Gram_Schmidt

June 20, 2021

1 L'algorithme de Gram-Schmidt

Marc Lorenzi

20 juin 2021

[1]:
```
from sympy import *
init_printing()
```

2 1. Le théorème de Gram-Schmidt

Soit E un $\mathbb{R}$-espace vectoriel de dimension finie n muni d'un produit scalaire $< \bullet, \bullet >$. Soit $\mathcal{B} = (e_0, e_1, \ldots, e_{n-1})$ une base de E. On fabrique par récurrence forte sur k une famille $\mathcal{B}' = (u_0, u_1, \ldots, u_{n-1})$ de vecteurs de E en posant :

- $u_0 = e_0$.
- Pour $1 \leq k \leq n-1$,

$$u_k = e_k - \sum_{j=0}^{k-1} \frac{< e_k, u_j >}{< u_j, u_j >} u_j$$

Théorème : $\mathcal{B}'$ est une base orthogonale de E, vérifiant de plus $\forall k \in [0, n-1], Vect(u_0, \ldots, u_k) = Vect(e_0, \ldots, e_k)$.

Démonstration : elle se trouve dans tous les bons cours de mathématiques :-).

3 2. L'algorithme

3.0.1 2.1 La fonction `gram_schmidt`

Écrivons la fonction `gram_schmidt`. Elle prend en paramètre une base $\mathcal{B}$ de notre espace vectoriel E. Oui, mais c'est qui, E ? Il faut assi passer E en paramètre, d'une façon ou d'une autre. Si on regarde bien les formules de Gram-Schmidt, on voit qu'on a besoin

- du produit scalaire
- de la soustraction dans E

- de la multiplication d'un vecteur de E par un réel

Nous allons donc représenter E par un triplet `(ps, sub, mul)` de trois fonctions. L'écriture de l'algorithme est alors immédiate.

[2]:
```
def gram_schmidt(B, E):
    ps, sub, mul = E
    n = len(B)
    B1 = [B[0]]
    for k in range(1, n):
        u = B[k]
        for j in range(k):
            mu = ps(B[k], B1[j]) / ps(B1[j], B1[j])
            u = sub(u, mul(mu, B1[j]))
        B1.append(u)
    return B1
```

Et voici une fonction qui renvoie `True` lorsque la base $\mathcal{B}$ est orthogonale.

[3]:
```
def est_orthogonale(B, E):
    n = len(B)
    ps = E[0]
    for i in range(n):
        for j in range(n):
            if j != i and ps(B[i], B[j]) != 0: return False
    return True
```

3.0.2 2.2 Orthonormalisation

Si l'on veut une base orthonormée, il suffit évidemment d'appliquer Gram-Schmidt puis de diviser les vecteurs de la base obtenue par leur norme.

La fonction `normaliser` prend en paramètre un vecteur u et renvoie $u/||u||$.

[4]:
```
def normaliser(u, E):
    ps, sub, mul = E
    return mul(1 / sqrt(ps(u, u)), u)
```

La fonction `orthonormer` prend en paramètre une base orthogonale $\mathcal{B}$. Elle renvoie une base orthonormée.

[5]:
```
def orthonormer(B, E):
    return [normaliser(u, E) for u in B]
```

3.0.3 2.3 Complexité

Nous appelons dans ce qui suit « opération » toute opération sur des réels (addition, multiplication, etc.). Notons

- $C_p(E)$ le nombre d'opérations nécéssaires pour effectuer le produit scalaire de deux vecteurs de E.
- $C_s(E)$ le nombre d'opérations nécéssaires pour effectuer le produit d'un vecteur de E par un réel.
- $C_m(E)$ le nombre d'opérations nécéssaires pour effectuer la soustraction de deux vecteurs de E.

La fonction `gram_schmidt` présente deux boucles imbriquées. La boucle indexée par k effectue $n-1$ itérations, où n est la dimension de E. Pour chaque k, la boucle indexée par j effectue k itérations. Le nombre total d'itérations est donc

$$\sum_{k=1}^{n-1} k = \frac{1}{2}n(n-1)$$

À chaque itération la fonction effectue :

- 2 produits scalaires (on pourrait faire mieux en mémorisant certains résultats)
- 1 soustraction
- 1 produit par un scalaire
- 1 division

Tout cela ne dépend ni de j ni de k. Si l'on note, avec un peu de redondance, $C_n(E)$ la complexité de l'algorithme de Schmidt, on a donc

$$C_n(E) = \frac{1}{2}n(n-1)(2C_p(E) + C_s(E) + C_m(E) + 1)$$

On ne peut pas aller plus loin dans le cas général : tout dépend de l'espace E sur lequel on travaille. Regardons deux exemples.

4 3. Exemples

4.0.1 3.1 $\mathbb{R}^n$

Munissons $\mathbb{R}^n$ du produit scalaire canonique. Comme `sympy` sait faire des produits scalaires, soustraire des vecteurs et les multiplier par un réel, il n'y a rien à définir.

Voici $\mathbb{R}^n$:

[6]:
```
E1 = (
    lambda u, v: u.dot(v),
    lambda u, v: u - v,
    lambda t, u: t * u)
```

Faisons quelques tests.

[7]:
```
ps1 = E1[0]
ps1(Matrix([1,2,3]), Matrix([4,5,6]))
```

[7]:
32

[8]:
```
sub1 = E1[1]
sub1(Matrix([1, 2, 3]), Matrix([2, 1, 4]))
```

[8]:
$$\begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$$

Considérons la base $((1,2,3),(4,5,6),(7,8,8))$.

[9]:
```
B1 = [Matrix([1, 2, 3]), Matrix([4, 5, 6]), Matrix([7, 8, 8])]
B1
```

[9]:
$$\left[\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, \begin{bmatrix} 7 \\ 8 \\ 8 \end{bmatrix}\right]$$

Orthogonalisons.

[10]:
```
B2 = gram_schmidt(B1, E1)
B2
```

[10]:
$$\left[\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} \frac{12}{7} \\ \frac{3}{7} \\ -\frac{6}{7} \end{bmatrix}, \begin{bmatrix} -\frac{1}{6} \\ \frac{1}{3} \\ -\frac{1}{6} \end{bmatrix}\right]$$

La base est-elle bien orthogonale ?

[11]:
```
est_orthogonale(B2, E1)
```

[11]: True

Tant que nous y sommes, orthonormalisons.

[12]:
```
orthonormer(B2, E1)
```

[12]:
$$\left[\begin{bmatrix} \frac{\sqrt{14}}{14} \\ \frac{\sqrt{14}}{7} \\ \frac{3\sqrt{14}}{14} \end{bmatrix}, \begin{bmatrix} \frac{4\sqrt{21}}{21} \\ \frac{\sqrt{21}}{21} \\ -\frac{2\sqrt{21}}{21} \end{bmatrix}, \begin{bmatrix} -\frac{\sqrt{6}}{6} \\ \frac{\sqrt{6}}{3} \\ -\frac{\sqrt{6}}{6} \end{bmatrix}\right]$$

Quelle est la complexité de l'algorithme ?

- Pour effectuer un produit scalaire dans $\mathbb{R}^n$, on fait n multiplications de réels et $n-1$ additions de réels. Donc, $C_p = 2n-1$.
- $C_s = n$ et $C_m = n$, de façon évidente.

Ainsi,

$$2C_p + C_s + C_m + 1 = 4n - 2 + n + n + 1 = 6n - 1$$

et

$$C(n) = \frac{1}{2}n(n-1)(6n-1)$$

Ainsi,

$$C(n) \sim 3n^3$$

4.0.2 3.2 $\mathbb{R}_n[X]$

Il existe beaucoup de produits scalaires intéressants sur l'espace $\mathbb{R}[X]$. Nous allons choisir celui défini par

$$< P, Q >= \int_{-1}^{1} P(t)Q(t)\,dt$$

[13]:
```
X = Symbol('X')
```

[14]:
```
E2 = (
    lambda P, Q: integrate(P * Q, (X, -1, 1)),
    lambda P, Q: P - Q,
    lambda t, P: t * P
)
```

Prenons la base canonique de $\mathbb{R}_4[X]$.

[15]:
```
B3 = [1, X, X ** 2, X ** 3, X ** 4]
```

[16]:
```
B4 = gram_schmidt(B3, E2)
B4
```

[16]:
$$\left[1,\ X,\ X^2 - \frac{1}{3},\ X^3 - \frac{3X}{5},\ X^4 - \frac{6X^2}{7} + \frac{3}{35}\right]$$

[17]:
```
est_orthogonale(B4, E2)
```

[17]: True

Et orthonormalisons.

[18]:
```
orthonormer(B4, E2)
```

[18]:
$$\left[\frac{\sqrt{2}}{2},\ \frac{\sqrt{6}X}{2},\ \frac{3\sqrt{10}\left(X^2 - \frac{1}{3}\right)}{4},\ \frac{5\sqrt{14}\left(X^3 - \frac{3X}{5}\right)}{4},\ \frac{105\sqrt{2}\left(X^4 - \frac{6X^2}{7} + \frac{3}{35}\right)}{16}\right]$$

[19]:
```
list(map(simplify, _))
```

[19]:

$$\left[\frac{\sqrt{2}}{2},\ \frac{\sqrt{6}X}{2},\ \frac{\sqrt{10}\left(3X^2-1\right)}{4},\ \frac{\sqrt{14}X\left(5X^2-3\right)}{4},\ \frac{3\sqrt{2}\left(35X^4-30X^2+3\right)}{16}\right]$$

Les polynômes que nous obtenons font partie d'une famille importante, ils sont appelés les **polynômes de Legendre**.

Quelle est ici la complexité de gram_schmidt ? Plaçons nous dans $\mathbb{R}_n[X]$. On a $C_s = C_m = n+1$ puisque soustraction et produit par un réel se font par des soustractions ou multiplications coefficient par coefficient.

Concernant C_p, je n'ai évidemment pas la moindre idée de la complexité de la fonction d'intégration de sympy. Cela dit, si l'on code soi-même les calculs d'intégrales, il faut, pour calculer $\int_{-1}^{1} PQ$:

- Multiplier P et Q : cela donne, si on utilise la définition du produit de deux polynômes, $O(n^2)$ opérations élémentaires. On peut calculer précisément cette complexité, mais n'entrons pas dans les détails.
- Intégrer terme à terme : $O(n)$ opérations.

Ainsi, $C_p = O(n^2)$, donc

$$2C_p + C_s + C_m + 1 = O(n^2)$$

et

$$C(n) = O(n^4)$$