Images_Noyaux

March 18, 2024

1 Espaces engendrés, rangs, images, noyaux

Marc Lorenzi

16 mars 2024

[1]:
```
import random
```

Étant donnée une famille de vecteurs, quel est l'espace engendré par cette famille ? Étant donnée une application linéaire, que sont son noyau et son image ? Le point commun des réponses à ces questions est que l'on cherche à déterminer des *sous-espaces vectoriels* d'un espace vectoriel donné. Il existe essentiellement deux façons de se donner un sous-espace vectoriel :

- Par une base
- Par des équations cartésiennes

Nous allons dans ce notebook parler surtout de la vision « base ». Nous parlerons à la fin de l'aspect « équation cartésienne ».

Afin de pouvoir calculer de façon exacte, nous nous placerons dans des $\mathbb{Q}$-espaces vectoriels. La classe `Fraction` implémente les rationnels en Python.

[2]:
```
from fractions import Fraction
```

1.1 1. Quelques fonctions utiles

1.1.1 1.1 La classe `Matrice`

Nous représenterons une matrice $A \in \mathcal{M}_{pq}(\mathbb{Q})$ en Python par un objet de la classe `Matrice` définie ci-dessous. L'appel au constructeur `Matrice(p, q)` crée une matrice ayant p lignes et q colonnes. Si A est une matrice, le coefficient de A, ligne i, colonne j, est $A[i][j]$.

[3]:
```
class Matrice:

    def __init__(self, p, q, mat=None):
        self.nblig = p
        self.nbcol = q
        if mat != None:
            self.mat = mat
```

```
        else:
            self.mat = p * [None]
            for i in range(p):
                self.mat[i] = q * [0]

    def taille(self):
        return (self.nblig, self.nbcol)

    def __getitem__(self, i): return self.mat[i]
```

La fonction `transposée` renvoie la transposée de son paramètre A.

[4]:
```
def transposee(A):
    p, q = A.taille()
    B = Matrice(q, p)
    for i in range(p):
        for j in range(q):
            B[j][i] = A[i][j]
    return B
```

1.1.2 1.2 Copier une matrice

Nous allons dans ce notebook écrire des fonctions qui modifient le contenu d'une matrice. Afin de ne pas perdre la latrice originale, il est judicieux de travailler sur une *copie* de celle-ci. La fonction `copie` prend en paramètre une matrice A et renvoie une copie de A.

[5]:
```
def copie(A):
    p, q = A.taille()
    B = Matrice(p, q)
    for i in range(p):
        for j in range(q):
            B[i][j] = A[i][j]
    return B
```

1.1.3 1.3 Afficher des matrices

La fonction `print_mat` affiche proprement la matrice A.

[6]:
```
def print_mat(A):
    p, q = A.taille()
    if p == 0: return
    else:
        ws = q * [0]
        for i in range(p):
            for j in range(q):
                w = len(str(A[i][j]))
```

```
                if w > ws[j]: ws[j] = w
        for i in range(p):
            for j in range(q):
                print(('%' + str(ws[j]) +'s ') % A[i][j], end='')
            print()
```

[7]:
```
A = Matrice(3, 6)
print_mat(A)
```

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

[8]:
```
print_mat(transposee(A))
```

```
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
```

1.1.4 1.4 Vecteurs, familles de vecteurs

Un vecteur de $\mathbb{Q}^p$ est un p-uplet de rationnels. Nous le représenterons en Python par une liste de p rationnels.

Soit $\mathcal{F} = (u_0, \dots, u_{q-1})$ une famille de q vecteurs de $\mathbb{Q}^p$.

La matrice de la famille $\mathcal{F}$ (dans la base canonique de $\mathbb{Q}^p$) est la matrice A de taille $p \times q$ dont les **colonnes** sont les coordonnées des vecteurs de $\mathcal{F}$.

La fonction `famille` prend en paramètres deux entiers q et p, ainsi, qu'une liste q vecteurs de $\mathbb{Q}^p$ censée représenter une famille de vecteurs. Elle renvoie la matrice de cette famille.

[9]:
```
def mat_famille(q, p, V):
    A = Matrice(q, p, V)
    return transposee(A)
```

Par exemple, soit $\mathcal{F} = ((1,2,3,4),(5,6,7,8),(9,10,11,12))$. La matrice de $\mathcal{F}$ est

[10]:
```
V = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
A = mat_famille(3, 4, V)
print_mat(A)
```

```
1 5  9
2 6 10
3 7 11
4 8 12
```

Remarque. Le lecteur pourra se demander pourquoi faire passer en paramètre à la fonction `mat_famille` les entiers p et q. Après tout, ne peut on pas récupérer p et q si on connaît V ? En réalité, pas tout à fait. Si la liste V n'est pas vide, alors q est la longueur de V et p est la longueur de $V[0]$. En revanche, si V est vide alors $q = 0$ mais que vaut p ? Il est dans ce cas impossible de retrouver la valeur de p. Nous savons que nous avons une famile de 0 vecteur mais nous ne savons pas dans quel espace vectoriel $\mathbb{Q}^p$ nous nous trouvons.

1.2 2. Opérations élémentaires sur les colonnes

Étant donnée une famille $\mathcal{F}$ de vecteurs, il existe trois opérations qui ne changent pas l'espace engendré par la famille. Nous allons les examiner une par une. Les vecteurs sont rangés dans les colonnes d'une matrice, nous parlerons donc **d'opérations sur les colonnes**.

1.2.1 2.1 Multiplier une colonne par un nombre

La fonction `mult` prend en paramètres

- Une matrice A
- un entier j
- Un rationnel non nul t

Elle multiplie la colonne j de A par t.

[11]:
```
def mult(A, j, t):
    p, q = A.taille()
    for i in range(p):
        A[i][j] = t * A[i][j]
```

1.2.2 2.2 Échanger deux colonnes

La fonction `ech` prend en paramètres

- Une matrice A
- Deux entiers j_1 et j_2

Elle échange les colonnes j_1 et j_2 de A.

[12]:
```
def ech(A, j1, j2):
    p, q = A.taille()
    for i in range(p):
        A[i][j1], A[i][j2] = A[i][j2], A[i][j1]
```

1.2.3 2.3 Ajouter à une colonne un multiple d'une autre colonne

La fonction `add_mult` prend en paramètres

- Une matrice A

- Un entier j_1
- Un rationnel t
- Un entier j_2

Elle ajoute à la colonne j_1 de A t fois la colonne j_2 de A..

[13]:
```
def add_mult(A, j1, t, j2):
    p, q = A.taille()
    for i in range(p):
        A[i][j1] = A[i][j1] + t *  A[i][j2]
```

1.3 3. Combiner le tout

1.3.1 3.1 Historiques

Étant donnée une famille $\mathcal{F} = (u_0, \dots, u_{q-1})$ de vecteurs, la recherche de l'espace engendré par $\mathcal{F}$, de son rang (et encore d'autres choses) se fait en effectuant des opérations élémentaires sur les colonnes de la matrice de cette famille. Il peut être intéressant de conserver un historique de ces opérations. Après un certain nombre d'opérations, nous saurons, grâce au contenu de l'historique, quelles sont, en fonction de $u_0, \dots, u_{q-1}$, les colonnes de notre matrice.

L'historique est une matrice H de taille $q \times q$. Après avoit effectué un certain nombre d'opérations élémentaires à partir de la famille $\mathcal{F}$, on obtient une famille $\mathcal{F}' = (u'_0, \dots, u'_{q-1})$. Pour tout $j \in [|0, q-1|]$,

$$u'_j = \sum_{i=0}^{q-1} H_{ij} u_i$$

Avant toute opération, notre historique est la matrice identité I_q. La fonction `hist0` prend un entier q en paramètre et renvoie la matrice I_q.

[14]:
```
def hist0(q):
    hist = Matrice(q, q)
    for i in range(q):
        hist[i][i] = 1
    return hist
```

Par exemple, si $q = 4$:

[15]:
```
hist = hist0(4)
print_mat(hist)
```

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

À chaque opération sur les colonnes d'une matrice A (censée être la matrice d'une famille de vecteurs $\mathcal{F} = (u_0, \dots, u_{q-1})$), nous effectuerons la même opération sur l'historique. Par exemple, si notre première opération est d'ajouter à la colonne 0 de A 3 fois la colonne 2, nous faisons aussi

[16]:
```
add_mult(hist, 0, 3, 2)
print_mat(hist)
```

```
1 0 0 0
0 1 0 0
3 0 1 0
0 0 0 1
```

En regardant la colonne 0 de l'historique nous voyons que la colonne 0 de A contient maintenant $u_0 + 3u_2$.

1.3.2 3.2 Pivoter

La fonction `pivoter` prend en paramètres

- Une matrice A
- Deux entiers i et j.
- Un historique `hist`.

En notant $t = \frac{1}{A_{ij}}$, elle effectue les opérations

- $C_j \longleftarrow tC_j$
- Pour tout $j_1 \in [|0, q-1|] \quad \{j\}$, $C_{j_1} \longleftarrow C_{j_1} - A_{ij_1}C_j$.

L'effet de cette fonction est d'annuler tous les coefficients de la ligne i de A, sauf celui situé en colonne j, qui devient égal à 1. Les colonnes de la « nouvelle » matrice A engendrent le même espace que les colonnes de l'ancienne matrice A.

[17]:
```
def pivoter(A, i, j, hist):
    p, q = A.taille()
    t = Fraction(1, A[i][j])
    mult(A, j, t)
    mult(hist, j, t)
    for j1 in range(q):
        if j1 != j:
            t = -A[i][j1]
            add_mult(A, j1, t, j)
            add_mult(hist, j1, t, j)
```

1.3.3 3.2 Exemples

Nous allons dans la suite travailler sur trois familles de vecteurs.

[18]:
```
```

```
F1 = mat_famille(3, 3, [[1, 2, 3], [4, 5, 6], [7, 8, 9]])
F2 = mat_famille(4, 5, [[1, 2, -4, 3, 1], [2, 5, -3, 4, 8], [6, 17, -7, 10, 22],␣
 ↪[1, 3, -3, 2, 0]])
F3 = mat_famille(6, 6, [[1, 1, 1, 2, 2, 2], [2, 2, 2, 3, 3, 3], [3, 3, 3, 5, 5,␣
 ↪5], [1, 2, 3, 2, 3, 4], [2, 3, 4, 3, 4, 5],
      [3, 5, 7, 5, 7, 9]])
```

Le premier exemple est un exemple facile.

[19]:
```
A = copie(F1)
print_mat(A)
```

```
1 4 7
2 5 8
3 6 9
```

Initialisons aussi un historique.

[20]:
```
p, q = A.taille()
hist = hist0(q)
```

Pivotons sur le 1, situé ligne 0, colonne 0.

[21]:
```
pivoter(A, 0, 0, hist)
print_mat(A)
```

```
1  0   0
2 -3  -6
3 -6 -12
```

Pivotons ensuite sur le -3 situé ligne 1, colonne 1.

[22]:
```
pivoter(A, 1, 1, hist)
print_mat(A)
```

```
 1 0 0
 0 1 0
-1 2 0
```

Qu'avons-nous dans l'historique ?

[23]:
```
print_mat(hist)
```

```
-5/3  4/3  1
 2/3 -1/3 -2
   0    0  1
```

Si nous appelons e_0, e_1, e_2 les trois vecteurs de la famille $\mathcal{F}_1$, l'historique nous dit que

$$-\frac{5}{3}e_0 + \frac{2}{3}e_1 = (1, 0, -1)$$

$$\frac{4}{3}e_0 - \frac{1}{3}e_1 = (0, 1, 2)$$

Et surtout :

$$e_0 - 2e_1 + e_2 = 0$$

Passons au second exemple, un peu plus difficile que le premier.

[24]:
```
A = copie(F2)
p, q = A.taille()
hist = hist0(q)
print_mat(A)
```

```
 1  2  6  1
 2  5 17  3
-4 -3 -7 -3
 3  4 10  2
 1  8 22  0
```

Pivotons ligne 0, colonne 0.

[25]:
```
pivoter(A, 0, 0, hist)
print_mat(A)
```

```
 1  0  0  0
 2  1  5  1
-4  5 17  1
 3 -2 -8 -1
 1  6 16 -1
```

Pivotons ensuite ligne 1, colonne 1.

[26]:
```
pivoter(A, 1, 1, hist)
print_mat(A)
```

```
  1  0   0  0
  0  1   0  0
-14  5  -8 -4
  7 -2   2  1
-11  6 -14 -7
```

Et enfin, pivotons ligne 2, colonne 2.

[27]:
```
pivoter(A, 2, 2, hist)
print_mat(A)
```

```
  1    0    0 0
  0    1    0 0
  0    0    1 0
```

```
 7/2  -3/4 -1/4 0
27/2 -11/4  7/4 0
```

Regardons l'historique.

[28]:
```
print_mat(hist)
```

```
  -2   1/2 -1/2   -1
27/4 -17/8  5/8  3/2
-7/4   5/8 -1/8 -1/2
   0     0    0    1
```

La dernière colonne de l'historique nous donne une combinaison linéaire non triviale des vecteurs de $\mathcal{F}_2$:

$$-e_0 + \frac{3}{2}e_1 - \frac{1}{2}e_2 + e_3 = 0$$

Passons à l'exemple 3, un peu plus subtil.

[29]:
```
A = copie(F3)
p, q = A.taille()
hist = hist0(q)
print_mat(A)
```

```
1 2 3 1 2 3
1 2 3 2 3 5
1 2 3 3 4 7
2 3 5 2 3 5
2 3 5 3 4 7
2 3 5 4 5 9
```

Pivotons d'abord ligne 0, colonne 0

[30]:
```
pivoter(A, 0, 0, hist)
print_mat(A)
```

```
1  0  0 0  0  0
1  0  0 1  1  2
1  0  0 2  2  4
2 -1 -1 0 -1 -1
2 -1 -1 1  0  1
2 -1 -1 2  1  3
```

Un échange des colonnes 1 et 3 paraît judicieux. N'oublions pas d'échanger aussi les colonnes de l'historique.

[31]:
```
ech(A, 1, 3)
ech(hist, 1, 3)
print_mat(A)
```

```
1 0  0  0  0  0
1 1  0  0  1  2
1 2  0  0  2  4
2 0 -1 -1 -1 -1
2 1 -1 -1  0  1
2 2 -1 -1  1  3
```

Pivotons ligne 1, colonne 1.

[32]:
```
pivoter(A, 1, 1, hist)
print_mat(A)
```

```
 1 0  0  0  0  0
 0 1  0  0  0  0
-1 2  0  0  0  0
 2 0 -1 -1 -1 -1
 1 1 -1 -1 -1 -1
 0 2 -1 -1 -1 -1
```

Encore un « pivotage » ligne 3, colonne 2 …

[33]:
```
pivoter(A, 3, 2, hist)
print_mat(A)
```

```
 1 0 0 0 0 0
 0 1 0 0 0 0
-1 2 0 0 0 0
 0 0 1 0 0 0
-1 1 1 0 0 0
-2 2 1 0 0 0
```

[34]:
```
print_mat(hist)
```

```
-4 -1  3  1  2  2
 0  0  0  1  0  0
 2  0 -1 -1 -1 -1
-1  1  0  0 -1 -2
 0  0  0  0  1  0
 0  0  0  0  0  1
```

Comme les trois dernières colonnes de notre matrice finale A sont nulles, les trois dernières colonnes de l'historique nous donnent des combinaisons linéaires nulles non triviales des vecteurs de $\mathcal{F}_3$.

$$\begin{cases} e_0 + e_1 - e_2 & = & 0 \\ 2e_0 - e_2 - e_3 + e_4 & = & 0 \\ 2e_0 - e_2 - 2e_3 + e_5 & = & 0 \end{cases}$$

1.3.4 3.3 Automatiser

Bien entendu, les opérations que nous avons effectuées manuellement en « regardant » les matrices pour trouver quoi faire et où pivoter peuvent être automatisées.

La fonction `trouver_pivot` prend en paramètres une matrice A et deux entiers i et j. Elle renvoie un couple (u, v) tel que $u \geq i$, $v \geq j$ et $A_{uv} \neq 0$. Plus précisément, elle renvoie `(True, (u,v))` si ce couple existe et `(False, None)` sinon.

Le couple (u, v) a en plus la propriété que u est minimal.

[35]:
```
def trouver_pivot(A, i, j):
    p, q = A.taille()
    for u in range(i, p):
        for v in range (j, q):
            if A[u][v] != 0:
                return (True, (u, v))
    return False, None
```

La fonction `reduire` automatise tout le travail. Elle prend une matrice A et un historique en paramètres et effectue des pivots sur A tant que cela est possible.

[36]:
```
def reduire(A, hist):
    p, q = A.taille()
    i, j = -1, -1
    while True:
        b, c = trouver_pivot(A, i + 1, j + 1)
        if not b: return
        else:
            u, v = c
            if v != j + 1:
                ech(A, j + 1, v)
                ech(hist, j + 1, v)
            i, j =  u, j + 1
            pivoter(A, i, j, hist)
```

Testons sur nos exemples.

[37]:
```
A = copie(F1)
p, q = A.taille()
hist = hist0(q)
reduire(A, hist)
print_mat(A)
print()
print_mat(hist)
```

```
 1 0 0
 0 1 0
-1 2 0
```

```
-5/3  4/3  1
 2/3 -1/3 -2
   0    0  1
```

[38]:
```
A = copie(F2)
p, q = A.taille()
hist = hist0(q)
reduire(A, hist)
print_mat(A)
print()
print_mat(hist)
```

```
   1     0    0 0
   0     1    0 0
   0     0    1 0
 7/2  -3/4 -1/4 0
27/2 -11/4  7/4 0

  -2   1/2 -1/2   -1
27/4 -17/8  5/8  3/2
-7/4   5/8 -1/8 -1/2
   0     0    0    1
```

[39]:
```
A = copie(F3)
p, q = A.taille()
hist = hist0(q)
reduire(A, hist)
print_mat(A)
print()
print_mat(hist)
```

```
 1 0 0 0 0 0
 0 1 0 0 0 0
-1 2 0 0 0 0
 0 0 1 0 0 0
-1 1 1 0 0 0
-2 2 1 0 0 0

-4 -1  3  1  2  2
 0  0  0  1  0  0
 2  0 -1 -1 -1 -1
-1  1  0  0 -1 -2
 0  0  0  0  1  0
 0  0  0  0  0  1
```

1.3.5 3.4 Faire disparaître les fractions

Ce qui nous intéresse dans les matrices que nous manipulons ici c'est l'espace engendré par leurs colonnes. Multiplier une colonne de la matrice par un rationnel non nul ne change pas l'espace engendré. Il est donc facile, une fois terminée l'opération de réduction, de faire disparaître les fractions. Il suffit de multiplier chaque colonne par le ppcm des dénominateurs de ses coefficients.

La fonction `pgcd` renvoie le pgcd des entiers a et b. Elle utilise l'algorithme d'Euclide.

[40]:
```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

La fonction `ppcm` renvoie le ppcm des entiers a et b. Elle utilise l'égalité $\text{pgcd}(a,b)\times\text{ppcm}(a,b) = ab$, en examinant à part le cas $a = b = 0$.

[41]:
```
def ppcm(a, b):
    if a == 0 and b == 0:
        return 0
    else:
        return a * (b // pgcd(a, b))
```

La fonction `ppcm_liste` prend en paramètre une liste xs d'entiers. Elle renvoie le ppcm de ces entiers.

[42]:
```
def ppcm_liste(xs):
    m = 1
    for x in xs:
        m = ppcm(m, x)
    return m
```

[43]:
```
ppcm_liste([3, 4, 9, 6])
```

[43]:
```
36
```

Tant que nous y sommes, écrivons une fonction `pgcd_liste` qui prend en paramètre une liste xs d'entiers et renvoie le pgcd de ces entiers. Elle nous servira à la fin du notebook.

[44]:
```
def pgcd_liste(xs):
    d = 0
    for x in xs:
        d = pgcd(d, x)
    return d
```

[45]:
```
pgcd_liste([2, 4, 0, 6])
```

[45]:
```
2
```

La fonction `simplifier_frac` prend en paramètres une matrice et un historique. Elle élimine les fractions. Les matrices obtenues à l'issue du calcul contiennent des entiers (c'est à dire des objets Python de type `int`).

[46]:
```
def simplifier_frac(A, hist):
    p, q = A.taille()
    for j in range(q):
        xs = [A[i][j].denominator for i in range(p)]
        ys = [hist[i][j].denominator for i in range(q)]
        m = ppcm_liste(xs + ys)
        mult(A, j, m)
        mult(hist, j, m)
    for i in range(p):
        for j in range(q):
            A[i][j] = A[i][j].numerator
    for i in range(q):
        for j in range(q):
            hist[i][j] = hist[i][j].numerator
```

Pour terminer, voici une fonction `reduire_simplifier` qui fait ce que l'on imagine.

[47]:
```
def reduire_simplifier(A, hist):
    reduire(A, hist)
    simplifier_frac(A, hist)
```

Prenons l'exemple de la famille $\mathcal{F}_2 = (e_0, e_1, e_2, e_3)$ de 4 vecteurs de $\mathbb{Q}^5$.

[48]:
```
A = copie(F2)
p, q = A.taille()
hist = hist0(q)
reduire_simplifier(A, hist)
print_mat(A)
print()
print_mat(hist)
```

```
 4   0  0 0
 0   8  0 0
 0   0  8 0
14  -6 -2 0
54 -22 14 0

-8   4 -4 -2
27 -17  5  3
-7   5 -1 -1
 0   0  0  2
```

Nous avons donc

$$\begin{cases} -8e_1 + 27e_2 - 7e_3 & = & (4,0,0,14,54) \\ 4e_1 - 17e_2 + 5e_3 & = & (0,8,0,-6,-22) \\ -4e_1 + 5e_2 - e_3 & = & (0,0,8,-2,14) \\ -2e_1 + 3e_2 - e_3 + 2e_4 & = & (0,0,0,0,0) \end{cases}$$

La dernière égalité nous donne une combinaison nulle non triviale à coefficients entiers des vecteurs de la famille.

1.4 4. Espace engendré, rang

1.4.1 4.1 Introduction

Soit $\mathcal{F} = (u_0, \ldots, u_{q-1})$ une famille de q vecteurs $\mathbb{Q}^p$. Soit A la matrice de $\mathcal{F}$ dans la base canonique de $\mathbb{Q}^p$. La réduction de A effectuée par la fonction `reduire` (ou `reduire_simplifier`) fournit une matrice A'. Les colonnes non nulles de A' sont les matrices dans la base canonique de $\mathbb{Q}^p$ d'une famille $\mathcal{F}' = (v_0, \ldots, v_{r-1})$ de vecteurs de $\mathbb{Q}^p$. Comme les opérations effectuées à partir de A pour obtenir A' ne changent pas les espaces engendrés, nous avons le résultat suivant.

- La famille $\mathcal{F}'$ est libre.
- $< \mathcal{F}' > = < \mathcal{F} >$.
- $r = \text{rg}(\mathcal{F})$.

1.4.2 4.2 Le code

La fonction `est_nul` prend en paramètre un vecteur $u \in \mathbb{Q}^n$ représenté par une liste de rationnels. Elle renvoie `True` si $u = 0$ et `False` sinon.

[49]:
```
def est_nul(u):
    n = len(u)
    for i in range(n):
        if u[i] != 0: return False
    return True
```

La fonction `base_espace_engendre` prend en paramètre une famille $\mathcal{F}$ de vecteurs de $\mathbb{Q}^q$. et renvoie la matrice d'une famille $\mathcal{F}'$ qui est une base de $< \mathcal{F} >$. En bref, cette fonction supprime les colonnes nulles de la matrice réduite.

Nous sommes ici confrontés au problème inverse de celui de l'élimination des dénominateurs de fractions. Pour avoir une réponse la plus simple possible, écrivons une fonction qui **divise** chaque colonne d'une matrice à coefficients entiers par le **pgcd** des coefficients de la colonne.

[50]:
```
def eliminer_facteurs(A):
    p, q = A.taille()
    for j in range(q):
        xs = [A[i][j] for i in range(p)]
        m = pgcd_liste(xs)
        for i in range(p):
```

```
            A[i][j] = A[i][j] // m
```

[51]:
```
def base_espace_engendre(F):
    A = copie(F)
    p, q = A.taille()
    hist = hist0(q)
    reduire_simplifier(A, hist)
    A1 = transposee(A)
    V = []
    for u in A1:
        if not est_nul(u):
            V.append(u)
    B = transposee(Matrice(len(V), p, V))
    eliminer_facteurs(B)
    return B
```

[52]:
```
A = copie(F1)
print()
print_mat(base_espace_engendre(A))
```

```
-1 0
 0 1
 1 2
```

[53]:
```
A = copie(F2)
print()
print_mat(base_espace_engendre(A))
```

```
 2  0  0
 0 -4  0
 0  0  4
 7  3 -1
27 11  7
```

[54]:
```
A = copie(F3)
print()
print_mat(base_espace_engendre(A))
```

```
-1 0 0
 0 1 0
 1 2 0
 0 0 1
 1 1 1
 2 2 1
```

Le rang d'une famille de vecteurs est la dimension de l'espace engendré par cette famille. La fonction `rang` est donc immédiate à écrire.

[55]:
```
def rang(F):
    B = base_espace_engendre(F)
    p, r = B.taille()
    return r
```

[56]:
```
print(rang(F1), rang(F2), rang(F3))
```

```
2 3 3
```

Nous en déduisons gratuitement une fonction qui détermine si une famille est libre. En effet, une famille de q vecteurs de $\mathbb{Q}^p$ est libre si et seulement si son rang est égal à q.

[57]:
```
def est_libre(F):
    p, q = F.taille()
    return rang(F) == q
```

Et une fonction qui détermine si une famille est génératrice. En effet, une famille de q vecteurs de $\mathbb{Q}^p$ est génératrice si et seulement si son rang est égal à p.

[58]:
```
def est_generatrice(F):
    p, q = F.taille()
    return rang(F) == p
```

[59]:
```
print(est_libre(F2))
print(est_generatrice(F2))
```

```
False
False
```

1.4.3 4.3 Familles de vecteurs « aléatoires »

Nos familles d'exemples ne sont ni libres ni génératrices. Pour tester nos fonctions `est_libre` et `est_generatrice`, créons des familles « aléatoires ».

La fonction `rationnel_aleatoire` prend en paramètre deux entiers $M \in \mathbb{N}$ et $N \in \mathbb{N}^*$. Elle renvoie un rationnel a/b où $a \in [\![-M, M]\!]$ et $b \in [\![1, N]\!]$ sont deux entiers aléatoires.

[60]:
```
def rationnel_aleatoire(M, N):
    a = random.randint(-M, M)
    b = random.randint(1, N)
    return Fraction(a, b)
```

[61]:
```
print(rationnel_aleatoire(99, 99))
```

```
-39/31
```

La fonction `famille_aleatoire` renvoie une famille de q vecteurs aléatoires de $\mathbb{Q}^p$.

[62]:
```
def famille_aleatoire(p, q, M=99, N=99):
    V = []
    for j in range(q):
        u = [rationnel_aleatoire(M, N) for k in range(p)]
        V.append(u)
    return mat_famille(q, p, V)
```

[63]:
```
F = famille_aleatoire(10, 10)
print_mat(F)
```

```
 64/67 -23/16 -43/20  11/91  -24/7  12/85  -43/6  -27/2   4/53  27/44
 -3/23 -57/13  12/73   9/17 -37/16   7/51  -2/89  10/17   32/3 -51/28
-14/31  10/89   -5/6      9  11/95  85/93  69/70  32/17   19/8 -14/11
 -3/23  19/54 -65/69 -17/37 -84/95   45/2  20/57 -17/40   3/10  50/57
 -3/17     19 -78/73  -73/7  -6/23  -5/51 -15/31 -29/13 -94/27    7/5
 19/24 -88/83  78/41      1   -7/3 -79/48   -5/8  28/81 -79/80  73/88
-73/90 -17/28   45/7 -39/22  59/91  -31/9 -49/99  95/77 -36/49 -71/10
 22/69  97/36 -61/25   9/44  17/13 -25/79      1 -39/10  16/25  64/93
 55/62  81/20 -43/79  13/16 -82/63  16/59   5/13 -21/16  39/50  18/19
-42/43  18/11  40/37 -17/12  85/39  -5/78 -15/37 -29/34   43/6  78/61
```

Une famille de 10 vecteurs de $\mathbb{Q}^{10}$ a de grandes chances d'être une base.

[64]:
```
print(est_libre(F))
print(est_generatrice(F))
```

```
True
True
```

Prenons maintenant une famille de 10 vecteurs de $\mathbb{Q}^8$. Elle est forcément liée et elle a de grandes chances d'être génératrice.

[65]:
```
F = famille_aleatoire(8, 10)
print_mat(F)
print()
print(est_libre(F))
print(est_generatrice(F))
```

```
 27/29   15/4 -20/23  95/16   9/46    -50  64/83   13/6  44/79      2
 56/69 -91/18 -89/24 -76/39  79/72  11/91    2/9     -3   -1/8 -16/47
-91/69  42/85  11/31    3/2  -7/17  31/57      5 -45/58  41/25 -29/12
     1   13/7 -16/17 -10/53 -35/61  39/76 -32/85  18/95 -88/63  -15/2
-57/52 -89/35  13/16   13/2  76/87 -62/67      7    -11  31/81   7/24
 92/79 -21/16 -23/38 -22/21  48/17 -13/50  95/87  21/16 -65/33  79/73
 45/43 -70/23  13/44 -19/34  81/35   19/6  83/59  98/89 -65/62  76/29
 28/81      0   3/10   19/8 -86/61 -41/15  -17/5 -33/38   2/15  73/21
```

```
False
True
```

Prenons enfin une famille de 8 vecteurs de $\mathbb{Q}^{10}$. Elle a de grandes chances d'être libre et elle ne peut pas être génératrice.

[66]:
```
F = famille_aleatoire(10, 8)
print_mat(F)
print()
print(est_libre(F))
print(est_generatrice(F))
```

```
 -89/3 -31/50 -74/19    2/3  -4/49  28/33   -3/2 -15/13
  40/9 -10/13      3  53/83  79/30  -9/77  68/89   4/17
-45/82  43/21   11/4   54/5 -33/74 -12/31 -46/55  -5/74
  2/29 -56/97 -24/97 -55/14  -2/73 -35/19 -14/47 -80/69
    17  91/31  -5/73  24/11 -53/66  -37/5     12  82/39
   1/2  75/86  57/28 -50/19  27/46  91/97  87/53 -16/19
-35/53    1/7 -27/89 -89/57  31/43   1/18 -74/59 -22/27
     3  -6/91 -25/32 -23/29  70/57 -44/59  25/61  -27/4
  8/53 -27/35 -31/49  17/27 -67/69 -41/95 -43/45  18/23
 31/78 -37/33   -7/6 -49/30  -6/17 -11/12 -14/11  25/14

True
False
```

1.5 5. Noyau et image d'une application linéaire

1.5.1 5.1 Introduction

Notons $\mathcal{B} = (e_0, \dots, e_{q-1})$ la base canonique de $\mathbb{Q}^q$ et $\mathcal{B}' = (e'_0, \dots, e'_{p-1})$ la base canonique de $\mathbb{Q}^p$.

Soit $f \in \mathcal{L}(\mathbb{Q}^q, \mathbb{Q}^p)$. Notons A la matrice de la famille $\mathcal{F} = f(\mathcal{B})$ (dans la base $\mathcal{B}'$).

L'image de f est

$$\mathrm{Im}(f) = f(\mathbb{Q}^q) = f(< \mathcal{B} >) =< f(\mathcal{B}) >=< \mathcal{F} >$$

La fonction `base_espace_engendre` fournit donc une base de $\mathrm{Im}(f)$.

1.5.2 5.2 Noyau

Comment obtenir une base de $\mathrm{Ker}(f)$? C'est facile. Soit r le rang de f. Après un appel à la fonction `reduire`, la matrice contient r colonnes non nulles, et donc $q - r$ colonnes nulles. Les vecteurs de l'historique correspondant à ces $q - r$ colonnes sont dans le noyau de f.

L'historique de départ contenait une famille de q vecteurs libres (en fait, la base canonique de $\mathbb{Q}^q$. Après réduction, l'historique n'a pas changé de rang, il contient donc toujours q vecteurs libres.

Les $q-r$ vecteurs correspondant aux colonnes nulles de A sont donc (sous-famille d'une famille libre) libres. Or, par le théorème du rang,

$$\dim(\text{Ker } f) = \dim(\mathbb{Q}^q) - \text{rg}(f) = q - r$$

Ces $q-r$ vecteurs forment donc une base de Ker f.

[67]:
```
def base_noyau(F):
    A = copie(F)
    p, q = A.taille()
    hist = hist0(q)
    reduire_simplifier(A, hist)
    A1 = transposee(A)
    hist = transposee(hist)
    V = []
    for k in range(q):
        if est_nul(A1[k]):
            V.append(hist[k])
    B = transposee(Matrice(len(V), q, V))
    eliminer_facteurs(B)
    return B
```

Prenons l'exemple 1. Notons $\mathcal{F}_1 = (u_0, u_1, u_2)$. Interprétons les u_i comme les images des vecteurs de la base canonique de $\mathbb{Q}^3$ par un endomorphisme f de $\mathbb{Q}^3$.

[68]:
```
print_mat(F1)
```

```
1 4 7
2 5 8
3 6 9
```

L'endomorphisme f est défini (théorème des 5 « de ») par

$$\begin{cases} f(e_0) &= (1,2,3) \\ f(e_1) &= (4,5,6) \\ f(e_2) &= (7,8,9) \end{cases}$$

f est de rang 2, son noyau est donc une droite.

[69]:
```
print_mat(base_noyau(F1))
```

```
 1
-2
 1
```

Ainsi, Ker$(f) =< (1,-2,1) >$.

Passons à l'exemple 2. Notons $\mathcal{F}_2 = (u_0, u_1, u_2, u_3)$. Interprétons les u_i comme les images des vecteurs de la base canonique de $\mathbb{Q}^5$ par $f \in \mathcal{L}(\mathbb{Q}^4, \mathbb{Q}^5)$.

[70]:
```
print_mat(F2)
```

```
 1  2  6  1
 2  5 17  3
-4 -3 -7 -3
 3  4 10  2
 1  8 22  0
```

L'application linéaire f est définie par

$$\begin{cases} f(e_0) &= (1,2,-4,3,1) \\ f(e_1) &= (2,5,-3,4,8) \\ f(e_2) &= (6,17,-7,10,22) \\ f(e_3) &= (1,3,-3,2,0) \end{cases}$$

f est de rang 3. Par le théorème du rang, son noyau est donc une droite.

[71]:
```
print_mat(base_noyau(F2))
```

```
-2
 3
-1
 2
```

Ainsi, $\text{Ker}(f) =< (-2,3,-1,2) >$.

Prenons l'exemple 3. Soit $\mathcal{F}_3 = (u_0, \dots, u_5)$. Interprétons les u_i comme les images des vecteurs de la base canonique de $\mathbb{Q}^6$ par $f \in \mathcal{L}(\mathbb{Q}^6)$.

[72]:
```
print_mat(F3)
```

```
1 2 3 1 2 3
1 2 3 2 3 5
1 2 3 3 4 7
2 3 5 2 3 5
2 3 5 3 4 7
2 3 5 4 5 9
```

L'application linéaire f est définie par

$$\begin{cases} f(e_0) &= (1,1,1,2,2,2) \\ f(e_1) &= (2,2,2,3,3,3) \\ f(e_2) &= (3,3,3,5,5,5) \\ f(e_3) &= (1,2,3,2,3,4) \\ f(e_4) &= (2,3,4,3,4,5) \\ f(e_4) &= (3,5,7,5,7,9) \end{cases}$$

f est de rang 3, son noyau est donc de dimension 3.

[73]:
```
print_mat(base_noyau(F3))
```

```
-1  2  2
-1  0  0
 1 -1 -1
 0 -1 -2
 0  1  0
 0  0  1
```

Ainsi, $\mathrm{Ker}(f) =< (-1,-1,1,0,0,0), (2,0,-1,-1,1,0), (2,0,-1,-2,0,1) >$.

1.5.3 5.3 Applications linéaires aléatoires

Prenons enfin quelques exemples aléatoires. Tout d'abord, un endomorphisme de $\mathbb{Q}^{10}$. Celui-ci a de grandes chances d'être un automorphisme.

[74]:
```
F = famille_aleatoire(10, 10, M=99, N=99)
print_mat(F)
```

```
-28/13   -1/2  -23/5  33/86   -9/2   8/7  21/74  31/22   16/9   1/72
  8/87   3/97   79/2  19/72   28/3 12/17  13/19 -20/99  31/17    1/3
 -10/9 -61/63   -5/3 -29/27      0 40/47  39/50  53/90 -49/57  32/35
-58/79 -86/79 -43/65  -10/3 -19/61 13/95   37/6 -82/59  79/19 -97/13
-70/61  44/39 -30/19 -20/11 -15/86 11/17 -73/22  43/38  37/45  52/11
 24/89 -85/79 -38/17   -8/7  40/13 89/72  43/27   25/7    7/2 -88/35
 82/97 -79/85  33/41 -25/84   -3/2 41/88 -31/19 -44/31  17/18  73/36
  45/7   -1/6 -83/22   9/25 -81/44  5/58 -49/40  30/71  -2/43  17/89
-21/47 -59/51  37/24  23/26   13/8  17/2 -61/22    4/3 -11/28  13/38
    -1  -71/4 -44/31  76/71  31/59 -9/14  37/40  34/31  81/14 -83/50
```

[75]:
```
print_mat(base_espace_engendre(F))
```

```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

[76]:
```
print_mat(base_noyau(F))
```

Sauf miracle, on devrait avoir trouvé la base canonique de $\mathbb{Q}^{10}$ comme base de l'image de f (surjection !) et la famille vide comme base du noyau (injection !).

Maintenant, une application linéaire de $\mathbb{Q}^5$ vers $\mathbb{Q}^{10}$, qui a de grandes chances d'être injective et qui ne peut pas être surjective. Restons raisonnable sur la taille des coefficients.

[77]:
```
F = famille_aleatoire(10, 5, M=5, N=1)
print_mat(F)
```

```
 4 -3 -4  5  2
 3  5 -5 -4  4
 2 -5  2 -1  5
-4  0  2 -1  5
 0  2 -5  2 -4
 3 -4 -5  5 -4
-2 -2 -3 -2  2
-4 -3  3 -2 -5
-3  2  2 -3  3
 0  0 -3  5  0
```

[78]:
```
print_mat(base_espace_engendre(F))
```

```
 5803    0      0     0      0
    0 -829      0     0      0
    0    0  -5803     0      0
    0    0      0  5803      0
    0    0      0     0  -5803
 1533  303  -4704 -2058  -7756
-4334  -26 -10004  3706 -12251
-7449  528  -7918  -459  -9157
-1718 -153    627  3124   1019
 4939  118   3889  2375    249
```

[79]:
```
print_mat(base_noyau(F))
```

Enfin, une application linéaire de $\mathbb{Q}^{10}$ vers $\mathbb{Q}^5$, qui a de grandes chances d'être surjective et qui ne peut pas être injective.

[80]:
```
F = famille_aleatoire(5, 10, M=5, N=1)
print_mat(F)
```

```
-4 -4  3  5 -3  5  2  4 -4  1
 4 -2 -2  4 -3  2  5 -4 -2  4
 0  5  3  4  1 -4 -5 -3 -2  1
 0  1  2 -2 -2 -3  0 -3  1 -3
-2  2  1 -5 -2  2  5  0 -5 -4
```

[81]:
```
print_mat(base_espace_engendre(F))
```

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

[82]:
```
print_mat(base_noyau(F))
```

```
1007  318 1908 -1113   53
-230 -180  375   870  -70
1258  602 1122 -1302  452
-284  -31 -196   406 -386
 632  543 -482 -1008    8
 530    0    0     0    0
   0  265    0     0    0
   0    0 1325     0    0
   0    0    0   530    0
   0    0    0     0  530
```

1.6 6. Équations cartésiennes

Si F est un sous-espace vectoriel de $\mathbb{Q}^q$ de dimension r, il est l'intersection de $q-r$ hyperplans de $\mathbb{Q}^q$. Comment obtenir des équations cartésiennes de ces hyperplans ? Le secret de la réussite consiste à *transposer*.

1.6.1 6.1 Transposition

Définition. Soient E et F deux $\mathbb{K}$-espaces vectoriels. Soit $f \in \mathcal{L}(E,F)$. La *transposée* de f est l'application linéaire $f^T \in \mathcal{L}(F^*,E^*)$ définie, pour toute forme linéaire $\varphi \in F^*$, par

$$f^T(\varphi) = \varphi \circ f$$

La linéarité de f^T est immédiate à vérifier.

Proposition. Soit $f \in \mathcal{L}(\mathbb{Q}^q, \mathbb{Q}^p)$. Soient $\mathcal{B} = (e_0, \dots, e_{q-1})$ et $\mathcal{B}' = (e'_0, \dots, e'_{p-1})$ des bases de $\mathbb{Q}^q$ et $\mathbb{Q}^p$. Soient $\mathcal{B}^* = (e_0^*, \dots, e_{q-1}^*)$ et $\mathcal{B}'^* = ({e'_0}^*, \dots, {e'_{p-1}}^*)$ les bases duales de $\mathcal{B}$ et $\mathcal{B}'$. La matrice de f^T dans les bases $\mathcal{B}'^*$ et $\mathcal{B}^*$ est la transposée de la matrice de f dans les bases $\mathcal{B}$ et $\mathcal{B}'$.

Démonstration. Soit A la matrice de f dans les bases $\mathcal{B}$ et $\mathcal{B}'$. Soit B la matrice de f^T dans les bases $\mathcal{B}'^*$ et $\mathcal{B}^*$. Pour tout $i \in [|0, p-1|]$ et tout $j \in [|0, q-1|]$,

$$f^T({e'_i}^*)(e_j) = {e'_i}^*(f(e_j))$$

En interprétant matriciellement,

$$\sum_{k=0}^{p-1} B_{ki} {e_k}^*(e_j) = \sum_{k=0}^{p-1} A_{kj} {e'_i}^*(e'_k)$$

c'est à dire

$$\sum_{k=0}^{p-1} B_{ki}\delta_{kj} = \sum_{k=0}^{p-1} A_{kj}\delta_{ik}$$

ou encore

$$B_{ji} = A_{ij}$$

Ainsi, $B = A^T$.

1.6.2 6.2 Hyperplans

Soit $\mathcal{F} = (u_0, \dots, u_{q-1})$ une famille de q vecteurs de $\mathbb{Q}^p$. Interprétons ces vecteurs comme les images des vecteurs de la base canonique $(e_0, \dots, e_{q-1})$ de $\mathbb{Q}^q$ par une application $f \in \mathcal{L}(\mathbb{Q}^q, \mathbb{Q}^p)$. Soit $H = \mathrm{Ker}(\varphi)$ un hyperplan de $\mathbb{Q}^p$, où φ est une forme linéaire non nulle sur $\mathbb{Q}^p$. On a $< \mathcal{F} > \subseteq H$ si et seulement si pour tout $i \in [|0, q-1|]$, $\varphi(u_i) = 0$, c'est à dire $\varphi(f(e_i)) = 0$. Par linéarité, ceci équivaut à $\varphi \circ f = 0$, ou encore $f^T(\varphi) = 0$, c'est à dire $\varphi \in \mathrm{Ker}(f^T)$.

Trouver les hyperplans contenant $\mathcal{F}$ revient donc à trouver le noyau de f^T. Si le rang de $\mathcal{F}$ est r, ce noyau est de dimension $q-r$. En obtenant une base de celui-ci, nous obtenons $q-r$ hyperplans dont l'intersection est $< \mathcal{F} >$. Or, nous savons déjà faire cela. Il suffit d'appliquer `base_noyau` à la matrice de la transposée de f, c'est à dire à la transposée de la matrice de f, qui est aussi la transposée de la matrice de la famille $\mathcal{F}$.

La fonction `intersection_hyperplans` ci-dessous est immédiate. Elle prend en paramètre une famille $\mathcal{F}$ de vecteurs $\mathbb{Q}^q$ de rang r et renvoie une matrice ayant $q-r$ colonnes représentant $q-r$ hyperplans (plus précisément, les coefficients de leurs équations cartésiennes) dont l'intersection est $< \mathcal{F} >$.

[83]:
```
def intersection_hyperplans(F): return base_noyau(transposee(F))
```

Testons sur nos exemples usuels. Tout d'abord, la famille $\mathcal{F}_1$.

[84]:
```
print_mat(F1)
```

```
1 4 7
2 5 8
3 6 9
```

Comme $\mathcal{F}_1$ est une famille de rang 2 de vecteurs de $\mathbb{Q}^3$, nous allons donc obtenir, en appelant `intersection_hyperplans`, une matrice colonne.

[85]:
```
print_mat(intersection_hyperplans(F1))
```

```
 1
-2
 1
```

Ainsi, $< \mathcal{F}_1 >$ est le plan de $\mathbb{Q}^3$ d'équation cartésienne $x - 2y + z = 0$.

Passons à $\mathcal{F}_2$ qui est une famille de rang 3 de 4 vecteurs de $\mathbb{Q}^5$.

[86]:
```
print_mat(F2)
```

```
 1  2  6  1
 2  5 17  3
-4 -3 -7 -3
 3  4 10  2
 1  8 22  0
```

Nous allons donc obtenir, en appelant `intersection_hyperplans`, une matrice 5×2.

[87]:
```
print_mat(intersection_hyperplans(F2))
```

```
-14 -54
  3  11
  1  -7
  4   0
  0   4
```

Ainsi, un couple d'équations cartésiennes de $< \mathcal{F}_2 >$ est

$$\begin{cases} -14x + 3y + z + 4t = 0 \\ -54x + 11y - 7z + 4u = 0 \end{cases}$$

La famille $\mathcal{F}_3$ est une famille de rang 3 de 6 vecteurs de $\mathbb{Q}^6$.

[88]:
```
print_mat(F3)
```

```
1 2 3 1 2 3
1 2 3 2 3 5
1 2 3 3 4 7
2 3 5 2 3 5
```

```
2 3 5 3 4 7
2 3 5 4 5 9
```

Nous allons donc obtenir, en appelant `intersection_hyperplans`, une matrice 6×3.

[89]:
```
print_mat(intersection_hyperplans(F3))
```

```
 1  1  1
-2  0  0
 1 -1 -1
 0 -2 -1
 0  2  0
 0  0  1
```

Un triplet d'équations cartésiennes de $< \mathcal{F}_3 >$ est donc

$$\begin{cases} x - 2y + z = 0 \\ x - z - 2t + 2u = 0 \\ x - z - t + v = 0 \end{cases}$$

Terminons par une famille aléatoire de 5 vecteurs de $\mathbb{Q}^{10}$.

[90]:
```
F = famille_aleatoire(10, 4, M=5, N=1)
print_mat(F)
```

```
 4 -5  0  2
 5 -3 -2  5
 2  2  5 -5
-4  1  5  5
 0  4  4  5
-3  2 -5 -1
 5 -5  3  4
-5  1 -3  1
 5  1  2 -2
 4 -4 -4  5
```

[91]:
```
print_mat(intersection_hyperplans(F))
```

```
 1615  855 -95  57   475 -285
-1770  220 -35 136 -1060 -890
 -783  857 -34 213 -1179  517
 -989  561 -37   1   223  191
 1330    0   0   0     0    0
    0 1330   0   0     0    0
    0    0  95   0     0    0
    0    0   0 266     0    0
    0    0   0   0  1330    0
    0    0   0   0     0 1330
```

Comme nous n'avons pas très envie de taper en LaTeX les équations des hyperplans, terminons ce notebook en beauté en écrivant une fonction `vers_latex` qui fait le travail.

[92]:
```
def vers_latex(A):
    s = r'$$\left\lbrace\begin{array}{l}'
    p, q = A.taille()
    for j in range(q):
        for i in range(p):
            if A[i][j] > 0:
                if i != 0: s += '+'
                s += str(A[i][j])
            if A[i][j] < 0:
                s += str(A[i][j])
            if A[i][j] != 0:
                s += 'x_{' + str(i) + '}'
        s += r'=0\\'
    s += r'\end{array}\right.$$'
    return s
```

[93]:
```
print(vers_latex(intersection_hyperplans(F)))
```

```
$$\left\lbrace\begin{array}{l}1615x_{0}-1770x_{1}-783x_{2}-
989x_{3}+1330x_{4}=0\\855x_{0}+220x_{1}+857x_{2}+561x_{3}+1330x_{5}=0\\-95x_{0}-
35x_{1}-34x_{2}-
37x_{3}+95x_{6}=0\\57x_{0}+136x_{1}+213x_{2}+1x_{3}+266x_{7}=0\\475x_{0}-
1060x_{1}-1179x_{2}+223x_{3}+1330x_{8}=0\\-285x_{0}-
890x_{1}+517x_{2}+191x_{3}+1330x_{9}=0\\\end{array}\right.$$
```

Un copier-coller dans une cellule Markdown affichera ce que nous voulons. Encore plus simple, le module `IPython.display` contient deux fonctions qui permettent d'afficher du code LaTeX. Si ce module est installé sur votre machine, plus besoin de copier-coller.

[94]:
```
from IPython.display import display, Math
```

[95]:
```
display(Math(vers_latex(intersection_hyperplans(F))))
```

$$\left\lbrace\begin{array}{l}1615x_{0}-1770x_{1}-783x_{2}-989x_{3}+1330x_{4}=0\\855x_{0}+220x_{1}+857x_{2}+561x_{3}+1330x_{5}=0\\-95x_{0}-35x_{1}-34x_{2}-37x_{3}+95x_{6}=0\\57x_{0}+136x_{1}+213x_{2}+1x_{3}+266x_{7}=0\\475x_{0}-1060x_{1}-1179x_{2}+223x_{3}+1330x_{8}=0\\-285x_{0}-890x_{1}+517x_{2}+191x_{3}+1330x_{9}=0\\\end{array}\right.$$