ImKer

February 3, 2020

1 Noyau et image d'une application linéaire

Marc Lorenzi

3 février 2020

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

1.1 0. Introduction

1.1.1 0.1 Objectifs, notations

Soit $f \in \mathcal{L}(E, F)$ une application linéaire entre deux $\mathbb{R}$-espaces vectoriels de dimension finie. Ce notebook se propose d'utiliser les méthodes vue en cours pour le calcul pratique de certaines quantités associées à f : son rang, son noyau et son image. Les fonctions que nous allons développer permettront évidemment de déterminer aussi l'espace engendrée par une famille de vecteurs le rang de cette famille.

L'idée n'est pas d'avoir une boîte noire, d'excellentes bibliothèques existent déjà pour calculer ces quantités. Bien au contraire, le but est de comprendre comment on calcule un rang, un espace engendré, un noyau, etc. On laisse à l'utilisateur l'initiative des opérations à effectuer. L'automatisation viendra petit à petit … jusqu'à l'écriture de deux fonctions `ker` et `im` qui feront toutes seules le calcul d'une base du noyau et de l'image de f.

Nous verrons ensuite comment ce que nous avons fait permet de travailler sur les sous-espaces vectoriels de E : bases, équations cartésiennes, sommes, intersections, supplémentarité, etc.

Remarque : Comme beaucoup de calculs pratiques font intervenir des matrices d'entiers, nous ferons en sorte que nos opérations préservent cette propriété. Tous nos calculs seront donc exacts et jamais approchés.

1.1.2 0.2 Matrices d'exemples

Nous allons bien évidemment travailler sur des matrices, que nous représenterons en Python comme des listes de listes. Voici une fonction renvoyant une matrice que nous prendrons fréquemment comme exemple.

[2]:
```
def exemple(p=3, q=3):
    n = p * q
    A = [q * [0] for i in range(p)]
    for k in range(n):
        A[k % p][k // p] = k + 1
    return A
```

[3]:
```
exemple(5, 4)
```

[3]:
```
[[1, 6, 11, 16],
 [2, 7, 12, 17],
 [3, 8, 13, 18],
 [4, 9, 14, 19],
 [5, 10, 15, 20]]
```

Nous aurons souvent besoin du nombre de lignes et du nombre de colonnes d'une matrice.

[4]:
```
def nblig(A): return len(A)
def nbcol(A):
    if A == []: raise Exception('Nombre de colonnes indetermine')
    return len(A[0])
```

[5]:
```
A = exemple(7, 8)
nblig(A), nbcol(A)
```

[5]:
```
(7, 8)
```

Écrivons également une fonction qui permet d'avoir un affichage des matrices plus agréable. On évalue et on stocke pour chaque colonne de la matrice la longueur maximale de la représentation de ses coefficients sous forme de chaîne de caractères. Puis on affiche la matrice ligne par ligne en utilisant les longueurs maximales pré-calculées. La matrice est alors parfaitement tabulée.

[6]:
```
def prnt(A):
    if A == []: print('_')
    else:
        p, q = nblig(A), nbcol(A)
        ws = q * [0]
        for j in range(q):
            for i in range(p):
                w = len('%s' % (A[i][j]))
                if w > ws[j]: ws[j] = w
        for i in range(p):
            s = ''
            for j in range(q):
                s1 = '%d' % A[i][j]
                sp = ws[j] - len(s1) + 1
                s1 = (sp * ' ') + s1
                s = s + s1
```

```
            print(s)
```

[7]:
```
prnt(exemple(12, 25))
```

```
  1 13 25 37 49 61 73 85  97 109 121 133 145 157 169 181 193 205 217 229 241 253
265 277 289
  2 14 26 38 50 62 74 86  98 110 122 134 146 158 170 182 194 206 218 230 242 254
266 278 290
  3 15 27 39 51 63 75 87  99 111 123 135 147 159 171 183 195 207 219 231 243 255
267 279 291
  4 16 28 40 52 64 76 88 100 112 124 136 148 160 172 184 196 208 220 232 244 256
268 280 292
  5 17 29 41 53 65 77 89 101 113 125 137 149 161 173 185 197 209 221 233 245 257
269 281 293
  6 18 30 42 54 66 78 90 102 114 126 138 150 162 174 186 198 210 222 234 246 258
270 282 294
  7 19 31 43 55 67 79 91 103 115 127 139 151 163 175 187 199 211 223 235 247 259
271 283 295
  8 20 32 44 56 68 80 92 104 116 128 140 152 164 176 188 200 212 224 236 248 260
272 284 296
  9 21 33 45 57 69 81 93 105 117 129 141 153 165 177 189 201 213 225 237 249 261
273 285 297
 10 22 34 46 58 70 82 94 106 118 130 142 154 166 178 190 202 214 226 238 250 262
274 286 298
 11 23 35 47 59 71 83 95 107 119 131 143 155 167 179 191 203 215 227 239 251 263
275 287 299
 12 24 36 48 60 72 84 96 108 120 132 144 156 168 180 192 204 216 228 240 252 264
276 288 300
```

Nous utiliserons aussi parfois des matrices aléatoires. La fonction `randmat` renvoie une matrice "aléatoire" de taille $p \times q$ dont les coefficients sont des entiers de l'intervalle $[-r, r]$.

[8]:
```
def randmat(p, q, r=10):
    A = [q * [0] for i in range(p)]
    for i in range(p):
        for j in range(q):
            A[i][j] = random.randint(-r, r)
    return A
```

[9]:
```
A = randmat(10, 15, r=10000)
prnt(A)
```

```
  -366  2938  2963  5677  5580  9834 -6363  5652  6824  3329 -5185  1201 -3916
7654 -9916
 -4800 -9801 -3055 -9659 -2466 -1664 -7415 -4223 -7735  9303  5158 -8293  4491
4197  1930
 -5151 -4980   710  8456 -6010  5220  2041  6685  7595 -2506  6839  8116 -7589
-1926 -1642
```

```
   517 -9974 -9753  4110 -2433  1533  8716  6883  2855 -6513 -4803 -1358 -6632
-6610  3171
  5997 -8143  3712  6607  4884  4500  7763  -764  6339  6408 -2846  1942   177
1750 -4062
 -8181 -1567 -7770  2040 -8089 -3017 -8821 -5175 -4556  9023  8923  9507 -2160
8457  1117
 -6786 -9213  7539 -6722 -6773  7550  8443 -3356 -5877 -8252  6684 -4629 -2110
-6969  4803
 -6719  6809 -7881 -2207   211 -7476 -8474  8421 -4790  3087 -7800 -1389   340
9220 -6044
 -7131 -3869  5187 -2227 -8913 -1661  3394  4328  8899 -2554 -2631 -7087  3375
-6270  9163
  6997 -4071  8885  -361 -4337  6898  -843   119 -9733 -3075  9029 -1498  3733
-5433 -1522
```

1.2 1. Rang, espace engendré

1.2.1 1.1 Introduction

On ne change pas l'espace engendré par les colonnes de la matrice A (et donc pas non plus son rang) en effectuant les opérations suivantes :

- Échanger deux colonnes.
- Multiplier une colonne par un réel non nul.
- Ajouter à une colonne un multiple d'un autre colonne.

Nous allons tout d'abord définir ces opérations au moyen de quelques fonctions qui se passent de commentaires.

1.2.2 1.2 Échange de deux colonnes

La fonction `echanger` échange les colonnes j_1 et j_2 de la matrice A.

$$C_{j_1} \longleftrightarrow C_{j_2}$$

[10]:
```
def echanger(A, j1, j2, dbg=True):
    if dbg: print('C%s <-> C%s' % (j1, j2))
    for i in range(nblig(A)):
        A[i][j1], A[i][j2] = A[i][j2], A[i][j1]
```

[11]:
```
A = exemple(5, 8)
echanger(A, 3, 6)
prnt(A)
```

```
C3 <-> C6
 1  6 11 31 21 26 16 36
 2  7 12 32 22 27 17 37
```

```
 3  8 13 33 23 28 18 38
 4  9 14 34 24 29 19 39
 5 10 15 35 25 30 20 40
```

1.2.3 1.3 Produit d'une colonne par un scalaire

La fonction `mult` multiplie la colonne j de la matrice A par le scalaire t.

$$C_j \longleftarrow tC_j$$

[12]:
```
def mult(A, j, t, dbg=True):
    if dbg: print('C%s <- (%s)C%s' % (j, t, j))
    for i in range(nblig(A)):
        A[i][j] = t * A[i][j]
```

[13]:
```
A = exemple(5, 10)
mult(A, 4, -5)
prnt(A)
```

```
C4 <- (-5)C4
 1  6 11 16 -105 26 31 36 41 46
 2  7 12 17 -110 27 32 37 42 47
 3  8 13 18 -115 28 33 38 43 48
 4  9 14 19 -120 29 34 39 44 49
 5 10 15 20 -125 30 35 40 45 50
```

Histoire de pouvoir aussi diviser une colonne par un facteurs commun de ses éléments, définissons une fonction `div`.

[14]:
```
def div(A, j, t, dbg=True):
    if dbg: print('C%s <- (1/%s)C%s' % (j, t, j))
    for i in range(nblig(A)):
        A[i][j] = A[i][j] // t
```

[15]:
```
A = [[5, 10], [15, 20]]
prnt(A)
div(A, 0, 5)
div(A, 1, 10)
prnt(A)
```

```
  5 10
 15 20
C0 <- (1/5)C0
C1 <- (1/10)C1
 1 1
 3 2
```

1.2.4 1.4 Ajout à une colonne d'un multiple d'une autre colonne

La fonction `add` ajoute à la colonne j_1 de la matrice A t fois la colonne j_2.

$$C_{j_1} \longleftarrow C_{j_1} + tC_{j_2}$$

[16]:
```
def add(A, j1, t, j2, dbg=True):
    if dbg: print('C%s <- C%s + (%s)C%s' % (j1, j1, t, j2))
    for i in range(nblig(A)):
        A[i][j1] = A[i][j1] + t * A[i][j2]
```

[17]:
```
A = exemple(5, 10)
add(A, 2, -11, 0)
add(A, 7, -36, 0)
prnt(A)
```

```
C2 <- C2 + (-11)C0
C7 <- C7 + (-36)C0
 1  6   0 16 21 26 31    0 41 46
 2  7 -10 17 22 27 32  -35 42 47
 3  8 -20 18 23 28 33  -70 43 48
 4  9 -30 19 24 29 34 -105 44 49
 5 10 -40 20 25 30 35 -140 45 50
```

1.2.5 1.5 Pivot

Première étape de l'automatisation, la fonction `pivoter` combine certaines des opérations déjà décrites. Soit A une matrice $p \times q$. La fonction `pivoter` prend en paramètres un indice de ligne ℓ et un indice de colonne k tels que $t = A_{\ell k} \neq 0$. Le but de `pivoter` est d'annuler les coefficients $A_{\ell(k+1)}, A_{\ell(k+2)}, \ldots, A_{\ell q}$.

Pour $j = k+1, \ldots, q-1$, on effectue l'opération

$$C_j \longleftarrow \frac{t}{\delta} C_j - \frac{u}{\delta} C_k$$

où $u = A_{\ell j}$ et $\delta = pgcd(t, u)$. La fonction `pivoter` effectue au total $2(q-k)$ opérations "élémentaires" sur les colonnes de A. Elle remplit bien son rôle puisque

$$\frac{t}{\delta} A_{\ell j} - \frac{u}{\delta} A_{\ell k} = \frac{t}{\delta} u - \frac{u}{\delta} t = 0$$

Remarque : On pourrait automatiser `pivoter` afin que la fonction trouve elle-même les indices ℓ et k. Nous le ferons plus loin mais pour l'instant il est du devoir de l'utilisateur d'observer la matrice A et de choisir les valeurs de ℓ et de k qui lui semblent pertinentes !

[18]:
```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

[19]:
```
def pivoter(A, l, k, dbg=True):
    t = A[l][k]
    for j in range(k + 1, nbcol(A)):
        u = A[l][j]
        d = gcd(t, u)
        if dbg: print('C%s <- (%s)C%s + (%s)C%s' % (j, t // d, j, -u // d, k))
        for i in range(0, nblig(A)):
            A[i][j] = (t // d) * A[i][j] - (u // d) * A[i][k]
```

[20]:
```
A = exemple(5, 6)
pivoter(A, 0, 0)
prnt(A)
```

```
C1 <- (1)C1 + (-6)C0
C2 <- (1)C2 + (-11)C0
C3 <- (1)C3 + (-16)C0
C4 <- (1)C4 + (-21)C0
C5 <- (1)C5 + (-26)C0
 1   0   0   0   0    0
 2  -5 -10 -15 -20  -25
 3 -10 -20 -30 -40  -50
 4 -15 -30 -45 -60  -75
 5 -20 -40 -60 -80 -100
```

Maintenant que les fonctions de base sont mises en place, regardons ce que l'on peut faire avec.

1.3 2. Quelques exemples

1.3.1 2.1 Notre exemple préféré

Prenons tout d'abord notre matrice favorite.

[21]:
```
A = exemple(6, 9)
prnt(A)
```

```
 1  7 13 19 25 31 37 43 49
 2  8 14 20 26 32 38 44 50
 3  9 15 21 27 33 39 45 51
 4 10 16 22 28 34 40 46 52
 5 11 17 23 29 35 41 47 53
 6 12 18 24 30 36 42 48 54
```

On pivote en 0,0.

[22]:
```
pivoter(A, 0, 0)
prnt(A)
```

```
C1 <- (1)C1 + (-7)C0
C2 <- (1)C2 + (-13)C0
C3 <- (1)C3 + (-19)C0
C4 <- (1)C4 + (-25)C0
C5 <- (1)C5 + (-31)C0
C6 <- (1)C6 + (-37)C0
C7 <- (1)C7 + (-43)C0
C8 <- (1)C8 + (-49)C0
 1   0   0   0    0    0    0    0    0
 2  -6 -12 -18  -24  -30  -36  -42  -48
 3 -12 -24 -36  -48  -60  -72  -84  -96
 4 -18 -36 -54  -72  -90 -108 -126 -144
 5 -24 -48 -72  -96 -120 -144 -168 -192
 6 -30 -60 -90 -120 -150 -180 -210 -240
```

Puis en 1, 1.

[23]:
```
pivoter(A, 1, 1)
prnt(A)
```

```
C2 <- (1)C2 + (-2)C1
C3 <- (1)C3 + (-3)C1
C4 <- (1)C4 + (-4)C1
C5 <- (1)C5 + (-5)C1
C6 <- (1)C6 + (-6)C1
C7 <- (1)C7 + (-7)C1
C8 <- (1)C8 + (-8)C1
 1   0 0 0 0 0 0 0 0
 2  -6 0 0 0 0 0 0 0
 3 -12 0 0 0 0 0 0 0
 4 -18 0 0 0 0 0 0 0
 5 -24 0 0 0 0 0 0 0
 6 -30 0 0 0 0 0 0 0
```

Effectuons une dernière opération.

[24]:
```
div(A, 1, A[1][1])
prnt(A)
```

```
C1 <- (1/-6)C1
 1 0 0 0 0 0 0 0 0
 2 1 0 0 0 0 0 0 0
 3 2 0 0 0 0 0 0 0
 4 3 0 0 0 0 0 0 0
 5 4 0 0 0 0 0 0 0
 6 5 0 0 0 0 0 0 0
```

Cette matrice a même rang que A et ses colonnes engendrent le même espace que les colonnes de A. Ainsi, $rgA = 2$ et l'image d'une application linéaire associée à A a pour base les deux premières colonnes de A.

1.3.2 2.2 Rang d'une famille de vecteurs

On se place dans $E = \mathbb{R}^5$. Soient - $x_1 = (1, 2, -4, 3, 1)$ - $x_2 = (2, 5, -3, 4, 8)$ - $x_3 = (6, 17, -7, 10, 22)$ - $x_4 = (1, 3, -3, 2, 0)$ Quel est le rang de la famille (x_1, x_2, x_3, x_4) ?

Tout d'abord, voici une fonction bien utile qui renvoie la transposée d'une matrice. Cela évite de faire de la gymnastique pour entrer les données en lignes alors qu'on nous les donne en colonnes.

[25]:
```
def transp(A):
    if A == []: return []
    p, q = nblig(A), nbcol(A)
    B = [p * [0] for i in range(q)]
    for i in range(q):
        for j in range(p):
            B[i][j] = A[j][i]
    return B
```

[26]:
```
A = transp([[1, 2, -4, 3, 1], [2, 5, -3, 4, 8], [6, 17, -7, 10, 22], [1, 3, -3,␣
↪2, 0]])
prnt(A)
```

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

Et maintenant, pivotons.

[27]:
```
pivoter(A, 0, 0)
prnt(A)
```

```
C1 <- (1)C1 + (-2)C0
C2 <- (1)C2 + (-6)C0
C3 <- (1)C3 + (-1)C0
 1  0  0  0
 2  1  5  1
-4  5 17  1
 3 -2 -8 -1
 1  6 16 -1
```

[28]:
```
pivoter(A, 1, 1)
prnt(A)
```

```
C2 <- (1)C2 + (-5)C1
C3 <- (1)C3 + (-1)C1
  1  0   0  0
  2  1   0  0
 -4  5  -8 -4
  3 -2   2  1
  1  6 -14 -7
```

[29]:
```
pivoter(A, 2, 2)
prnt(A)
```

```
C3 <- (2)C3 + (-1)C2
  1  0   0 0
  2  1   0 0
 -4  5  -8 0
  3 -2   2 0
  1  6 -14 0
```

Encore une petite simplification …

[30]:
```
div(A, 2, -2)
prnt(A)
```

```
C2 <- (1/-2)C2
  1  0  0 0
  2  1  0 0
 -4  5  4 0
  3 -2 -1 0
  1  6  7 0
```

Le rang de la famille $(x_1, \ldots, x_4)$ est donc 3.

1.3.3 2.3 Matrices aléatoires

Ne prenez surtout pas exemple sur la fonction qui vient … La fonction `kamikaze` appelle `pivot` successivement sur les indices $(0,0), (1,1), \ldots, (m,m)$ où $m = \min(p,q)$, en priant pour ne jamais tomber sur un pivot nul. Cela a de bonnes chances de fonctionner sur une matrice aléatoire, et quasiment aucune chance de fonctionner sur des matrices particulières, surtout sur les matrices qui apparaissent dans les exercices classiques :-).

[31]:
```
def kamikaze(A, dbg=True):
    m = min(nblig(A), nbcol(A))
    for k in range(m):
        pivoter(A, k, k, dbg)
```

[32]:
```
A = randmat(4, 6)
kamikaze(A, dbg=True)
prnt(A)
```

```
C1 <- (1)C1 + (-4)C0
C2 <- (-1)C2 + (-5)C0
C3 <- (2)C3 + (-7)C0
C4 <- (2)C4 + (-9)C0
C5 <- (-1)C5 + (-4)C0
C2 <- (32)C2 + (-49)C1
C3 <- (32)C3 + (-43)C1
C4 <- (32)C4 + (-87)C1
C5 <- (8)C5 + (-7)C1
C3 <- (-95)C3 + (-91)C2
C4 <- (19)C4 + (-37)C2
C5 <- (-95)C5 + (-23)C2
C4 <- (835)C4 + (-33)C3
C5 <- (1336)C5 + (-127)C3
 2   0   0       0 0 0
 9 -32   0       0 0 0
 8 -30 190       0 0 0
 6 -34 450 -106880 0 0
```

Depuis quelque temps nous avons remarqué que, souvent, des entiers sont en facteurs dans les colonnes. Écrivons une fonction qui divise une colonne d'une matrice par le pgcd des coefficients de cette colonne.

[33]:
```
def gcdcol(A, j):
    d = 0
    for i in range(nblig(A)):
        d = gcd(d, A[i][j])
    return d
```

[34]:
```
def normaliser_colonne(A, j, dbg):
    d = gcdcol(A, j)
    if d != 0:
        div(A, j, d, dbg)
```

Voici notre nouvelle fonction `kamikaze` qui simplifie, après chaque appel à pivot, la colonne dans laquelle se trouve le pivot.

[35]:
```
def kamikaze2(A, dbg=True):
    for k in range(min(nblig(A), nbcol(A))):
        pivoter(A, k, k, dbg)
        normaliser_colonne(A, k, dbg)
```

Testons.

[36]:
```
A = randmat(4, 6)
kamikaze2(A, dbg=True)
prnt(A)
```

```
C1 <- (-5)C1 + (-2)C0
```

```
C2 <- (5)C2 + (-3)C0
C3 <- (-10)C3 + (-3)C0
C4 <- (-5)C4 + (-2)C0
C5 <- (-2)C5 + (-1)C0
C0 <- (1/-1)C0
C2 <- (-13)C2 + (-33)C1
C3 <- (-13)C3 + (-83)C1
C4 <- (13)C4 + (-23)C1
C5 <- (-13)C5 + (-1)C1
C1 <- (1/1)C1
C3 <- (-1)C3 + (-3)C2
C4 <- (4)C4 + (-41)C2
C5 <- (-40)C5 + (-167)C2
C2 <- (1/-80)C2
C4 <- (245)C4 + (-1868)C3
C5 <- (245)C5 + (-8056)C3
C3 <- (1/15925)C3
 10  0  0 0 0 0
 -1 13  0 0 0 0
 -9 17  1 0 0 0
 10 65 39 1 0 0
```

Une matrice 13×13 pour terminer ? Des matrices de taille 100×100 sont parfaitement envisageables, mais inutile alors d'espérer les afficher joliment …

[37]:
```
A = randmat(13, 13)
prnt(A)
```

```
  1  6  9   1 -8  2 -8  7  -6  -9 -2 -10  7
 -5  7  4  -5 -8 -9 -1  7  -1  -1 -2  -3 -7
  9  3  2   2  6 -6  7 -2  -9  -3 -1   9 -9
 -9 -8  8  -1  3 -4 -4  2  -5   3  7   6 -2
 -8 10  7   8  4 10 -1  4 -10   4 -9   0  4
 -9  0 -8   5  2  4 10 -9   8  -2 -8  -7 -3
 -4  9  7   1  6  2 -9  4  -8  -6 -4  -8 10
  4  4  4  -6 -8  5 -3 -9   5 -10  2   1 -6
 -3 -7  4  -3 -2 -2 -7 -3  -8   6 -6   7 -5
  3  3 -8   9  8  7 -1  2   2   7  3   1 -9
 -2 -1  5  -4 -4  0 -6 -1  -6   3  7  -9 -1
 -1  0  9 -10  8  3  2 -3  -6  10 -3   3 -7
 -6 -6 -7  10 -1  0 -7  9   0  -2  2   8 -3
```

En zéro seconde, `kamikaze2` renvoie le résultat.

[38]:
```
kamikaze2(A, dbg=False)
prnt(A)
```

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

```
  5  37    0     0      0        0         0          0           0
0             0               0 0
 -9 -51 -424     0      0        0         0          0           0
0             0               0 0
  9  46 1039 -3881      0        0         0          0           0
0             0               0 0
  8  58   81  6217 190311        0         0          0           0
0             0               0 0
  9  54   55  5551 115435   898650         0          0           0
0             0               0 0
  4  33  -26  2302 107738  -792882  17085042          0           0
0             0               0 0
 -4 -20 -204 -2812 -85008 -1128699 -34769781 -534306933           0
0             0               0 0
  3  11  608 -4256 -63418  -125921   1534351  -83343403  5686719091
0             0               0 0
 -3 -15 -560  6464 115228   671246  37230374  400041868 -1184549896
84300195858             0               0 0
  2  11  312 -3032 -48112  -520103  -9407057 -185905657  3816228886
-18005977385  555328204909               0 0
  1   6  372 -6420 -55854 -2627902 -59396938 -679150184  -823259143
-121752522009 -173888949105 -23556540852182 0
  6  30  269  4901  86494  2228242  63998848  784781120  2240154319
40231985075  392039558162  46235861277275 1
```

1.4 3. Noyau et image

1.4.1 3.1 Introduction

Soit $A \in \mathcal{M}_{pq}(\mathbb{K})$. La matrice A est la matrice dans les bases canoniques de $\mathbb{K}^q$ et $\mathbb{K}^p$ d'une application linéaire f. Que valent le noyau et l'image de f ?

Pour l'image de f c'est facile. En effet les opérations "élémentaires" sur les colonnes de A ne changent pas l'espace engendré par les colonnes. Bref, nous avons déjà écrit une fonction qui calcule une base Imf. On pivote et on récupère les colonnes non nulles de la matrice finale.

Pour le noyau de f, il va nous falloir être un peu plus soigneux.

Appelons $\mathcal{B} = (e_1, \ldots, e_q)$ la base canonique de $\mathbb{K}^q$. Soit $\mathcal{B}'$ la base canonique de $\mathbb{K}^p$.

À chaque opération sur les colonnes de A nous allons enregistrer quelle opération a été effectuée. Où enregistrer cela ? Dans une matrice $B \in GL_q(\mathbb{K})$ qui est la matrice de passage d'une base de $\mathbb{K}^q$ à la base $\mathcal{B}$. La matrice B est initialement la matrice identité $I_q = P^{\mathcal{B}}_{\mathcal{B}}$. Lors d'une opération élémentaire sur les colonnes de A on effectue la même opération sur la matrice B. Et alors ? La matrice A est la matrice de f dans les bases $\mathcal{B}$ et $\mathcal{B}'$: la $j$ème colonne de A contient donc le vecteur $f(e_j)$ (ou plutôt ses coordonnées dans la base $\mathcal{B}'$).

Prenons un exemple concret : effectuons l'opération $C_2 \longleftarrow C_2 + 3C_1$. La colonne 2 de A contient alors $f(e_2 + 3e_1)$. Si l'on effectue la même opération sur la matrice B, celle-ci devient la matrice de

passage de $\mathcal{B}$ à une nouvelle base qui a les mêmes vecteurs que $\mathcal{B}$, sauf le second qui vaut $e_2 + 3e_1$. Nous avons ainsi "stocké" $e_2 + 3e_1$ dans la matrice B.

De façon générale, $B = P_{\mathcal{B}}^{\mathcal{B}_1}$ où $\mathcal{B}_1$ est telle que $A = mat_{\mathcal{B}_1\mathcal{B}'} f$.

Au travail !

1.4.2 3.2 Réécriture des opérations élémentaires

Ces fonctions sont identiques à celles qui ont déjà été écrites, mis à part le fait qu'elle prennent un paramètre B supplémentaire qui est la matrice de passage décrite au paragraphe précédent. Chaque opération sur A est répercutée sur B.

[39]:
```
def echanger2(A, j1, j2, B):
    for i in range(nblig(A)):
        A[i][j1], A[i][j2] = A[i][j2], A[i][j1]
    for i in range(nblig(B)):
        B[i][j1], B[i][j2] = B[i][j2], B[i][j1]
```

[40]:
```
def mult2(A, j, t, B):
    for i in range(nblig(A)):
        A[i][j] = t * A[i][j]
    for i in range(nblig(B)):
        B[i][j] = t * B[i][j]
```

[41]:
```
def div2(A, j, t, B):
    for i in range(nblig(A)):
        A[i][j] = A[i][j] // t
    for i in range(nblig(B)):
        B[i][j] = B[i][j] // t
```

[42]:
```
def add2(A, j1, t, j2, B):
    for i in range(nblig(A)):
        A[i][j1] = A[i][j1] + t * A[i][j2]
    for i in range(nblig(B)):
        B[i][j1] = B[i][j1] + t * B[i][j2]
```

[43]:
```
def pivoter2(A, l, k, B):
    t = A[l][k]
    for j in range(k + 1, nbcol(A)):
        u = A[l][j]
        d = gcd(t, u)
        for i in range(0, nblig(A)):
            A[i][j] = (t // d) * A[i][j] - (u // d) * A[i][k]
        for i in range(0, nblig(B)):
            B[i][j] = (t // d) * B[i][j] - (u // d) * B[i][k]
```

La fonction `eye` renvoie la matrice identité d'ordre n, valeur initiale pour la matrice B.

[44]:
```
def eye(n):
    A = [n * [0] for i in range(n)]
    for i in range(n):
        A[i][i] = 1
    return A
```

Testons sur un exemple.

[45]:
```
A = exemple(5, 6)
B = eye(6)
prnt(A)
print()
prnt(B)
```

```
1  6 11 16 21 26
2  7 12 17 22 27
3  8 13 18 23 28
4  9 14 19 24 29
5 10 15 20 25 30

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

[46]:
```
pivoter2(A, 0, 0, B)
prnt(A)
print()
prnt(B)
```

```
1   0   0   0   0    0
2  -5 -10 -15 -20  -25
3 -10 -20 -30 -40  -50
4 -15 -30 -45 -60  -75
5 -20 -40 -60 -80 -100

1 -6 -11 -16 -21 -26
0  1   0   0   0   0
0  0   1   0   0   0
0  0   0   1   0   0
0  0   0   0   1   0
0  0   0   0   0   1
```

[47]:
```
pivoter2(A, 1, 1, B)
prnt(A)
print()
```

```
prnt(B)
```

```
1   0 0 0 0 0
2  -5 0 0 0 0
3 -10 0 0 0 0
4 -15 0 0 0 0
5 -20 0 0 0 0

1 -6  1  2  3  4
0  1 -2 -3 -4 -5
0  0  1  0  0  0
0  0  0  1  0  0
0  0  0  0  1  0
0  0  0  0  0  1
```

Bilan :

- Le rang de A est 2.
- Une base de Imf est la famille des deux premières colonnes de A (les colonnes non nulles).
- Une base de $\ker f$ est la famille des 4 dernières colonnes de B, celles qui correspondent aux colonnes nulles de A.

1.4.3 3.3 Automatisons le tout

Tout le problème, à chaque étape, est la recherche du pivot. Quelles valeurs prendre pour k et ℓ, passés en paramètres à `pivoter` ? À la $k$ième itération, le couple (k, k) paraît judicieux. Mais il ne l'est pas toujours car le coefficient correspondant dans la matrice A est peut-être nul. Sans justification dans ce notebook, l'idée est de choisir un couple (ℓ, k') où

$$\ell = \min\{i \geq k, \exists j \geq k, A_{ij} \neq 0\}$$

$$k' = \min\{j \geq k, A_{\ell j} \neq 0\}$$

Ce couple d'indices se trouve à l'aide de deux boucles imbriquées. On parcourt les lignes $k, k+1, k+2, \ldots$ à partir de la colonne k et on s'arrête dès qu'on trouve un coefficient non nul. On renvoie alors les indices de ligne et de colonne de ce coefficient. On cherche donc une ligne d'indice minimal à partir de la ligne k et, pour cette ligne, un indice de colonne minimal, qui fournissent un coefficient non nul de la matrice.

Si un tel coefficient n'existe pas c'est qu'il n'y a plus aucune opération à faire sur la matrice A parce que tous les coefficients observés sont nuls.

La fonction `chercher_pivot` cherche un tel couple (ℓ, k) d'indices. Elle renvoie $(\ell, k, True)$ si elle en trouve un, et $(-1, -1, False)$ sinon.

[48]:
```
def chercher_pivot(A, k):
    for i in range(k, nblig(A)):
        for j in range(k, nbcol(A)):
```

```
            if A[i][j] != 0: return (i, j, True)
    return (-1, -1, False)
```

La fonction `imker` automatise tout ce que nous venons de raconter. Elle prend une matrice A en paramètre et renvoie un couple (A, B) où A est la matrice "trigonalisée" et B est une matrice de passage.

Petits détails : - On opère sur une **copie** de A pour ne pas modifier la matrice initiale. - On divise en fin de fonction les colonnes d e$A* et B par les pgcd de leurs coefficients pour avoir un résultat aussi simple que possible.

[49]:
```
def imker(A):
    A = [A[i].copy() for i in range(nblig(A))]
    B = eye(nbcol(A))
    k = 0
    while True:
        i, j, b = chercher_pivot(A, k)
        if not b: break
        if j != k: echanger2(A, j, k, B)
        pivoter2(A, i, k, B)
        k = k + 1
    for j in range(nbcol(A)): normaliser_colonne(A, j, dbg=False)
    for j in range(nbcol(B)): normaliser_colonne(B, j, dbg=False)
    return (A, B)
```

Testons.

[50]:
```
A = exemple(6, 10)
prnt(A)
```

```
 1  7 13 19 25 31 37 43 49 55
 2  8 14 20 26 32 38 44 50 56
 3  9 15 21 27 33 39 45 51 57
 4 10 16 22 28 34 40 46 52 58
 5 11 17 23 29 35 41 47 53 59
 6 12 18 24 30 36 42 48 54 60
```

[51]:
```
A1, B = imker(A)
prnt(A1)
print()
prnt(B)
```

```
1 0 0 0 0 0 0 0 0 0
2 1 0 0 0 0 0 0 0 0
3 2 0 0 0 0 0 0 0 0
4 3 0 0 0 0 0 0 0 0
5 4 0 0 0 0 0 0 0 0
6 5 0 0 0 0 0 0 0 0
```

```
 1 -7  1  2  3  4  5  6  7  8
 0  1 -2 -3 -4 -5 -6 -7 -8 -9
 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
```

1.4.4 3.4 Noyau, image

Tout a été dit : on appelle `imker`, qui renvoie deux matrices A_1 et B. Pour obtenir une base de Imf on récupère les colonnes non nulles de A_1. Pour obtenir une base de $\ker f$, récupère les colonnes de B qui correspondent aux colonnes nulles de A_1. Pourquoi cela fonctionne-t-il ?

Les opérations effectuées sur B ne changent pas son rang. Or, $B = I$ au début de notre algorithme. La matrice renvoyée est donc inversible. De plus, par le théorème du rang, le nombre de colonnes nulles de A_1 est égal à $\dim \ker f$. En récupérant les colonnes correspondantes de B on obtient donc une famille libre de $\ker f$ dont le cardinal est justement la dimension du noyau de f, c'est à dire une base.

[52]:
```
def noyau(A):
    A, B = imker(A)
    return transp([col(B, k) for k in col_nulles(A)])
```

[53]:
```
def image(A):
    A, B = imker(A)
    return transp([col(A, k) for k in range(nbcol(A)) if not (est_col_nulle(A,␣
 ↪k))])
```

La fonction `col` renvoie la **liste** des coefficients de la $j$ème colonne de A.

[54]:
```
def col(A, j):
    return [A[i][j] for i in range(nblig(A))]
```

La fonction `est_col_nulle` teste si la $j$ème colonne de A ne contient que des zéros.

[55]:
```
def est_col_nulle(A, j):
    for i in range(nblig(A)):
        if A[i][j] != 0: return False
    return True
```

La fonction `col_nulles` renvoie la liste des numéros des colonnes nulles de A.

[56]:
```
def col_nulles(A):
    return [j for j in range(nbcol(A)) if est_col_nulle(A, j)]
```

Testons sur notre exemple habituel.

[57]:
```
A = exemple(6, 10)
prnt(noyau(A))
print()
prnt(image(A))
```

```
 1  2  3  4  5  6  7  8
-2 -3 -4 -5 -6 -7 -8 -9
 1  0  0  0  0  0  0  0
 0  1  0  0  0  0  0  0
 0  0  1  0  0  0  0  0
 0  0  0  1  0  0  0  0
 0  0  0  0  1  0  0  0
 0  0  0  0  0  1  0  0
 0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  1

1 0
2 1
3 2
4 3
5 4
6 5
```

1.4.5 3.4 Vérifications

Voici une fonction calculant le produit de deux matrices A et B.

[58]:
```
def prodmat(A, B):
    p, q = nblig(A), nbcol(A)
    q1, r= nblig(B), nbcol(B)
    if q1 != q: raise Exception('Matrices incompatibles')
    C = [q * [0] for i in range(p)]
    for i in range(p):
        for j in range(r):
            for k in range(q):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

[59]:
```
A = exemple(8, 12)
prnt(A)
```

```
1  9 17 25 33 41 49 57 65 73 81 89
2 10 18 26 34 42 50 58 66 74 82 90
3 11 19 27 35 43 51 59 67 75 83 91
4 12 20 28 36 44 52 60 68 76 84 92
5 13 21 29 37 45 53 61 69 77 85 93
```

```
 6 14 22 30 38 46 54 62 70 78 86 94
 7 15 23 31 39 47 55 63 71 79 87 95
 8 16 24 32 40 48 56 64 72 80 88 96
```

[60]:
```
B = noyau(A)
prnt(B)
```

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

Le produit AB devrait être nul puisque (matriciellement) les colonnes de B sont dans le noyau de A

[61]:
```
prnt(prodmat(A, B))
```

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

CQFD :-).

1.5 4. Équations cartésiennes

1.5.1 4.1 Bases et équations cartésiennes

Soit F un sous-espace vectoriel de $\mathbb{R}^q$ de dimension r. Il existe essentiellement deux façons de se donner F.

- Par une base.
- Par un ensemble d'équations cartésiennes.

Deux questions se posent :

- Étant donnée une base de F, comment obtenir un ensemble minimal d'équations cartésiennes (la théorie nous annonce $q - r$ équations) ?

- Étant donné un ensemble minimal d'équations cartésiennes de F, comment en obtenir une base ?

Remarque : Nous allons bientôt être frappés de plein fouet par le problème des **matrices vides**. Imaginons que $F = \{0\}$. Une base de F est la famille vide, représentée par la matrice [] qui a zéro colonne. Oui, mais combien de lignes a-t-elle ? On ne sait pas. Et pourtant, cette information est vitale, puisqu'elle nous donne la dimension de l'espace E tout entier. Même problème avec les familles d'équations cartésiennes : L'espace tout entier est représenté par zéro équation, donc par une matrice de zéro ligne. Et nous ne savons plus quelle est sa dimension. Nous devrons donc, lors de l'écriture de nos fonctions, prévoir ce cas troublant.

1.5.2 4.2 Bases vers équations

La théorie nous dit que F, sev de dimension r d'un espace E de dimension q, est l'intersection de $q - r$ hyperplans $H_1, \dots, H_{q-r}$. Comment obtenir des équations de tels hyperplans ? Soit $\mathcal{F} = (x_1, \dots, x_r)$ une base de F. Soient $\varphi_1, \dots, \varphi_{q-r}$ $q - r$ formes linéaires non nulles dont les hyperplans cherchés sont les noyaux. Soit $f = (\varphi_1, \dots, \varphi_{q-r})$. L'appplication f est linéaire de E vers $\mathbb{R}^{q-r}$ et son noyau est précisément F. Notons A la matrice de f dans une base de E et la base canonique de $\mathbb{R}^{q-r}$. Notons B la matrice de $\mathcal{F}$ dans la base de E susnommée. On a alors $AB = 0$, ou encore

$$B^T A^T = 0$$

où le T désigne la transposition. Ainsi, les colonnes de A^T sont les matrices dans la base de E de vecteurs du noyau de B^T. Soit $g \in \mathcal{L}(\mathbb{R}^{q-r}, E)$ dont la matrice dans les bases (toujours les mêmes) de E et $\mathbb{R}^{q-r}$ est B^T. Alors A^T est la matrice d'une base du noyau de g.

Or, le théorème du rang nous dit que $\dim \ker g + rg\ g = \dim E = q$. Donc, $\dim \ker g = q - r$. Nous obtenons exactement ce que nous cherchions.

1.5.3 4.3 Mise en pratique

Conclusion de ce qui précède, nous savons déjà trouver des équations cartésiennes d'un sev. Il suffit d'utiliser la fonction `noyau`.

La fonction `base_vers_eq` ci_dessous prend en paramètre une matrice B de taille $q \times r$ censée représenter une base (une famille génératrice fonctionne aussi) d'un sous-espace vectoriel F de $\mathbb{R}^q$ de dimension r. Elle renvoie les coefficients d'une famille d'équations cartésiennes de F.

Remarque : Cette fonction **ne peut pas** fonctionner si la matrice B est vide, puisque dans ce cas on ne connaît pas la dimension de l'espace dans lequel on travaille. Voir une astuce dans les exemples qui suivent.

[62]:
```
def base_vers_eq(B):
    C = noyau(transp(B))
    return transp(C)
```

4.3.1 Exemples basiques Considérons la droite D de $\mathbb{R}^3$ engendrée par le vecteur $(1, 2, 3)$.

[63]:
```
A = transp([[1, 2, 3]])
prnt(base_vers_eq(A))
```

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

Une famille d'équations de D est donc

$$-2x + y = 0$$
$$-3x + z = 0$$

Reprenons notre exemple fétiche.

[64]:
```
A = exemple(3, 3)
prnt(base_vers_eq(image(A)))
```

```
1 -2 1
```

L'image de l'endomorphisme associé à A est le plan d'équation $x - 2y + z = 0$.

[65]:
```
A = exemple(5, 7)
B = noyau(A)
prnt(B)
print()
eq = base_vers_eq(B)
prnt(eq)
```

```
 1  2  3  4  5
-2 -3 -4 -5 -6
 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

 6  5  4  3  2 1 0
-5 -4 -3 -2 -1 0 1
```

Cette fois-ci, le noyau de l'application linéaire de $\mathbb{R}^7$ vers $\mathbb{R}^5$ associée à A est le sous-espace vectoriel de $\mathbb{R}^7$ de dimension 5 (deux équations !) d'équations

$$6x_1 + 5x_2 + 4x_3 + 3_4 + 2x_5 + x_6 = 0$$
$$5x_1 + 4x_2 + 3x_3 + 2_4 + x_5 - x_7 = 0$$

Une vérification ? Facile ...

[66]:
```
prnt(prodmat(eq, B))
```

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

L'espace tout entier est décrit par la famille vide d'équations cartésiennes. Cela fonctionne-t-il ?

[67]:
```
E = eye(5)
base_vers_eq(E)
```

[67]:
```
[]
```

Oui, mais impossible à partir de cela de retrouver E : nous avons perdu sa dimension.

À l'autre extrémité, quelles sont les équations de l'espace nul ? Une description de celui-ci par la base vide ne fonctionnera pas puisqu'on ne sait plus quelle est la dimension de l'espace entier. En revanche, en écrivant que le vecteur nul engendre l'espace nul, tout fonctionne.

[68]:
```
F = transp([[0, 0, 0, 0]])
prnt(base_vers_eq(F))
```

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

Le sev nul de $\mathbb{R}^4$ est donné par les équations $x = y = z = t = 0$, ce qui, avouons-le, est logique.

4.3.2 Un hyperplan de $\mathbb{R}^4$ Soit H l'hyperplan de $\mathbb{R}^4$ engendré par les vecteurs $(1, 4, 2, 7)$, $(8, 6, 5, 13)$ et $(7, 3, 11, 1)$. Trouvons une équation cartésienne de H.

[69]:
```
U = [1, 4, 2, 7]
V = [8, 6, 5, 13]
W = [7, 3, 11, 1]
B = transp([U, V, W])
Phi = base_vers_eq(B)
prnt(Phi)
```

```
-129 -399 173 197
```

Vérifions :

[70]:
```
prnt(prodmat(Phi, B))
```

```
0 0 0 0
```

1.5.4 4.4 Équations cartésiennes vers bases

Soit $M \in \mathcal{M}_{(q-r)q}(\mathbb{R})$ une matrice dont les lignes sont les coefficients des équations cartésiennes bans la base $\mathcal{B}$ d'un sev F de l'espace vectoriel E. Soit $x \in E$. Soit X la matrice de x dans la base $\mathcal{B}$. Le vecteur x est dans F si et seulement si $MX = 0$. En clair, trouver une base de F c'est

trouver une base du noyau d'une certaine application linéaire associée à M. En très clair, il n'y a quà utiliser la fonction `noyau`.

La fonction `eq_vers_base` prend en paramètre une matrice dont les ligne sont les coefficients des équations cartésiennes d'un sev F de E. Elle renvoie une base de F. En fait, `eq_vers_base` = `noyau`.

Remarque : Cette fonction **ne peut pas fonctionner** lorsque la famille d'équations est vide, c'est à dire lorsque le sev cherché est l'espace entier.

[71]:
```
def eq_vers_base(E):
    return noyau(E)
```

Soit, par exemple, l'hyperplan de $\mathbb{R}^4$ d'équation $2x + 3y + 6z + t = 0$.

[72]:
```
prnt(eq_vers_base([[2, 3, 6, 1]]))
```

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

Une base de F est $((-3, 2, 0, 0), (-3, 0, 1, 0), (-1, 0, 0, 2))$.

[73]:
```
A = exemple(6, 8)
B = noyau(A)
prnt(B)
```

```
 1  2  3  4  5  6
-2 -3 -4 -5 -6 -7
 1  0  0  0  0  0
 0  1  0  0  0  0
 0  0  1  0  0  0
 0  0  0  1  0  0
 0  0  0  0  1  0
 0  0  0  0  0  1
```

[74]:
```
E = base_vers_eq(B)
prnt(E)
```

```
 7  6  5  4  3  2 1 0
-6 -5 -4 -3 -2 -1 0 1
```

[75]:
```
B1 = eq_vers_base(E)
prnt(B1)
```

```
 1  2  3  4  5  6
-2 -3 -4 -5 -6 -7
 1  0  0  0  0  0
 0  1  0  0  0  0
```

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

1.6 5. Opérations sur les sous-espaces vectoriels

1.6.1 5.1 Introduction

Nous sommes maintenant capables de nous donner un sev F d'un espace vectoriel E : - Au moyen d'une base de F. - Au moyen d'une famille d'équations cartésiennes de F.

Dans ce qui suit nous allons voir qu'il est désormais facile de calculer des sommes, des intersections, de sev, ou de décider d'inclusions ou d'égalités de sev.

Remarque : Nous supposerons que nos sev **sont donnés par une famille génératrice**. S'ils sont donnés par une famille d'équations cartésiennes, il suffit d'appeler `base` pour en obtenir une base. Si l'on en veut une base et pas seulement une famille génératrice il suffit d'appeler la fonction `image`.

1.6.2 5.2 Dimension

La dimension d'un sev est le nombre de vecteurs d'une base. Remarquer le traitement séparé de l'espace nul.

[76]:
```
def dimension(F):
    if F == []: return 0
    else:
        return nbcol(image(F))
```

1.6.3 5.3 Somme

Soient F, G deux sev de E donnés par une famille génératrice. Il suffit d'accoler les deux matrices représentant F et G "l'une à côté de l'autre" et d'appeler la fonction `image` pour obtenir une base de $F + G$. On examine à part le cas où l'un des sev est nul.

[77]:
```
def somme(F, G):
    if F == []: return G
    elif G == []: return F
    else:
        A = transp(transp(F) + transp(G))
        return image(A)
```

[78]:
```
F = transp([[1, 1, 2], [1, 0, 3]])
G = transp([[1, -1, 1], [1, 2, 3]])
prnt(somme(F, G))
```

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

1.6.4 5.4 Intersection

Soient F, G deux sev de E donnés par une famille génératrice. Pour obtenir une base de $F \cap G$, il suffit de

- Déterminer avec la fonction `eq_cart` des familles d'équations cartésiennes de F et G.
- D'accoler ces matrices "l'une au-dessus de l'autre" : la matrice obtenue est la matrice d'une famille d'équations cartésiennes de $F \cap G$.
- D'appeler la fonction `base`.

On examine bien sûr à part le cas où l'un des deux espace est nul. Dans ce cas, l'intersection est l'espace nul. On renvoie la famille vide.

Il faut également régler le cas où $F = G = E$. Dans ce cas, la matrice A ci-dessous est vide, on renvoie alors la matrice identité, base de E, dont on connaît la taille en regardant F ou G.

[79]:
```
def intersect(F, G):
    if F == [] or G == []: return []
    else:
        A = base_vers_eq(F) + base_vers_eq(G)
        if A == []:
            if F != []: q = nblig(F)
            else: q = nblig(G)
            return eye(q)
        else:
            return eq_vers_base(A)
```

[80]:
```
F = transp([[1, 3, 3], [4, 5, 6]])
G = transp([[1, -1, 0],[1, 0, 2]])
prnt(intersect(F, G))
```

```
-8
11
 6
```

[81]:
```
F = []
G = transp([[1, -1, 0]])
prnt(intersect(F, G))
```

```
-
```

1.6.5 5.5 Inclusion, égalité

Soient F et G deux sev de E. Comment décider si $F \subset G$? C'est facile, puisque

$$F \subset G \iff \dim(F \cap G) = \dim F$$

[82]:
```
def inclus(F, G):
    return dimension(intersect(F, G)) == dimension(F)
```

[83]:
```
F = transp([[1, 1, 1]])
G = transp([[1, 2, 3], [4, 5, 6]])
inclus(F, G)
```

[83]: True

L'égalité de deux sev est alors immédiate.

[84]:
```
def egaux(F, G):
    return inclus(F, G) and dimension(F) == dimension(G)
```

[85]:
```
F = transp([[7, 8, 9], [10, 11, 12]])
G = transp([[1, 2, 3], [4, 5, 6]])
egaux(F, G)
```

[85]: True

Voici un dernier exemple où les deux espaces F et G sont égaux à $\mathbb{R}^5$, cas très particulier de la fonction `intersect`.

[86]:
```
F = exemple(5, 9)
G = exemple(5, 12)
egaux(F, G)
```

[86]: True

1.6.6 5.6 Somme directe, supplémentarité

Deux sev sont en somme directe si et seulement si leur intersection est $\{0\}$.

[87]:
```
def somme_directe(F, G):
    return dimension(intersect(F, G)) == 0
```

Deux sev sont supplémentaires si et seulement si - Ils sont en somme directe, et - La somme de leurs dimension est la dimension de l'espace.

Si F et G sont définis par la famille vide, il est impossible de trancher. En effet, si l'espace E est l'espace nul alors ils sont supplémentaires. Sinon ils ne le sont pas. Mais on n'a aucun moyen de connaître la dimension de E à partir des données !

Si, en revanche, l'un des deux sev est différent de $\{0\}$, la dimension de l'espace est le nombre de lignes de la matrice qui le définit.

[88]:
```
def dim_espace(F):
    if F == []: raise Exception('Indetermine')
    else: return nblig(F)
```

[89]:
```
def supplementaires(F, G):
    if F == [] and G == []: raise Exception('Indetermine')
    elif F == []: return dimension(G) == nblig(G)
    else:
        return somme_directe(F, G) and dimension(F) + dimension(G) ==␣
↪dim_espace(F)
```

[90]:
```
F = transp([[1, 2, 3]])
G = transp([[7, 5, 2], [6, 0, 4]])
supplementaires(F, G)
```

[90]:
```
True
```

1.6.7 5.7 Bilan

L'espace nul et l'espace tout entier nous ont donné un peu de fil à retordre dans les dernières fonctions que nous avons écrites. Mais nous avons réussi à nous en sortir sauf dans un seul et unique cas. Nous ne saurons hélas jamais si $\{0\} \oplus \{0\} = E$:-).