large) que l'on peut effectuer dans l'algèbre $\mathbb{R}[X]$ des polynômes à coefficients réels. Un polynôme $A \in \mathbb{R}[X]$ s'écrit de façon unique

$$A = \sum_{k=0}^{\infty} a_k X^k$$

où les a_k sont des réels tous nuls sauf un nombre fini. Les a_k sont les **coefficients** du polynôme A.

Nous représenterons un tel polynôme par la liste de ses coefficients. Soyons plus précis :

- Si $A = 0$ est le polynôme nul, nous représenterons A par la liste vide $[]$.
- Si $A \neq 0$, soit n le plus grand entier naturel tel que $a_n \neq 0$ (il n'a pas un nom, cet entier ?). Nous représenterons alors A par la liste $[a_0, a_1, \ldots, a_n]$.

Il pourrait arriver qu'à la suite de certaines opérations nous obtenions un "polynôme" représenté par une liste finissant par des zéros. Il nous faudra alors **normaliser** cette liste pour respecter nos conventions de représentation des polynômes. La fonction `normaliser` effectue ce travail. Elle prend en paramètre une liste et renvoie cette liste purgée des zéros finaux.

Il pourra également nous arriver de représenter des polynômes sous forme non normalisée, c'est à dire par des listes finissant par des zéros. Nous le signalerons lorsque cela arrive.

```
In [4]: def normaliser(A):
            n = len(A) - 1
            while n >= 0 and A[n] == 0: n = n - 1
            return A[:n + 1]

In [5]: normaliser([1, 2, 0, 1, 0, 0])

Out[5]: [1, 2, 0, 1]
```

1.1.2 1.2 Évaluer un polynôme en un point

Soit A un polynôme représenté par la liste $[a_0, a_1, \ldots, a_{n-1}]$. Soit $x \in \mathbb{R}$. Comment calculer $A(x)$? Ceci peut être fait en $O(n)$ opérations. L'idée est simple. Il suffit de remarquer que

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0 = (a_{n-1}x^{n-2} + a_{n-2}x^{n-3} + \ldots + a_1)x + a_0$$

En réitérant l'opération on obtient **l'algorithme de Horner** :

$$A(x) = (\ldots((a_{n-1}x + a_{n-2})x + a_{n-3})x + \ldots)x + a_0$$

```
In [6]: def evaluer(A, x):
            s = 0
            n = len(A)
            for k in range(len(A) - 1, -1, -1):
                s = s * x + A[k]
            return s

In [7]: evaluer([1, 2, 1, 1], 2)

Out[7]: 17
```

La boucle `for` effectue n itérations. À chaque itération on effectue une addition et une multiplication, soit, au total, $2n$ opérations. Inutile d'en dire plus.

1.1.3 1.3 Le degré d'un polynôme

Soit $A = \sum_{k=0}^{\infty} a_k X^k \in \mathbb{R}[X]$.

- Si $A \neq 0$, le degré de A est le plus grand entier k tel que $a_k \neq 0$.
- Le degré du polynôme nul est $-\infty$.

La fonction `degre` renvoie le degré du polynôme A. Enfin, presque. Si A est le polynôme nul la fonction renvoie -1. Il faudra nous en souvenir dans la suite. Je rappelle que A est implicitement supposé normalisé. Si ce n'est pas le cas, la fonction `degre` renvoie un résultat inexact.

```
In [8]: def degre(A):
            return len(A) - 1

In [9]: degre([1, 3, 2, 1])

Out[9]: 3

In [10]: degre([])

Out[10]: -1
```

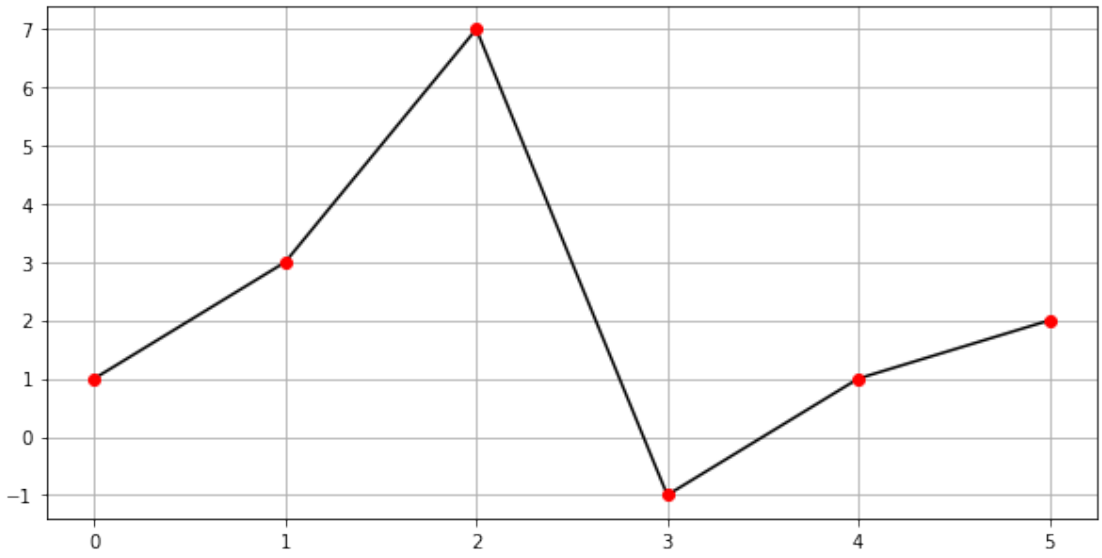
1.1.4 1.4 Représentations lisibles d'un polynôme

La représentation d'un polynôme sous forme de liste est parfaitement lisible. Mais bon, pour faire plaisir aux fanatiques de l'indéterminée X, voici une fonction `vers_chaine`. Elle prend un polynôme A en paramètre et renvoie une chaîne de caractères qui est la représentation "classique" du polynôme A.

```
In [11]: def vers_chaine(A):
             n = degre(A)
             if n == -1: return "0"
             s = ''
             for k in range(n + 1):
                 fmt = '%s*X**%s'
                 if A[k] > 0: s += ('+' + fmt) % (A[k], k)
                 elif A[k] < 0: s += fmt % (A[k], k)
             return s

In [12]: vers_chaine([1, -3, 0, 1])

Out[12]: '+1*X**0-3*X**1+1*X**3'
```

Prévoyons aussi une représentation sous forme de graphique, pour les artistes. Cela ne nous coûte pas grand chose.

```
In [13]: def plot_poly(A):
             plt.plot(A, '-k')
             plt.plot(A, 'or')
             plt.grid()

In [14]: plot_poly([1, 3, 7, -1, 1, 2])
```



1.1.5 1.5 Coefficients

Nous allons dans la suite passer notre temps à accéder aux coefficients d'un polynôme. Facile, me direz-vous, le coefficient de degré k du polynôme A, c'est $A[k]$? Eh bien pas exactement. Si $k \leq d^o A$, c'est vrai. Mais sinon ? Histoire d'éviter des problèmes, écrivons illico une fonction `coef`.

```
In [15]: def coef(A, k):
             if k <= degre(A): return A[k]
             else: return 0
```

Et tant que nous y sommes ...

```
In [16]: def coef_dominant(A):
             n = degre(A)
             if n < 0: raise Exception('Polynôme nul')
             else: return A[-1]

In [18]: coef_dominant([1, 3, 2, 7])

Out[18]: 7
```

1.2 2. Somme, produit par un scalaire

1.2.1 2.1 Somme et différence

Soient $A = \sum_{k=0}^{\infty} a_k X^k$ et $B = \sum_{k=0}^{\infty} b_k X^k$ deux polynômes. La somme de A et B est le polynôme

$$A + B = \sum_{k=0}^{\infty} (a_k + b_k) X^k$$

On définit de même $A - B$. Rappelons que

$$d^o(A + B) \leq \max(d^o A, d^o B)$$

et de même pour $A - B$. Les fonctions `somme` et `difference` ne présentent pas de difficulté. Il faut juste ne pas oublier de normaliser le résultat avant de le renvoyer.

Ces deux fonctions prennent un paramètre optionnel `norm`. Si `norm` vaut `True`, ce qui est vrai par défaut, le polynôme renvoyé est normalisé.

```
In [19]: def somme(A, B, norm=True):
             global cnta
             n = max(degre(A), degre(B))
             R = (n + 1) * [0]
             for k in range(n + 1):
                 R[k] = coef(A, k) + coef(B, k)
                 cnta += 1
             if norm: return normaliser(R)
             else: return R

In [20]: somme([1, 2, 1], [2, 3, -1])

Out[20]: [3, 5]

In [21]: somme([], [1, 2, 3])

Out[21]: [1, 2, 3]

In [22]: def difference(A, B, norm=True):
             global cnta
             n = max(degre(A), degre(B))
             R = (n + 1) * [0]
             for k in range(n + 1):
                 R[k] = coef(A, k) - coef(B, k)
                 cnta += 1
             if norm: return normaliser(R)
             else: return R

In [23]: difference([1, 2, 1], [2, 3, -1])

Out[23]: [-1, -1, 2]
```

1.2.2 2.2 Produit par un scalaire

Soit $A = \sum_{k=0}^{\infty} a_k X^k$un polynôme. Soit $t \in \mathbb{R}$. La produit de t et A est le polynôme

$$tA = \sum_{k=0}^{\infty} (ta_k) X^k$$

Aucune difficulté pour écrire une fonction `prod_scal`. Gardons juste à l'esprit que toutes nos fonctions doivent renvoyer des polynômes normalisés. Il faut donc traiter à part le cas où $t = 0$.

```
In [24]: def prod_scal(t, A):
             global cntm
             if t == 0: return []
             else:
                 cntm += len(A)
                 return [t * c for c in A]

In [25]: prod_scal(2, [1, 3, 2])

Out[25]: [2, 6, 4]
```

1.3 3. Produit, puissances

1.3.1 3.1 Produit de deux polynômes

Soient $A = \sum_{k=0}^{\infty} a_k X^k$ et $B = \sum_{k=0}^{\infty} b_k X^k$ deux polynômes. Le produit de A et B est le polynôme

$$AB = \sum_{k=0}^{\infty} c_k X^k$$

où $c_k = \sum_{i=0}^{k} a_i b_{k-i}$.
Rappelons que $d^o(AB) = d^oA + d^oB$.

Écrivons tout d'abord une fonction `somme_deg` qui ajoute des degrés. Euh, n'est-ce pas tout simplement "+" ? Non, à cause du polynôme nul. À cause des conventions que nous avons faites, nous devons faire en sorte que -1 soit absorbant pour l'addition (si j'ose dire).

```
In [26]: def somme_deg(m, n):
             if m == -1 or n == -1: return -1
             else: return m + n

In [27]: def produit(A, B):
             global cnta, cntm
             p = somme_deg(degre(A), degre(B))
             C = (p + 1) * [0]
             for k in range(p + 1):
                 for i in range(k + 1):
                     C[k] += coef(A, i) * coef(B, k - i)
                     cnta += 1
                     cntm += 1
             return C

In [28]: cnta, cntm = 0, 0
         print(produit([1, 2, 1], [1, 3, 3, 1]))
         print(cnta, cntm)

[1, 5, 10, 10, 5, 1]
21 21
```

Quelle est la complexité de la fonction `produit` en nombre d'opérations sur des coefficients de polynômes ? Soient $m = d^oA$ et $n = d^oB$. Supposons A et B non nuls. Le degré de $C = AB$ est $p = m + n$, nous avons donc $p + 1$ coefficients à calculer. Pour $k = 0, 1, \ldots, p$, on a

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

Le calcul de c_k nécessite donc $k + 1$ additions et $k + 1$ multiplications. Vous me direz qu'il n'y a que k additions, mais regardez le code de `produit` ...

Le nombre total d'additions ou de multiplications à effectuer est donc

$$\sum_{k=0}^{p} (k+1) = \frac{1}{2}(p+1)(p+2)$$

Le nombre total d'opérations est donc $(p + 1)(p + 2)$.

Sur l'exemple ci-dessus, $m = 2$, $n = 3$ et $p = 5$. D'où $\frac{1}{2}(p + 1)(p + 2) = \frac{1}{2}6 \times 7 = 21$, ce qui est bien la valeur des compteurs en fin de calcul.

Notre algorithme de multiplication a donc une complexité en $O((m + n)^2)$. Pour des polynômes de degrés raisonables, cela ne pose pas de problème. En revanche, cela n'est pas acceptable lorsque nos polynômes possèdent de grands degrés. Nous verrons à la fin de ce notebook un algorithme dû à Anatolii Alexevich Karatsuba meilleur que notre algorithme "naïf".

Remarque : "naïf", en algorithmique, cela veut dire la même chose que "j'utilise les définitions de mon cours de maths" :-).

Exercice : Les mathématiciens sont-ils naïfs ?

1.3.2 3.3 Puissances

Maintenant que nous savons calculer un produit, calculer une puissance ne pose pas de problème. La fonction `puissance` prend en paramètres un polynôme A et un entier n. Elle renvoie A^n. Cette fonction utilise l'algorithme d'exponentiation rapide. Si vous ne savez pas ce que sait, allez consulter le notebook à ce sujet.

```
In [29]: def puissance(A, n):
             if n == 0: return [1]
             else:
                 B = puissance(produit(A, A), n // 2)
                 if n % 2 == 0: return B
                 else: return produit(A, B)
```

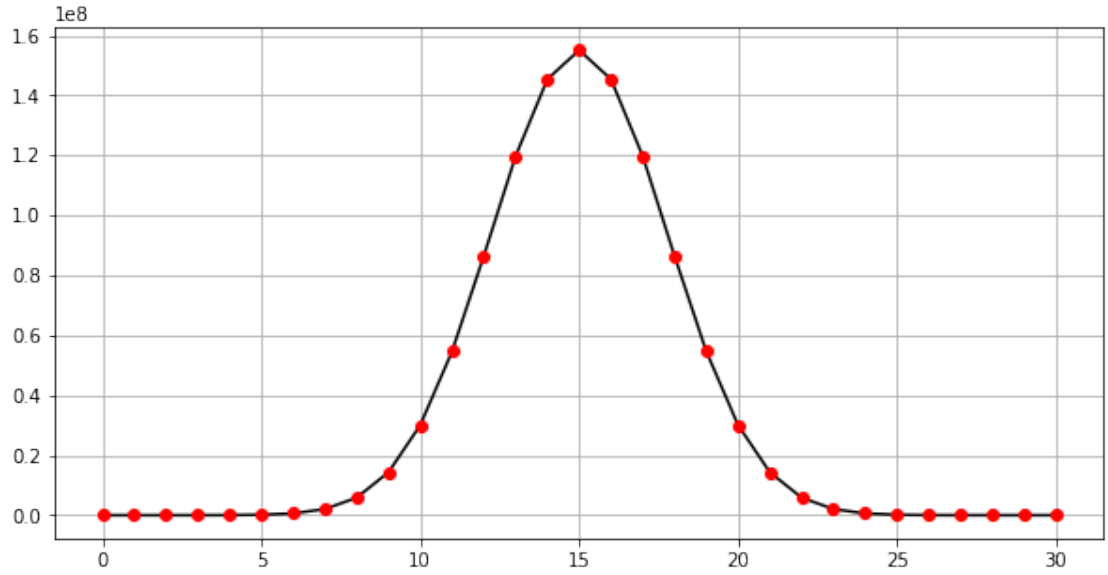
```
In [30]: puissance([1, 1], 30)
```

```
Out[30]: [1,
          30,
          435,
          4060,
          27405,
          142506,
          593775,
          2035800,
```

```
5852925,
14307150,
30045015,
54627300,
86493225,
119759850,
145422675,
155117520,
145422675,
119759850,
86493225,
54627300,
30045015,
14307150,
5852925,
2035800,
593775,
142506,
27405,
4060,
435,
30,
1]
```

De façon plus graphique, voici $(1+X)^{30}$. Si vous reconnaissez des coefficients binomiaux ne soyez pas surpris.

```
In [31]: plot_poly(puissance([1, 1], 30))
```



1.4 4. Composée

$\mathbb{R}[X]$ est également muni d'une opération de composition. Soient $A = \sum_{k=0}^{\infty} a_k X^k$ et B deux polynômes. La composée de A et B est

$$A \circ B = \sum_{k=0}^{\infty} a_k B^k$$

Si B n'est pas constant, on a $d^o(A \circ B) = d^o A \times d^o B$.

```
In [32]: def composee(A, B):
             C = []
             for k in range(degre(A) + 1):
                 C = somme(C, prod_scal(coef(A, k), puissance(B, k)))
             return C

In [33]: composee([1, 2, 3, 1], [1, 1, 2])

Out[33]: [7, 11, 28, 25, 30, 12, 8]
```

1.5 5. Division

1.5.1 5.1 La division euclidienne

Proposition : Soient $A, B \in \mathbb{R}[X]$, $B \neq 0$. Il existe un unique couple (Q, R) de polynômes tel que $A = BQ + R$ et $d^o R < d^o B$.

Comment calculer Q et R ? Soient n et m les degrés respectifs de A et B.

- Si $n < m$, c'est évident : $Q = 0$ et $R = A$.
- Sinon, soit $c = \frac{a_n}{b_m}$. Soit $D = cX^{n-m}B$. Soit $A' = A - BD$. Alors $d^o A' < d^o A$. Si nous savons diviser A' par B, mettons $A' = BQ' + R$, alors nous savons aussi diviser A par B puisque $A = B(D + Q') + R$.

La fonction `division` effectue donc une simple boucle `while`. On initialise deux variables Q et R à $Q = 0$ et $R = A$. On a donc $A = BQ + R$ avant la première itération. À chaque itération, on effectue les opérations décrites ci-dessus. À la fin de chaque itération, on a donc toujours $A = BQ + R$. Comme le degré de R diminue strictement, la boucle termine. À l'issue de la dernière itération, on a $A = BQ + R$ et $d^o R < d^o B$. On a donc trouvé le quotient et le reste de la division euclidienne de A par B.

```
In [34]: def division(A, B):
             global cntm
             m = degre(B)
             if m < 0: raise Exception('Division par zéro')
             Q = []
             R = A
             n = degre(R)
             while n >= m:
                 c = coef_dominant(R) / coef_dominant(B)
                 cntm += 1
```

```
            D = decaler([c], n - m)
            R = difference(R, decaler(prod_scal(c, B), n - m))
            Q = somme(Q, D)
            n = degre(R)
        return (Q, R)
```

La fonction `decaler` multiplie A par X^n, c'est à dire rajoute n zéros en tête de la liste A.

```
In [35]: def decaler(A, n):
             return (n * [0]) + A

In [36]: decaler([1, 2, 3], 5)

Out[36]: [0, 0, 0, 0, 0, 1, 2, 3]

In [37]: A = S([1, 4, 6, 4, 1])
         B = [1, 3, 2]
         Q, R = division(A, B)
         print(Q, R)

[7/8, 5/4, 1/2] [1/8, 1/8]
```

Petite vérification ...

```
In [39]: somme(produit(B, Q), R)

Out[39]: [1, 4, 6, 4, 1]
```

Remarque : À quoi sert le S dans l'exemple ci-dessus ? À effectuer des calculs exacts. Sans celui-ci, les calculs seraient faits en flottants.

1.5.2 5.2 Complexité

Combien d'opérations sur des coefficients de polynômes doit-on faire lors d'une division euclidienne ? Soient A, B de degrés respectifs n et m, où $n \geq m$. Le pire cas se produit (preuve ?) lorsque, à chaque itération, le degré du reste chute seulement de 1. On effectue alors $n - m + 1$ itérations que nous allons numéroter $k = 0, 1, \ldots, n - m$.

- Itération 0 : le calcul de c coûte une division. D ne coûte rien, simple décalage. Pour calculer R, on calcule d'abord cB : $m + 1$ multiplications. Puis on fait la différence $R - cDB$: $n + 1$ soustractions. Le calcul de Q demande $n - m + 1$ additions, la plupart, avouons-le, triviales. Total : $1 + (m + 1) + (n + 1) + (n - m + 1) = 2n + 4$ opérations.
- Itération 1 : R est maintenant de degré $n - 1$. le calcul de c coûte une division. D est toujours gratuit. Pour calculer R, on calcule d'abord cB : $m + 1$ multiplications. Puis on fait la différence $R - cDB$: n soustractions. Le calcul de Q demande toujours $n - m + 1$ additions. Total : $1 + (m + 1) + n + (n - m + 1) = 2n + 3$ opérations.
- Itération 2 : même raisonnement, $2n + 2$ opérations.

- ...
- Itération k : $2n+4-k$ opérations.

Au total, on effectue donc en pire cas $\sum_{k=0}^{n-m}(2n+4-k) = \frac{1}{2}(n-m+1)(3n+m+8)$ opérations. Petite vérification ?

```
In [40]: cnta, cntm = 0, 0
         A = 101 * [1]
         B = [1, 2, 3, 4, 1]
         Q, R = division(A, B)
         print(cnta + cntm)
```

```
15132
```

```
In [41]: n = 100
         m = 4
         print((n - m + 1) * (3 * n + m + 8) / 2)
```

```
15132.0
```

Nous sommes comblés.

Remarque : Notons $C_\times$ et $C_\div$ les complexités de la multiplcation et de la division euclidienne. Nous avons

$$C_\times = (m+n+1)(m+n+2)$$

et, en pire cas,

$$C_\div = \frac{1}{2}(n-m+1)(3n+m+8)$$

Histoire de comparer des choses comparables, fixons m et faisons tendre n vers l'infini. Il vient

$$C_\times \sim n^2 \quad \text{et} \quad C_\div \sim \frac{3}{2}n^2$$

Faire une division est à peine 50% plus coûteux que faire une multiplication.

1.5.3 5.3 Exercices

Nous avons maintenant défini les opérations présentes dans un tout bon cours de maths sur les polynômes. L'existence d'une division euclidienne dans l'anneau $\mathbb{R}[X]$ fait de celui-ci un **anneau euclidien**. Dans un tel anneau on dispose des notions de pgcd et de ppcm. Le théorème de Bézout y est valide.

1. Écrire une fonction `pgcd` prenant en paramètres deux polynômes A et B et renvoyant leur pgcd. Cette fonction implémentera bien entendu le célèbre algorithme d'Euclide.
2. Écrire une fonction `ppcm`. C'est immédiat à partir de `pgcd`.
3. Écrire une fonction `bezout` prenant en paramètres deux polynômes A et B et renvoyant un triplet (U, V, Δ) de polynômes tel que $UA + VB = \Delta$ et $\Delta = A \wedge B$. Il suffit de reprendre de façon un peu plus poussée la fonction de l'exercice 1.

1.6 6. L'algorithme de Karatsuba

Dans cette section, nous ne supposons pas forcément les polynômes représentés sous forme normalisée. Nous appellerons **longueur** d'un polynôme la longueur d'une liste qui le représente. Cette quantité est mal définie, nous devrions plutôt dire la longueur **d'une** représentation du polynôme par une liste. Mais faisons court ...

1.6.1 6.1 L'idée

Soit $p \in \mathbb{N}^*$. Soient A, B deux polynômes de longueur $n = 2p$. Écrivons $A = A' + A''X^p$ et $B = B' + B''X^p$ où les polynômes A', A'', B', B'' sont de longueur p. On a

$$AB = (A' + A''X^p)(B' + B''X^p) = A'B' + (A'B'' + A''B')X^p + A''B''X^{2p}$$

Pour multiplier A et B il suffit donc de faire **quatre** multiplications de polynômes de longueur p (je vous laisse compter). On ne compte pas les multiplications par des puissances de X, celles-ci peuvent être effectuées par de simples décalages. Peut-on faire mieux ? Oui !

Posons $C = (A' + A'')(B' + B'')$. On a $C = A'B' + A''B'' + A'B'' + A''B'$. Ainsi, si l'on pose $C' = A'B'$ et $C'' = A''B''$, on a

$$AB = C' + (C - C' - C'')X^p + C''X^{2p}$$

On constate donc que **trois** multiplications suffisent.

Ceci suggère donc un algorithme de type "diviser pour régner" pour multiplier deux polynômes. On coupe les polynômes en deux, on effectue trois produits de polynômes de longueurs "deux fois plus petites", on recolle les morceaux. Il n'y a plus qu'à écrire du code. Mais avant tout, voyons en quoi nous y gagnons.

Remarque : Si $n = 2p + 1$ (où $p \geq 0$) est un nombre impair, on peut décider, par exemple, de prendre les polynômes A et B "prime" de taille p et les polynômes A et B "seconde" de taille $p + 1$. Cela nous donne bien entendu les mêmes formules.

1.6.2 6.2 Complexité

Notons M_n le nombre de multiplications de coefficients à effectuer pour calculer avec l'algorithme de Karatsuba le produit de deux polynômes de longueur n.

- Nous n'avons pas parlé d'initialisation pour l'instant, mais Pour $n = 0$ il n'y a rien à faire. Ainsi, $M_0 = 0$.
- Pour $n = 1$, on a $M_1 = 1$. Multiplier deux polynômes constants ce n'est pas bien délicat.
- Soit $p \geq 1$. On a clairement $M_{2p} = 3M_p$.
- Enfin, soit $p \geq 0$. On a $M_{2p+1} = 2M_{p+1} + M_p$. Je vous laisse regarder pourquoi.

Je vais ici me contenter de regarder les polynômes dont la longueur est une puissance de 2. Que vaut $u_n = M_{2^n}$ pour $n \in \mathbb{N}$?

- Pour $n = 0$, on a $2^n = 1$: $u_0 = M_1 = 1$.
- Pour $n \geq 0$, $u_{n+1} = M_{2^{n+1}} = 3M_{2^n} = 3u_n$.

Ainsi, la suite $(u_n)_{n\geq 0}$ est géométrique de raison 3, de premier terme 1. Pour tout entier n, $u_n = 3^n$.

Si n est une puissance de 2, nous avons donc

$$M_n = 3^{\lg n} = n^{\lg 3}$$

où lg désigne le logarithme en base 2.

Pour l'algorithme naïf, nous avions besoin (voir paragraphe 3.2) de $n(2n-1) \sim 2n^2$ opérations. Y a-t-on gagné ? oui, car $\lg 3 < \lg 4 = 2$, et donc $n^{\lg 3} = o(n^2)$. L'algorithme de Karatsuba est donc asymptotiquement meilleur que l'algorithme naïf.

Euh, me direz-vous, et les additions dans tout ça ? Il faudrait peut-être les compter aussi ? Vous avez raison, nous le ferons, mais attendons d'avoir écrit la fonction `karatsuba`.

1.6.3 6.3 Complexité : esquisse du cas général

Si n n'est pas une puissance de 2, M_n n'est certainement pas égal à $n^{\lg 3}$ vu que ce nombre n'est en général pas un entier, alors que M_n en est un. On peut cependant démontrer que l'ordre de grandeur est le bon : $M_n = O(n^{\lg 3})$ lorsque n tend vers l'infini. Je ne le ferai pas ici, je me contenterai d'une petite illustration. Nous avons $M_0 = 0$, $M_1 = 1$, et la récurrence

$$M_{2p} = 3M_p, M_{2p+1} = 2M_{p+1} + M_p$$

Voici donc la fonction `nbmult_kara` qui prend en paramètre un entier n et renvoie M_n.

```
In [42]: def nbmult_kara(n):
             if n == 0: return 0
             elif n == 1: return 1
             else:
                 p = n // 2
                 if n % 2 == 0: return 3 * nbmult_kara(p)
                 else: return 2 * nbmult_kara(p + 1) + nbmult_kara(p)
```

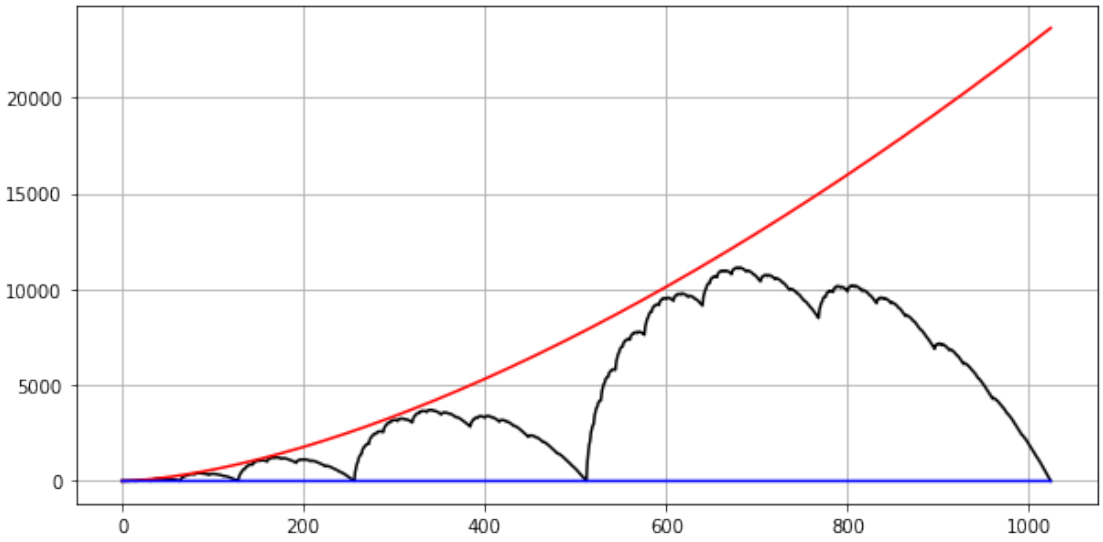
```
In [43]: print([nbmult_kara(n) for n in range(20)])
```

```
[0, 1, 3, 7, 9, 17, 21, 25, 27, 43, 51, 59, 63, 71, 75, 79, 81, 113, 129, 145]
```

Ci dessous,

- en noir le tracé de $M_n - n^{\lg 3}$.
- en rouge le tracé de $\frac{2}{5}n^{\lg 3}$. Le facteur $\frac{2}{5}$, choisi par tâtonnements, n'est sans doute pas optimal.
- en bleu l'axe Ox.

```
In [44]: l = math.log(3, 2)
         xs = range(0, 1025)
         ys = [nbmult_kara(n) - n ** l for n in xs]
         zs = [0.4 * n ** l for n in xs]
         ws = [0 for n in xs]
         plt.plot(xs, ys, 'k')
         plt.plot(xs, zs, 'r')
         plt.plot(xs, ws, 'b')
         plt.grid()
```



La courbe noire mériterait sans doute un notebook à elle seule … on peut en tout cas espérer (?) la

Conjecture non prouvée : Pour tout $n \in \mathbb{N}$,

$$n^{\lg 3} \leq M_n \leq \frac{7}{5} n^{\lg 3}$$

Exercice : Pour $n = 2^p + 2^{p-1}$, $p \geq 1$, montrer que

$$M_n = C n^{\lg 3}$$

où

$$C = \frac{7}{3} \left(\frac{2}{3}\right)^{\lg 3} \simeq 1.227$$

Exemple motivant pour l'exercice :

```
In [47]: 7 / 3 * (2 / 3 * (1024 + 512)) ** math.log(3, 2)

Out[47]: 137781.00000000012

In [48]: nbmult_kara(1024 + 512)

Out[48]: 137781
```

Bon, assez tergiversé, codons l'algorithme de Karatsuba.

1.6.4 6.4 L'algorithme

pour ne pas rendre le code inutilement compliqué, nous allons supposer que la fonction `karatsuba` que nous allons écrire prend en paramètres deux polynômes A et B de même longueur. Nous verrons plus loin comment procéder lorsque les longueurs sont distinctes. A et B sont donc représentés par des listes de même taille.

Remarque : Un tout petit détail avant de nous lancer. Pour que nos polynômes soient bien représentés par des listes de même taille, il ne faut pas normaliser les sommes et les différences, ce qui explique dans le code ci-dessous la présence du paramètre `False`. Dans le cas $n = 1$ (polynômes de degré 0), j'avoue avoir un peu abusé : par souci de cohérence la fonction renvoie une liste de taille 2, qui est en fait un polynôme de degré 0, non normalisé.

```
In [49]: def karatsuba(A, B):
             global cntm
             n = len(A)
             if n == 0: return []
             elif n == 1:
                 cntm += 1
                 return [coef(A, 0) * coef(B, 0), 0]
             else:
                 p = n // 2
                 A1, A2 = (A[:p], A[p:])
                 B1, B2 = (B[:p], B[p:])
                 C1 = karatsuba(A1, B1)
                 C2 = karatsuba(A2, B2)
                 C = karatsuba(somme(A1, A2, False), somme(B1, B2, False))
                 D = difference(difference(C, C1, False), C2, False)
                 U = decaler(D, p)
                 V = decaler(C2, 2 * p)
                 return somme(somme(C1, U, False), V, False)
```

Maintenant, testons. Prenons $A = (1 + X)^{15}$. Calculons A^2 avec `karatsuba` et `produit`, et comparons.

```
In [50]: A = puissance([1, 1], 15)
         cnta, cntm = 0, 0
         B = karatsuba(A, A)
         print(B)
         print(cnta, cntm)
         cnta, cntm = 0, 0
         C = produit(A, A)
         print(cnta, cntm)
         print(difference(B, C))
```

```
[1, 30, 435, 4060, 27405, 142506, 593775, 2035800, 5852925, 14307150, 30045015, 54627300, 864932
845 81
496 496
[]
```

Tout a l'air de fonctionner. De plus :

- `karatsuba` a effectué 845 additions et 81 multiplications.
- `produit` a effectué 496 additions et 496 multiplications.

1.6.5 6.5 Combien d'opérations, précisément ?

Nous avons déjà compté le nombre de multiplications de coefficients effectuées par l'algorithme de Karatsuba. Mais combien d'additions de coefficients, au juste ? Notons A_n le nombre d'additions nécessaires pour le produit de deux polynômes de longueur n.

Prenons pour A et B deux polynômes de longueur paire $n = 2p$, où $p \geq 1$.

Quelles sont les longueurs des polynômes qui interviennent dans la fonction `karatsuba` ? Faisons un petit briefing.

- A_1, A_2, B_1, B_2 : longueur p.
- C_1, C_2 : longueur $2p$.
- C : longueur $2p$. Rappelons que nous ne normalisons pas les sommes.
- D : longueur $2p$.
- U : longueur $3p$.
- V : longueur $4p$.
- Calcul de C : 2 appels à `somme`. Ces appels se font sur des polynômes de longueur p : il nous faut faire pour cela $2p$ additions de coefficients.
- Calcul de D : 2 appels à `difference`. Comme les polynômes C, C_1 et C_2 sont de longueur $2p$, il nous faut faire $2 \times 2p = 4p$ additions de coefficients.
- Nous avons également deux appels à `somme` à la dernière ligne de la fonction `karatsuba`. U est de longueur $3p$ et V est de longueur $4p$. Cela fait donc encore $7n$ additions de coefficients.
- Enfin, il ne faut pas oublier les additions effectuées lors des 3 appels récursifs : cela nous en fait encore $3A_p$ additions.

Finalement,

$$A_{2p} = 3A_p + 13p$$

Et, bien entendu, $A_0 = A_1 = 0$.

Le 13, avouons-le, nous fait un peu peur. En plus, 13 ça porte malheur. Mais pas d'inquiétude, on n'est pas superstitieux :-).

Exercice : Déterminer une relation reliant, pour $p \geq 0$, les quantités A_{2p+1}, A_p et A_{p+1}.

Que vaut précisément A_n ? Nous allons regarder uniquement le cas où n est une puissance de 2.

Posons $v_n = A_{2^n}$. On a $v_0 = A_1 = 0$ et, pour tout $n \in \mathbb{N}$,

$$v_{n+1} = 3v_n + 13 \times 2^n$$

Proposition : Pour tout entier naturel n, $v_n = 13(3^n - 2^n)$.

Démonstration : Récurrence sur n. Pour $n = 0$ c'est clair. Soit $n \in \mathbb{N}$. Supposons la propriété vraie pour n. On a alors

$$v_{n+1} = 3v_n + 13 \times 2^n = 39(3^n - 2^n) + 13 \times 2^n = 13(3^{n+1} - 2^{n+1})$$

Lorsque n est une puissance de 2, nous avons donc

$$A_n = 13(n^{\lg 3} - n)$$

Rappelons nous le nombre de multiplications : $M_n = n^{\lg 3}$. Le nombre total d'opérations effectuées par l'algorithme de Karatsuba lorsque n est une puissance de 2 est donc

$$K_n = 14n^{\lg 3} - 13n$$

Pour $n = 4$, par exemple :

```
In [51]: 13 * (3 ** 4 - 2 ** 4)
```

```
Out[51]: 845
```

On retrouve bien les 845 additions vues un peu plus haut.

1.6.6 6.6 Karatsuba vs naïf. Qui gagne ?

Alors, quel est le meilleur algorithme ? Karatsuba ou l'algorithme naïf ? Pour de grandes valeurs de n c'est clairement l'algorithme de Karatsuba. Mais grandes comment ? Grandes beaucoup ou grandes pas beaucoup ? Faisons un petit calcul.

Voici deux fonctions renvoyant le nombre d'opérations effectuées par nos algorithmes pour des polynômes de longueur n puissance de 2.

```
In [55]: def ops_karatsuba(n):
             return 14 * n ** math.log(3, 2) - 13 * n

         def ops_naif(n):
             return 2 * n * (2 * n - 1)
```

Faisons la différence, évaluons, et regardons ce que l'on obtient.

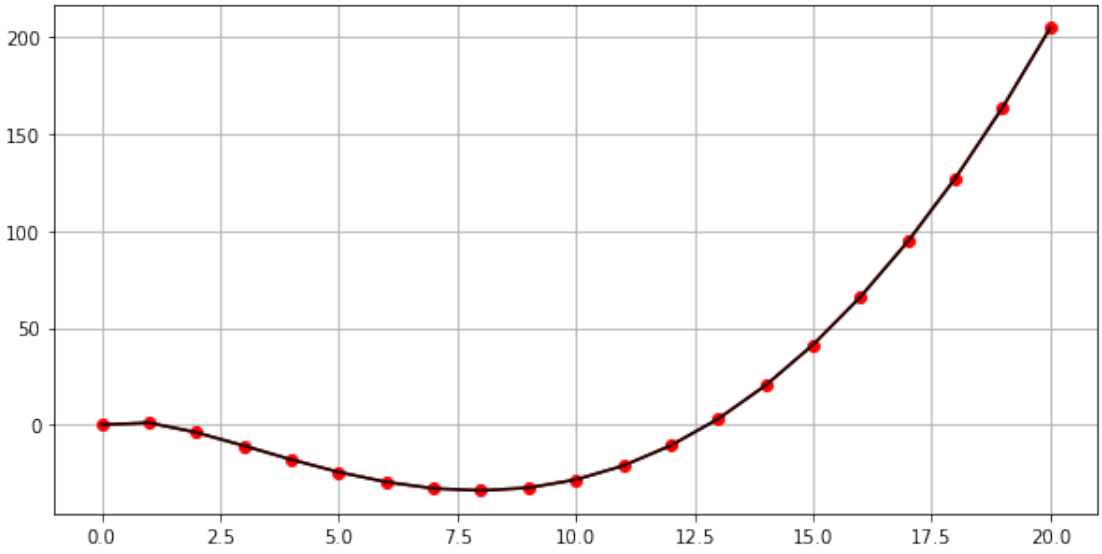
```
In [56]: def phi(n):
             return ops_naif(n) - ops_karatsuba(n)
```

```
In [57]: s = [phi(k) for k in range(21)]
         print(s)
```

```
[0.0, 1.0, -4.000000000000007, -10.863314925675652, -18.00000000000003, -24.46066870372249, -29.
```

On voit que $\varphi(k)$ devient positif pour $k \geq 13$ (le 13 est une pure coïncidence). L'algorithme de Karatsuba effectue donc moins d'opérations que l'algorithme naïf pour des polynômes de degré supérieur ou égal à 12. Petit dessin pour ceux qui ne veulent pas compter jusqu'à 13 ?

```
In [58]: plt.plot(s, '-or')
         plt.plot(s, '-k')
         plt.grid()
```



Dans les faits, notre fonction `karatsuba` n'est pas très efficace. Pour ne citer que quelques uns de ses défauts :

- appels récursifs entraînant des recopies de listes.
- les appels à `decaler` ne sont pas gratuits.
- Des additions et des soustractions sur des polynômes contenant beaucoup de zéros. Clairement, le nombre d'additions non triviales est bien moins important que notre estimation.

Telle quelle, notre fonction ne devient plus efficace que la fonction naïve que pour des polynômes de degré environ 100 (testez !). Allez, un dernier test avec un gros polynôme ...

1.6.7 6.7 Un dernier test

Prenons $A = 1 + X + X^2 + \ldots + X^{4095}$ et calculons A^2.

```
In [59]: A = 4096 * [1]
```

D'abord, A^2 avec `karatsuba`. Environ 14 secondes sur ma machine.

```
In [60]: cnta, cntm = 0, 0
         B = karatsuba(A, A)
         print(cnta, cntm)
```

```
6855485 531441
```

Puis A^2 avec `produit`. Environ 1 minute sur ma machine. Notre fonction `karatsuba` n'est donc pas totalement sans intérêt :-).

```
In [61]: cnta, cntm = 0, 0
         B = produit(A, A)
         print(cnta, cntm)
```

```
33550336 33550336
```

Exercice : Calculez explicitement A^2. Réponse : 1, 2, 3, 4, et ensuite ?

1.6.8 6.8 Et si les polynômes n'ont pas la même longueur ?

Que faire ? Il suffit par exemple de "dénormaliser" le polynôme de plus petite longueur pour que sa longueur soit égale à celle du second. Cela ne ne changera pas l'ordre de grandeur de la complexité, $O(n^{\lg 3})$, où n est la plus grande des longueurs des deux polynômes.

Voici notre fonction finale `produit_kara`. Elle renvoie, histoire de faire propre, un polynôme normalisé.

```
In [62]: def produit_kara(A, B):
             m, n = len(A), len(B)
             if m == 0 or n == 0: return []
             else:
                 A = A + (n - m) * [0]
                 return normaliser(karatsuba(A, B))
```

```
In [63]: print(produit_kara([1, 2, 3], [3, 4, 5, 6, 7]))
         print(produit([1, 2, 3], [3, 4, 5, 6, 7]))
```

```
[3, 10, 22, 28, 34, 32, 21]
[3, 10, 22, 28, 34, 32, 21]
```

Nous **savons** que `produit_kara` fonctionne. Mais comme un certain Matthieu, nous voudrions voir pour croire. Histoire de clôre le dossier, calculons 1000 produits de polynômes "aléatoires" avec `produit_kara` et `produit`, et comparons les résultats.

```
In [64]: def random_poly():
             n = random.randint(0, 20)
             s = n * [0]
             for k in range(n):
                 s[k] = random.randint(-100, 100)
             return s
```

```
In [65]: print(random_poly())
```

```
[27, -52]
```

```
In [66]: def test(n):
             b = True
             for k in range(n):
                 A = random_poly()
                 B = random_poly()
                 C = produit_kara(A, B)
                 D = produit(A, B)
                 E = difference(C, D)
                 if E != []:
                     b = False
                     break
             print(b)

In [67]: test(1000)

True
```

Nous avons de bonnes raisons, théoriques et pratiques, de penser que `produit_kara` fonctionne bien :-).

1.6.9 6.9 Peut-on faire mieux ?

Oui, en utilisant la transformée de Fourier rapide (FFT, voir le notebook à ce sujet). Pour multiplier deux polynômes de longueur n :

- On calcule leurs transformées de Fourier en ajustant auparavant les polynômes à la longueur $2n$: $O(n \lg n)$ opérations.
- On multiplie les transformées, qui sont des listes de longueur $2n$, **terme à terme** : $2n$ opérations.
- On calcule la transformée de Fourier inverse du résultat : $O(n \lg n)$ opérations.

On obtient alors le produit des deux polynômes. Cet algorithme permet de multiplier deux polynômes de longueur n en $O(n \lg n)$ opérations. Il devient efficace par rapport à l'algorithme de Karatsuba lorsque les polynômes sont de degré élevé.

L'algorithme basé sur la FFT est, à ma connaissance, le meilleur algorithme connu pour multiplier deux polynômes.