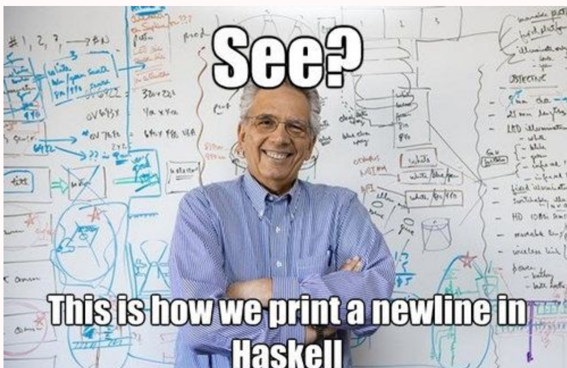
THÈME

2

# Pourquoi apprendre Haskell fait de moi un meilleur programmeur en Python ?




Comment notre travail sur les types et la programmation fonctionnelle peuvent nous aider à manipuler des images en Python et tout cela au grand dam' de GVR ?

# 1 Matrices

## 1 1 Préliminaire : Python est un langage fonctionnel ;-)

Vous avez une liste de nombres et une deuxième liste de nombres. Vous voulez retourner la liste des indices des éléments de la seconde dans la première.
Par exemple :

```
In [7]: recherche_liste([1,4,1,4,6,5,5,5,4,2,3], [1,3,5])
Out[7]: [[0, 2], [10], [5, 6, 7]]
```

Rien de plus simple :

```
recherche_liste = lambda l,e : [[ind for ind,el in enumerate(l) if el == elf] for elf in
    e]
```

Du Haskell ? Ben non, du Python :D
C'est quoi `enumerate` ?

```
In [16]: enumerate(['jan', 'fev', 'mar'])
Out[16]: <enumerate at 0x7f3e9089a8b8>

In [17]: list(enumerate(['jan', 'fev', 'mar']))
Out[17]: [(0, 'jan'), (1, 'fev'), (2, 'mar')]
```

On retrouve en Python `map`, `filter` et `reduce` qui est un synonyme de `foldl`.
Bref, pour bien programmer :

```
from fonctionnel import *
```

## 1 2 Fabriquons nos outils

### 1 2 1 Création d'une classe « Matrice »

Dans cette section, nous créerons nos matrices en donnant leur dimension et la fonction définissant leurs coefficients en fonction de leurs indices...comme nous l'avons fait en Haskell...sauf que c'est moins joli et mois contrôlable...

```
class Mat:
    """ une matrice sous la forme Mat([nb lignes, nb cols], fonction(i,j))"""

    def __init__(self, dim, f):
        self.F = f   # fonction (i,j) -> coeff en position i,j
        self.D = dim # liste [nb de lignes, nb de cols]
```

Par exemple, $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ sera creée avec :

```
>>> M = Mat([3,2],lambda i,j : 2*i + (j + 1))
```

On a quand même l'habitude de rentrer une matrice comme une liste de lignes. On va donc créer une fonction qui permet de convertir une liste de lignes en matrice :

```
def list2mat(mat):
    r,c = len(mat),len(mat[0])
    return Mat([r,c], lambda i,j : mat[i][j])
```

La matrice précédente aurait donc pu être définie par :

```
>>> M = list2mat([[1,2],[3,4],[5,6]])
```

```
>>> M.D # dimension de la matrice
[3, 2]
>>> M.F(0,1) # coefficient à la position 0,1
2
```

Recherche

Écrivez une fonction qui renvoie une matrice nulle de taille $n \times m$ puis une autre qui renvoie la matrice identité de taille $n$.

Nous rajoutons quelques méthodes habituellement définies pour les objets structurés de ce type :

```
   def __getitem__(self,cle):
        """ permet d'obtenir Mij avec M[i,j] """
        return self.F(*cle)

    def __iter__(self):
        """ pour itérer sur la liste des coefficients donnés par colonnes """
        [r,c] = self.D
        for j in range(c):
            for i in range(r):
                yield self.F(i,j)
```

L'emploi du mot-clé yield au lieu de return permet de ne *retourner* `self.F(i,j)` que lorsque cela est demandé. Ainsi la liste de tous les coefficients n'est pas créée en entier systématiquement ce qui économise la mémoire.
Nous pouvons ainsi créer une méthode qui va permettre un joli affichage :

```
    def __str__(self):
        """ joli affichage d'une matrice"""
        [r,c],f = self.D, self.F
        lmax = len(str(max(iter(self)))) + 1
        s = '\n'.join( (' '.join('{0:>.{1}G}'.format(f(i,j),l=lmax) for j in range(c)))
             ↪  for i in range(r))
        return s

    def __repr__(self):
        """ représentation dans le REPL"""
        return str(self)
```

La méthode join est à employer systématiquement pour la concaténation des chaînes de caractères : elle est beaucoup plus efficace que son pendant sous forme de boucle..

```
>>> M
1 2
3 4
5 6
```

Recherche

Créez deux fonctions qui créent un itérateur respectivement sur les lignes et sur les colonnes. Par exemple laliste des coefficients de la première colonne de M sera donnée par :

```
>>> [i for i in M.col(0)]
[1, 3, 5]
```

### 1.2.2 Transposée d'une matrice

La transposée d'une matrice $(M_{ij})_{\substack{1\leqslant i\leqslant n\\1\leqslant j\leqslant m}}$ est égale à la matrice $(M_{ji})_{\substack{1\leqslant i\leqslant n\\1\leqslant j\leqslant m}}$. Attention aux indices et aux nombre de lignes et de colonnes !

Recherche

Créez une fonction qui renvoie la transposée d'une matrice donnée en argument.

```
>>> M.transpose()
1 3 5
2 4 6
```

### 1.2.3 Somme de matrices

On voudrait créer une méthode qui prend comme arguments deux matrices et renvoie leur somme. Il faudra vérifier que les matrices à additionner sont de bonnes tailles : on utilisera la fonction assert qui est suivie d'une condition à vérifier et d'un message à affichher en cas de problème.
Pour pouvoir utiliser le symbole + par la suite, on va nommer notre méthode __add__ :

```
    def __add__(self,other):
        """ somme de deux matrices : utilisation du symbole +"""
        assert self.D == other.D, "tailles incompatibles"
        return Mat(self.D, lambda i,j : self.F(i,j) + other.F(i,j))
```

Par exemple, avec la matrice M précédemment introduite :

```
>>> M + M
  2   4
  6   8
 10  12
```

Recherche

On définit de même __neg__ pour l'opposé et __sub__ pour la soustraction : faites-le !

### 1.2.4 Produit par un scalaire

On voudrait obtenir la matrice $k \cdot M$ à partir d'une matrice $M$ et d'un scalaire $k$.

```
>>> A = Mat([2,3], lambda i,j : 3*i + j)
>>> A
 0  1  2
 3  4  5
>>> A.prod_par_scal(5)
  0   5  10
 15  20  25
```

Recherche

Écrivez une implémentation de la méthode prod_par_scal

### 1.2.5 Produit de matrices

Soit $A = (a_{ij})_{\substack{1\leqslant i\leqslant n\\1\leqslant i\leqslant m}}$ et $B = (b_{ij})_{\substack{1\leqslant i\leqslant m\\1\leqslant i\leqslant p}}$. Alors $A \times B = C$ avec $C = (c_{ij})_{\substack{1\leqslant i\leqslant n\\1\leqslant i\leqslant p}}$ et

$$\forall (i,j) \in \mathbb{N}_n^* \times \mathbb{N}_p^*, \quad c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

C'est l'algorithme habituel trois boucles imbriquées).

Nous allons l'abstraire d'un petit niveau : chaque coefficient $c_ij$ est en fait égal au produit scalaire de la ligne $i$ de A et de la colonne $j$ de B.
Commençons donc par construire une fonction qui calcule le produit scalaire de deux itérables.
Pour cela, nous allons utiliser une fonction extrêmement importante qui vient du monde de la programmation fonctionelle : map.
`map(fonc, iterable(s))` renvoie un itérateur de type `map` mais où chaque élément sera remplacé par son image par la fonction (si c'est une fonction de une variable) ou par l'image combinée des itérables si c'est une fonction de plusieurs variables.
Par exemple :

```
>>> map(lambda x: x*x, [1,2,3,4])
<map at 0x7fe8307f2ef0>
>>> [i for i in m]
[1 4 9 16]
```

arghhh : l'objet crée est de type `map` et on a son adresse mais on ne sait pas ce qu'il y a dedans...

```
>>> m = map(lambda x: x*x, [1,2,3,4])
>>> [i for i in m]
[1 4 9 16]
```

Mais atention ! m a maintenant été « consommé ». Si on en redemande :

```
>>> [i for i in m]
[]
```

Si on veut en garder une trace on peut utiliser par exemple list :

```
>>> m = map(lambda x: x*x, [1,2,3,4])
>>> l = list(m)
>>> l
[1 4 9 16]
```

Avec une fonction de deux variables et deux itérables :

```
>>> m = map(lambda x, y: x + y, [1,2,3,4], [-1,-2,-3,-4])
>>> list(m)
[0, 0, 0, 0]
```

On peut aller plus vite en utilisant les opérateurs arithmétiques écrits en notation préfixée dans la bibliothèque operator :

```
from operator import mul
>>> m = map(mul, [1,2,3,4], [-1,-2,-3,-4])
>>> list(m)
[-1, -4, -9, -16]
```

Pour notre produit scalaire, nous utilisons également la classique fonction sum.

Recherche

Créez une fonction `prod_scal`.
Utilisez `prod_scal` pour écrire en une ligne la matrice produit de deux matrices données en arguments.
Vous créerez ainsi une méthode `prod_mat(self,other)` en ajoutant à la ligne précédente une ligne vérifiant que les formats sont corrects.

Pour utiliser le symbole $*$ pour le produit d'une matrice par une autre matrice ou un scalaire, on crée dans notre classe une méthode `__mul__`. On utilisera la fonction type qui teste le type d'un objet.

```
1   def __mul__(self,other):
2       """ produit d'une matrice par un scalaire ou une matrice : utilisation du
         ↪  symbole *"""
3       if Mat == type(other):
4           return self.prod_mat(other)
5       else:
6           return self.prod_par_scal(other)
```

### 1 2 6 Comparaison de plusieurs produits matriciels

Notre produit n'est pas si mal : il s'écrit très simplement et est assez efficace. Comparons-le avec d'autres implémentations.

#### Avec les outils standards de Python

D'abord une implémentation classique en introduisant une liste Python et des matrices du type `array` de numpy. On travaillera par exemple avec des tableaux d'entiers (l'argument `dtype = int`) et on utilisera les commandes `shape` et `zeros` de numpy.

```
1  import numpy as np
2
3  def pymatmatprod(A, B):
4      '''
5      Multiplication matricielle avec les outils Python usuels
6      '''
7      ra, ca = A.shape
8      rb, cb = B.shape
9      assert ca == rb, "Tailles incompatibles"
10     C = np.zeros((ra, cb),dtype = int)
11     for i in range(ra):
12         Ai,Ci = A[i],C[i]
13         for j in range(cb):
14             for k in range(ca):
15                 Ci[j] += Ai[k] * B[k, j]
16     return C
```

#### Avec Cython

Cython (http://cython.org/) est un langage dont la syntaxe est très proche de celle de Python mais c'est un langage qui permet de générer des exécutables compilés et d'utiliser un style de programmation proche du C. Cela permet d'optimiser grandement Python.
Le logiciel de calcul formel Sage http://www.sagemath.org/ est principalement écrit en Cython pour gagner en efficacité.
Voyons un exemple en œuvre. L'usage conjoint de Python et Cython est grandement facilité par le travail dans l'environnement iPython.

```
1  In [1]: %load_ext cythonmagic
2
3  In [2]:
4  %%cython
5  cimport cython
6  import numpy as np
7  cimport numpy as np
8
9  cdef long[:, :] matprodC( long[:, :]  A, long[:, :]  B):
10     '''
11     multiplication matricielle via cython et les array de numpy
12     '''
13     cdef:
14         int i, j, k
15         int ra = A.shape[0]
16         int ca = A.shape[1]
```

```
17          int rb = B.shape[0]
18          int cb = B.shape[1]
19          long [:, :] C
20          long [:] Ai, Ci
21
22      assert ca == rb, 'Tailles non compatibles'
23
24      C = np.zeros((ra, cb),dtype=int)
25      for i in range(ra):
26          Ai,Ci = A[i],C[i]
27          for j in range(cb):
28              for k in range(ca):
29                  Ci[j] = Ci[j] + Ai[k] * B[k, j]
30      return C
31
32  def matprod(long [:, :]  A, B):
33      return matprodC(A, B)
```

On remarque qu'écrire en Cython revient un peu à écrire en Python mais en déclarant des variables comme en C.
iPython permet également de comparer facilement les temps de calcul avec la commande magique `%timeit`.
On commence par créer une matrice de taille $200 \times 200$ puis on va demander son coefficient [100,100] :

```
1  In [3]: A = np.ones((200,200),dtype = int)
```

On compare alors le produit via Python, Cython et la multiplication de `numpy` :

```
1  In [4]: %timeit matprod(A,A)[100,100]
2  100 loops, best of 3: 16.3 ms per loop
3
4  In [5]: %timeit pymatmatprod(A,A)[100][100]
5  1 loops, best of 3: 8.35 s per loop
6
7  In [6]: %timeit np.dot(A,A)[100,100]
8  100 loops, best of 3: 8.94 ms per loop
```

Notre fonction en Cython est 500 fois plus rapide !!! Elle est très légèrement plus lente que le produit de `numpy`.
Et notre implémentation personnelle avec la classe `Mat` ?

```
1  In [7]: A = Mat([200,200],lambda i,j:1)
2
3  In [8]: %timeit (A*A)[100,100]
4  10000 loops, best of 3: 79.6 µs per loop
```

Trop fort ! Nous battons tout le monde à plate couture ! Nous sommes 100 fois plus rapides que `numpy` !!!
...et nous avons bien le bon résultat :

```
1  In [9]: (A*A)[100,100]
2  Out[9]: 200
```

Bon, ça va pour un seul produit. Pour calculer une puissance 200, ce sera moins magique...

**1 2 7 Puissances d'une matrice**

Par exemple, avec $J = \left(\begin{smallmatrix}1 & 1\\ 1 & 1\end{smallmatrix}\right)$ :

```
>>> J = Mat([2,2], lambda i,j : 1)
>>> J ** 5
 16  16
 16  16
```

En effet, $J^n = 2^{n-1}J$ est un résultat classique. On utilise un algorithme d'exponentiation rapide. Par exemple, en voici une version récursive rapide à écrire qui se nomme __pow__ afin d'utiliser le symbole ** :

```
    def __pow__(self,n):
        r = self.D[0]
        if n == 0:
            return unite(r)
        def pui(m,k,acc):
            if k == 0:
                return acc
            return pui((m*m),k//2,acc if k % 2 == 0 else (m*acc))
        return pui(self,n,unite(r))
```

Recherche

Déterminez une version impérative de la méthode précédente.

## 2 Rappels sur l'inversion des matrices

**2 0 8 Matrice carrée inversible**

Soit $M \in \mathbb{A}^{n\times n}$. M est inversible (régulière) si, et seulement si, il existe une matrice N dans $\mathbb{A}^{n\times n}$ telle que :

$$M \times N = N \times M = \mathbb{I}_n$$

On note alors $N = M^{-1}$.

On remarque (n'est-ce pas) que, d'après cette définition, $(M^{-1})^{-1} = M$.

Lorsque nous aurons étudier les déterminants, nous pourrons démontrer que si A et B sont deux matrices carrées de taille $n$ vérifiant $A \times B = \mathbb{I}_n$, alors elles sont régulières et $A = B^{-1}$.

Recherche

Si A et B sont régulières et de taille $n$, alors comment calculer $(A\times B)^{-1}$ à partir des inverses de A te B ?

**2 0 9 Opérations sur les lignes**

Matrices élémentaires

Nous désignerons par $E_n^{ij}$ la matrice carrée de $\mathbb{A}^{n\times n}$ dont tous les coefficients sont nuls sauf le coefficient $(i, j)$ qui vaut $1_{\mathbb{A}}$. Par exemple, $E_3^{12} = \left(\begin{smallmatrix}0 & 1 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{smallmatrix}\right)$ dans $\mathbb{Z}^{3\times 3}$.

Leoplod KRONECKER (1823-1891)

Leopold KRONECKER est un mathématicien prussien né dans l'actuelle Pologne. On lui doit la célèbre citation : « *Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk* ».
En son honneur, on a donné son nom à la fonction suivante :

$$\delta : \begin{array}{ccc} \mathbb{N}\times\mathbb{N} & \to & \{0;1\} \\ (i,j) & \mapsto & 1 \text{ si } i=j,\ 0 \text{ sinon} \end{array}$$

On condense souvent la notation en $\delta_{ij}$ et on parle alors de **symbole de Kronecker**. Par exemple, la matrice identité peut être définie par :

$$\mathbb{I}_n = (\delta_{ij})_{\substack{1\leqslant i\leqslant n\\ 1\leqslant j\leqslant n}}$$

Recherche

Étudiez le produit $E_n^{ij}\times E_n^{\ell k}$ et exprimez-le à l'aide du symbole de KRONECKER.
Simplifiez ensuite le produit $(\mathbb{I}_n+\lambda E_n^{ij})\times(\mathbb{I}_n-\lambda E_n^{ij})$ : qu'en concluez-vous ?

Transvections de lignes
On s'intéresse à la fonction :

$$T_\lambda^{ij} : \begin{array}{ccc} \mathbb{A}^{n\times p} & \to & \mathbb{A}^{n\times p} \\ M & \mapsto & (\mathbb{I}_n+\lambda E_n^{ij})\times M \end{array}$$

Calculez par exemple l'image par $T_\lambda^{23}$ de $\begin{pmatrix} a_1 & b_1 & c_1 & d_1\\ a_2 & b_2 & c_2 & d_2\\ a_3 & b_3 & c_3 & d_3\end{pmatrix}$
Ainsi, $T_\lambda^{ij}(M)$ permet d'obtenir la matrice construite à partir de M en remplaçant la ligne $i$ par elle-même plus $\lambda$ fois la ligne $j$.
On note plus commodément cette transformation $L_i \leftarrow L_i \boxplus (\lambda \boxdot L_j)$.
Vous aurez bien noté que $(\mathbb{I}_n+\lambda E_n^{ij})^{-1}=\mathbb{I}_n-\lambda E_n^{ij}$.
Dilatations de lignes
On veut effectuer l'opération $L_i \leftarrow \lambda \boxdot L_i$. On note

$$\Delta_n^{i,\lambda} = \mathbb{I}_n + (\lambda \boxplus (-1_{\mathbb{A}}))E_n^{ii}$$

Calculez par exemple le produit de $\Delta_3^{2,\lambda}$ par $\begin{pmatrix} a_1 & b_1 & c_1 & d_1\\ a_2 & b_2 & c_2 & d_2\\ a_3 & b_3 & c_3 & d_3\end{pmatrix}$
Le produit par $\Delta_n^{i,\lambda}$ est une dilatation de ligne.
Échange de lignes
On considère la matrice $S_n^{ij}=\Delta_n^{j,-1_{\mathbb{A}}}\times(\mathbb{I}_n+E_n^{ij})\times(\mathbb{I}_n-E_n^{ji})\times(\mathbb{I}_n+E_n^{ij})$.
Développez ce produit. Que vaut le produit de $S_3^{23}$ par $\begin{pmatrix} a_1 & b_1 & c_1 & d_1\\ a_2 & b_2 & c_2 & d_2\\ a_3 & b_3 & c_3 & d_3\end{pmatrix}$ ?
On note cette opération $L_i \leftrightarrow L_j$.
Opérations sur les lignes
De manière plus générale, une fonction $\varphi$ de $\mathbb{A}^{n\times p}$ dans lui-même est une **opération sur les lignes** si c'est la composée finie de transvections et de dilatations de lignes.
L'image d'une matrice par $\varphi$ est donc le produit à gauche par des matrices de transvections ou de dilatations :

$$\varphi : M \mapsto (F_k\times F_{k-1}\cdots\times F_1)\times M$$

avec chaque $F_i$ inversible. La fonction $\varphi$ est donc elle-même totale bijective et sa réciproque est :

$$\varphi^{-1} : N \mapsto (F_1^{-1}\times F_2^{-1}\cdots\times F_k^{-1})\times N$$

On en déduit en particulier que l'inverse de $\varphi(\mathbb{I}_n)$ existe et que c'est $\varphi^{-1}(\mathbb{I}_n)$.
Lien avec les matrices régulières
Voici un théorème important qui nous sera très utile pour calculer l'inverse d'une matrice régulière.

Théorème 2 - 1

$\varphi$ étant une opération élémentaire sur les lignes,

$$M \text{ inversible} \leftrightarrow \varphi(M) \text{ inversible}$$

En effet nous savons que $\varphi(M) = \varphi(\mathbb{I}_n) \times M$. Si $M$ est inversible, $\varphi(M)$ s'exprime comme le produit de deux matrices inversibles et elle est donc inversible. Supposons maintenant $\varphi(M)$ inversible, comme $\varphi(\mathbb{I}_n)$ est inversible on obtient :

$$\varphi(\mathbb{I}_n)^{-1} \times \varphi(M) = \varphi(\mathbb{I}_n)^{-1} \times (\varphi(\mathbb{I}_n) \times M)$$

qui se transforme en

$$\varphi(\mathbb{I}_n)^{-1} \times \varphi(M) = \left(\varphi(\mathbb{I}_n)^{-1} \times \varphi(\mathbb{I}_n)\right) \times M = \mathbb{I}_n \times M$$

et en définitive

$$M = \varphi(\mathbb{I}_n)^{-1} \times \varphi(M) = \varphi^{-1}(\mathbb{I}_n) \times \varphi(M)$$

$M$ s'exprimant comme le produit de deux matrices inversibles est inversible.

**Théorème 2 - 2**

Si $\varphi_1, \varphi_2, ...., \varphi_k$ est une suite d'opérations sur les lignes de $M$ qui transforme $M$ en $\mathbb{I}_n$ alors $M$ est inversible et

$$M^{-1} = \varphi_k(\mathbb{I}_n) \times \varphi_{k-1}(\mathbb{I}_n) \times .... \times \varphi_1(\mathbb{I}_n) = \varphi_k \circ \varphi_{k-1} \circ ... \circ \varphi_1(\mathbb{I}_n)$$

En effet, si nous avons $\varphi_k \circ \varphi_{k-1} \circ ... \circ \varphi_1(M) = \mathbb{I}_n$ alors

$$\varphi_k \circ \varphi_{k-1} \circ ... \circ \varphi_1(M) = \mathbb{I}_n = (\varphi_k(\mathbb{I}_n) \times \varphi_{k-1}(\mathbb{I}_n) \times .... \times \varphi_1(\mathbb{I}_n)) \times M$$

Nous avons trouvé une matrice carrée qui multipliant $M$ donne $\mathbb{I}_n$, c'est son inverse.
Le théorème précédent nous indique une méthode pour trouver l'inverse de $M$ (s'il existe), nous allons étudier plus en détail cette technique et nous démontrerons plus loin que s'il est impossible de transformer $M$ en $\mathbb{I}_n$ en utilisant les opérations élémentaires sur les lignes, alors $M$ n'est pas inversible.

## 2 1 Rang d'une matrice

Dans tout ce qui suit nous allons utiliser les opérations élémentaires sur les lignes d'une matrice et on travaille dans $\mathbb{A}^{n\times p}$.

### 2 1 1 Matrices ligne-équivalentes

Nous dirons que deux matrices $M$ et $N$ sont **ligne-équivalentes** si, et seulement si, il existe une suite finie $(\varphi_i)_{i\in\mathbb{N}_k}$ d'opérations élémentaires sur les lignes de sorte que

$$N = \varphi_k \circ \varphi_{k-1} \circ ... \circ \varphi_1(M)$$

Nous écrirons alors $M \overset{\ell}{\equiv} N$. La relation $\overset{\ell}{\equiv}$ est manifestement une relation d'équivalence sur $\mathbb{A}^{n\times p}$, on démontre sans peine que

$$M \overset{\ell}{\equiv} M$$
$$M \overset{\ell}{\equiv} N \to N \overset{\ell}{\equiv} M$$
$$\left(M \overset{\ell}{\equiv} N \text{ et } N \overset{\ell}{\equiv} C\right) \to M \overset{\ell}{\equiv} C$$

Nous savons que $\varphi_k \circ \varphi_{k-1} \circ ... \circ \varphi_1(M) = \varphi_k \circ \varphi_{k-1} \circ ... \circ \varphi_1(\mathbb{I}_n) \times M$, par conséquent nous aurons $M \overset{\ell}{\equiv} N$ s'il existe une matrice $R$ inversible vérifiant

$$N = R \times M$$

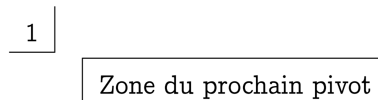
$R$ étant le résultat du produit de matrices du type $\varphi(\mathbb{I}_n)$. Pour obtenir cette matrice $R$ il suffit d'appliquer successivement en parallèle les opérations élémentaires qui transforment $M$ en $N$ sur $\mathbb{I}_n$. Pour cela on utilise un tableau où l'on juxtapose $M$ et $\mathbb{I}_n$, $M$ étant la partie gauche et $\mathbb{I}_n$ la partie droite de ce tableau. Toute opération élémentaire sur les lignes effectuée sur la partie gauche est simultanément effectuée sur la partie droite.

| opérations $\varphi$ | Partie gauche | Partie droite | Remarques |
|---|---|---|---|
| | $M$ | $\mathbb{I}_n$ | initialisation du tableau |
| $\varphi_1$ | $M_1$ | $R_1$ | $M_1 = \varphi_1(M)$, $R_1 = \varphi_1(\mathbb{I}_n)$ |
| $\varphi_2$ | $M_2$ | $R_2$ | $M_2 = \varphi_2(M_1)$, $R_2 = \varphi_2(R_1)$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $\varphi_i$ | $M_i$ | $R_i$ | $M_i = R_i \times M$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $\varphi_k$ | $N$ | $R$ | $N = R \times M$ |

### 2 1 2 L réduite échelonnée

La matrice $M = (m_{ij}) \in \mathbb{A}^{n\times p}(\mathbb{K})$ est dite **$\ell$-réduite** (**$\ell$** pour « ligne ») si, et seulement si, elle satisfait aux conditions suivantes :

1. Toutes les lignes nulles (une ligne est nulle si elle ne comporte que des zéros) sont au-dessous des lignes non nulles.
2. Dans chaque ligne non nulle le premier élément non nul est $1_{\mathbb{A}}$ (on lit une ligne de la gauche vers la droite), ce $1_{\mathbb{A}}$ est appelé **pivot** ou élément pivot. La colonne où se trouve ce $1_{\mathbb{A}}$ est appelée colonne pivot et c'est le seul élément non nul de cette colonne.
3. Si, **de plus**, les pivots apparaissent en ordre croissant par numéro de ligne **et** numéro de colonne, on dit que $M$ est **$\ell$-réduite échelonnée** (en abrégé lré ou LRé).




Donnons quelques exemples de matrices entières :

$$\begin{pmatrix} 0 & \boxed{1} & 2 & 0 \\ \boxed{1} & 0 & 3 & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ est } \ell\text{-réduite non échelonnée}$$

$$\begin{pmatrix} 0 & \boxed{1} & 0 \\ 0 & 2 & 0 \\ 0 & 0 & \boxed{1} \end{pmatrix} \text{ n'est pas } \ell\text{-réduite}$$

$$\begin{pmatrix} \boxed{1} & -2 & 0 \\ 0 & 0 & \boxed{1} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ est } \ell\text{-réduite échelonnée}$$

**Théorème 2 - 3**

Pour toute matrice $M \in \mathbb{A}^{n\times p}$, il existe une unique matrice $\ell$-réduite échelonnée $N \in \mathbb{A}^{n\times p}$ telle que $M \overset{\ell}{\equiv} N$, $N$ est appelée la $\ell$-réduite échelonnée de $M$ que l'on peut noter par $N = \text{lré}(M)$.

La démonstration de ce théorème se fait par récurrence sur $n$ et elle est un peu difficile.
Le **rang** de $M$, noté $\text{rang}(M)$, est le nombre de lignes non nulles de sa $\ell$-réduite échelonnée, c'est donc aussi le nombre de pivots da sa lré. $M$ est dite de plein rang si son rang est égal à son nombre de lignes.
Nous énoncons les évidences (conséquences directes de la définition) :

1. Si $M \in \mathbb{A}^{n\times p}$ le rang de $M$ est inférieur ou égal à $n$ **et** à $p$.

**2.** Si $M \in \mathbb{A}^{n\times n}$ et si $\operatorname{rang}(M) = n$ alors sa $\ell$-réduite échelonnée est $\mathbb{I}_n$ et nous avons précédemment démontré que dans ce cas $M$ est inversible.

**3.** Deux matrices ligne-équivalentes ont la même $\ell$-réduite échelonnée et ont donc même rang.

**4.** Si $M'$ est une matrice extraite de $M$, on a $\operatorname{rang}(M') \leq \operatorname{rang}(M)$.

### 2 2 Algorithme Fang-Tcheng

Wilhelm Jordan (1842-1899)

Notre eurocentrisme préfère nommer cet algorithme Gauss-Jordan...
Nous allons le présenter sur un exemple. Soit à chercher la $\ell$-réduite échelonnée de la matrice

$$A = \begin{pmatrix} 2 & 5 & -2 & 3 \\ 3 & 6 & 3 & 6 \\ 1 & 2 & -1 & 2 \end{pmatrix}$$

Nous allons faire apparaître les pivots ordonnés par numéro de ligne et numéro de colonne, cela veut dire que si un pivot (qui sera toujours égal à 1) est obtenu à la ligne $i$ et colonne $j$, le pivot suivant sera au moins en ligne $i+1$ et en colonne $j+1$ et on s'imposera de trouver la solution minimale en numéro de ligne et numéro de colonne.

— **Première étape.** On repère l'élément de la première colonne qui est le plus grand en valeur absolue (il peut y avoir plusieurs choix). Si le résultat est nul (la première colonne ne contient que des zéros), la première colonne ne peut être une colonne pivot et on passe à la colonne suivante. Ici c'est 3 qui se trouve sur la deuxième ligne première colonne. On permute alors la première ligne avec la deuxième ligne et on obtient

$$A_1 = \begin{pmatrix} 3 & 6 & 3 & 6 \\ 2 & 5 & -2 & 3 \\ 1 & 2 & -1 & 2 \end{pmatrix}$$

On divise tous les éléments de la première ligne par 3 :

$$A_2 = \begin{pmatrix} \boxed{1} & 2 & 1 & 2 \\ 2 & 5 & -2 & 3 \\ 1 & 2 & -1 & 2 \end{pmatrix}$$

On fait apparaître ensuite des zéros sous le premier 1 de la première colonne en utilisant les opérations $L_2 \leftarrow L_2 - 2L_1$ et $L_3 \leftarrow L_3 - 1L_1$ :

$$A_3 = \begin{pmatrix} \boxed{1} & 2 & 1 & 2 \\ 0 & 1 & -4 & -1 \\ 0 & 0 & -2 & 0 \end{pmatrix}$$

La première colonne est une colonne pivot et elle ne devra pas être modifiée par la suite.

— **Deuxième étape.** On repère dans la deuxième colonne (en fait la colonne qui suit la dernière colonne pivot obtenue), à partir de la deuxième ligne (en fait à partir du numéro de ligne qui suit le numéro de la ligne qui a donné le dernier pivot), le plus grand élément en valeur absolue (si on obtient zéro on passe à la colonne suivante, etc⋯). Ici on obtient 1 qui se trouve sur la deuxième ligne et deuxième colonne et il n'y a pas de permutation de lignes à faire. Maintenant il faut faire apparaître des zéros dans la colonne pivot en ligne 1 et en ligne 3. On utilise les opérations $L_1 \leftarrow L_1 - 2L_2$ et $L_3 \leftarrow L_3 - 0L_2$ (qui ne sert à rien) ; on obtient :

$$A_4 = \begin{pmatrix} \boxed{1} & 0 & 9 & 4 \\ 0 & \boxed{1} & -4 & -1 \\ 0 & 0 & -2 & 0 \end{pmatrix}$$

La deuxième colonne est une colonne pivot, elle ne devra pas être modifiée dans la suite.

— **Troisième étape.** On repère dans la colonne qui suit la dernière colonne pivot obtenue et à partir de la ligne qui suit la ligne qui a donné le dernier pivot le plus grand élément en valeur asolue. Ici c'est -2 qui se trouve sur la troisième ligne et troisième colonne, la troisième colonne est alors la troisième colonne pivot. Il n'y a pas de permutation de lignes à faire, nous n'avons qu'à faire apparaître un 1 à la place de -2 puis faire apparaître des zéros dans la colonne pivot en conservant évidemment le pivot 1. Appliquons à $A_4$ l'opération $L_3 \leftarrow -\frac{1}{2}L_3$

$$A_5 = \begin{pmatrix} \boxed{1} & 0 & 9 & 4 \\ 0 & \boxed{1} & -4 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

puis appliquons $L_1 \leftarrow L_1 - 9L_3$ et $L_2 \leftarrow L_2 + 4L_3$

$$A_6 = \begin{pmatrix} \boxed{1} & 0 & 0 & 4 \\ 0 & \boxed{1} & 0 & -1 \\ 0 & 0 & \boxed{1} & 0 \end{pmatrix}$$

Nous venons d'obtenir la $\ell$-réduite échelonnée de $A$, $A$ est de rang 3.

Nous avons utilisé, ici, la méthode dite du pivot partiel en cherchant dans chaque colonne le plus grand élément en valeur absolue. Si nous avions choisi de prendre le premier élément rencontré non nul, en vertu de l'unicité de la $\ell$-réduite échelonnée, nous aurions obtenu le même résultat final. Il est conseillé, en programmation, d'utiliser la méthode dite pivot partiel pour éviter une trop grande propagation des erreurs lors des divisions par des petits nombres. Si on fait les calculs à la main il vaut mieux se contenter de choisir, tant que c'est possible, des nombres sympathiques.

**Théorème 2 - 4** $A$ est une matrice carrée d'ordre $n$ inversible si, et seulement si, le rang de $A$ est égal à $n$.

Nous avons en fait déjà démontré une partie de ce théorème mais, vu son importance, nous allons reprendre ce qui a été dit. Supposons donc que le rang de $A$ est égal à $n$, dans ces conditions la $\ell$-réduite échelonnée de $A$ est $\mathbb{I}_n$, cela signifie que $A \overset{\ell}{\equiv} \mathbb{I}_n$ et donc qu'il existe $A'$ vérifiant $A' \times A = \mathbb{I}_n$, $A'$ est l'inverse de $A$. Supposons maintenant que la rang de $A$ est strictement inférieur à $n$, il est alors sûr que la dernière ligne de la $\ell$-réduite échelonnée de $A$ est une ligne nulle. Notons $B$ cette $\ell$-réduite échelonnée, nous savons que $A$ inversible équivaut à écrire que $B$ est inversible puisque $A \overset{\ell}{\equiv} B$. Supposons $B$ inversible, il existe alors $B'$ vérifiant $B \times B' = \mathbb{I}_n$ or cette égalité est impossible car la dernière ligne de $B$ étant nulle, la dernière ligne du produit $B \times B'$ sera aussi nulle et donc distincte de la dernière ligne de $\mathbb{I}_n$. Nous venons de démontrer

$$\mathrm{rg}(A) < n \to A \text{ non inversible}$$

et donc $A$ inversible $\to \mathrm{rg}(A) = n$.

**Pratique** : pour rechercher la matrice inverse de $A$, si elle existe, on applique simultanément sur $\mathbb{I}_n$ les opérations faites pour déterminer la $\ell$-réduite échelonnée de $A$. Pour cela on considère le tableau

$$[A \,|\, \mathbb{I}_n]$$

on applique l'algorithme Fang Tcheng tableau, et chaque opération faite sur la partie gauche est reproduite sur la partie droite. Si la $\ell$-réduite de $A$ est la matrice $\mathbb{I}_n$ (le résultat de la partie gauche est $\mathbb{I}_n$) alors $A$ est inversible et sa matrice inverse est donnée par la partie droite. Si la partie gauche ne donne pas $\mathbb{I}_n$, $A$ n'est pas inversible, son rang est strictement inférieur à $n$.

*Licence Creative Commons*

— Si $\text{rang}(T) = \text{rang}(R) = r$, c'est que $T' = T''$ et la résolution de $A \times X = B$ équivaut à la résolution des $r$ équations non nulles de $R \times X = H$. Deux cas peuvent se présenter :

**1.** $r = p$, le système à résoudre est alors de la forme :

$$\begin{cases} x_1 & & & = h_1 \\ & x_2 & & = h_2 \\ & & \vdots & \vdots \\ & & & x_p = h_p \end{cases}$$

et il est résolu, il y a unicité de la solution.

**2.** $r < p$. Nous appelons inconnues pivots ou inconnues principales $(IP)$ les $r$ inconnues $x_{j_1}, x_{j_2}, \cdots, x_{j_r}$ correspondant aux $r$ colonnes pivots $j_1, j_2, \cdots, j_r$ de $R$. Les $p - r$ autres inconnues sont appelées inconnues non principales $(INP)$, certains les appellent aussi des inconnues ou des variables libres ou encore des paramètres. Pour résoudre le système la méthode consiste à faire passer dans les seconds membres les inconnues non principales, le système devient alors un sytème de Cramer résolu dont les solutions dépendent des $p - r$ inconnues non principales (ces inconnues non principales sont, à ce stade, considérées comme des paramètres), **il y a donc une infinité de solutions** ; si l'on désire une solution particulière, il suffit de donner des valeurs arbitraires aux inconnues non principales.

$$\begin{cases} x_{j_1} & & & = h_{j_1} + e_{j_1} \\ & x_{j_2} & & = h_{j_2} + e_{j_2} \\ & & \vdots & \vdots \\ & & & x_{j_r} = h_{j_r} + e_{j_r} \end{cases}$$

où les $e_{j_i}$ sont des expressions linéaires des $(p - r)$ $INP$.

Il faut remarquer que si l'on modifie l'ordre d'écriture des inconnues nous n'obtiendrons pas forcément les mêmes inconnues principales et non principales mais l'ensemble solution sera évidemment le même.

# 3 Back to code

## 3 1 Inverse d'une matrice par la méthode de Gauss-Jordan

### 3 1 1 Opérations élémentaires

Pour effectuer une combinaison linéaire des lignes, on va créer une fonction :
`comb_ligne(ki,kj,M,i,j)`
qui renverra la matrice construite à partir de M en remplaçant la ligne $L_j$ par $k_j \times L_j + k_i \times L_i$.

```
1    def comb_lignes(self,ki,kj,i,j):
2        """Li <- ki*Li + kj * Lj"""
3        f = self.F
4        g = lambda r,c : ki*f(i,c) + kj*f(j,c) if r == i else f(r,c)
5        return Mat(self.D,g)
```

Recherche

On crée également une fonction `mult_ligne(k,M,j)`: qui renverra la matrice construite à partir de M en remplaçant la ligne $L_j$ par $k \times L_j$.

### 3 1 2 Calcul de l'inverse étape par étape

On commence par créer une fonction qui renvoie une matrice où se juxtaposent la matrice initiale et la matrice identité :
Créons une matrice :

```
>>> M = list2mat([[1,0,1],[0,1,1],[1,1,0]])
```

$$M = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Complétons-la par la matrice identité de même taille :

```
>>> T = M.cat_carre_droite()
>>> T
 1  0  1  1  0  0
 0  1  1  0  1  0
 1  1  0  0  0  1
```

Recherche

Complétez...

```
def cat_carre_droite(self):
    """
    Colle l'identité à droite pour la méthode de GJ

    """
    ???
```

On effectue $L_3 \leftarrow L_3 - L_1$ :

```
>>> T = T.comb_lignes(1,-1,2,0)
>>> T
 1  0  1  1  0  0
 0  1  1  0  1  0
 0  1 -1 -1  0  1
```

puis $L_3 \leftarrow L_3 - L_2$ :

```
>>> T = T.comb_lignes(1,-1,2,1)
>>> T
 1  0  1  1  0  0
 0  1  1  0  1  0
 0  0 -2 -1 -1  1
```

puis $L_3 \leftarrow \frac{1}{2} L_3$ :

```
>>> T = T.mult_ligne(-0.5,2)
>>> T
 1  0  1  1    0    0
 0  1  1  0    1    0
 0  0  1  0.5 0.5 -0.5
```

On effectue $L_1 \leftarrow L_1 - L_3$ :

```
>>> T = T = T.comb_lignes(1,-1,1,2)
>>> T
 1  0  1  1    0    0
 0  1  0 -0.5 0.5 0.5
 0  0  1  0.5 0.5 -0.5
```

et enfin $L_2 \leftarrow L_2 - L_3$ :

```
>>> T = T.comb_lignes(1,-1,0,2)
>>> T
   1    0    0  0.5 -0.5  0.5
   0    1    0 -0.5  0.5  0.5
  -0   -0    1  0.5  0.5 -0.5
```

Il ne reste plus qu'à extraire la moitié droite du tableau qui sera l'inverse cherchée à l'aide d'une petite fonction :

Recherche

Complétez...

```
def extrait_carre_droite(self):
    """
    Extrait le carré de droite d'un tableau de GJ

    """
    ???
```

alors :

```
>>> iM = T.extrait_carre_droite()
>>> iM
 0.5 -0.5  0.5
-0.5  0.5  0.5
 0.5  0.5 -0.5
```

On vérifie que c'est bien la matrice inverse de $M$ :

```
>>> iM * M
   1    0    0
   0    1    0
   0    0    1
```

### 3 1 3 Réduction sous-diagonale d'une matrice

On pourrait calculer l'inverse d'une matrice en généralisant la méthode précédente mais cela s'avèrerait beaucoup trop gourmand en temps.
Nous allons dans un premier temps trigonaliser la matrice en effectuant des opérations élémentaires.
Pour éviter de faire trop de transformations, nous allons petit à petit réduire la taille de la matrice à trigonaliser en ne considérant que le carré inférieur droit situé sous le pivot courant et mémoriser chaque ligne obtenue dans une matrice.
Cela permet également de généraliser l'emploi de la méthode de Gauss à des matrices non carrées pour la résolution de systèmes par exemple. Il faut donc faire attention maintenant à utiliser le minimum entre le nombre de lignes et le nombre de colonnes de la matrice.
On place des « mouchards » pour comprendre l'évolution du code.

```
    def triangle(self):
        """ renvoie la triangulation d'une matrice de haut en bas """
        [r,c] = self.D
        m     = min(r,c)
        S     = self
        T     = zeros(r,c)
        while m > 0:
```

```
8            NoLigne = 0
9            while S[NoLigne, 0] == 0 and (NoLigne < m - 1):
10               NoLigne += 1
11           if S[NoLigne, 0] != 0:
12               pivot = S[NoLigne,0]
13               for k in range(1,m):
14                   if S[k,0] != 0:
15                       S = S.comb_lignes(pivot, -S[k,0],k,0)
16                       print("pivot = "+str(pivot))
17                       print("S dans for :")
18                       print(S)
19           T = T.remplace_ligned(r - m,S.F)
20           print("Évolution de T :")
21           print(T)
22           S = S.dswap(NoLigne)
23           m -= 1
24       return T
```

Par exemple :

```
1  In [1]: M
2  Out[1]:
3   1  2  3
4   4  5  6
5   7  8  8
6
7  In [2]: M.triangle()
8  pivot = 1
9  S dans for :
10  1  2  3
11  0 -3 -6
12  7  8  8
13 pivot = 1
14 S dans for :
15  1  2  3
16  0 -3 -6
17  0 -6 -13
18 Évolution de T :
19  1  2  3
20  0  0  0
21  0  0  0
22 pivot = -3
23 S dans for :
24 -3 -6
25  0  3
26 Évolution de T :
27  1  2  3
28  0 -3 -6
29  0  0  0
30 Évolution de T :
31  1  2  3
32  0 -3 -6
33  0  0  3
```

Nous avons besoin de deux fonctions intermédiaires, une remplissant le tableau T et l'autre réduisant S.

```
1      def decoupe(self,i):
2          """
3          Fonction interne à triangle qui retire la 1ère ligne et la 1ère colonne
4
5          """
6          [lig, col], f = self.D, self.F
```

```
        return Mat([lig-1,col-1],lambda r,c : f(r+1,c+1))

    def remplace_ligned(self,i,g):
        """
        Fonction interne à triangle qui remplace dans la ligne i
        les coefficients à partir de la colonne i par ceux du tableau S
        """
        [lig, col], f = self.D, self.F
        h = lambda r,c: g(r-i,c-i) if r == i and c >= i else  f(r,c)
        return Mat([lig,col],h)
```

Recherche

Déterminez une fonction qui calcule rapidement le rang d'une matrice.

### 3 1 4 Calcul du déterminant

On triangularise la matrice et le déterminant est égal au produit des éléments diagonaux à un détail près : il ne faut pas oublier de tenir compte de la parité du nombre d'échanges de lignes ainsi que des multiplications des lignes modifiées par combinaisons.

Recherche

Créez une fonction qui calcule le déterminant :

```
def det(self):
 """ renvoie le déterminant de self par Gauss-Jordan """
????
```

C'est assez efficace pour du Python basique : 555 ms pour un déterminant d'une matrice de taille 200.

```
In [1]: M = unite(200)
In [2]: %timeit M.det()
1 loops, best of 3: 555 ms per loop
```

### 3 1 5 Calcul de l'inverse d'une matrice

La trigonalisation est assez rapide du fait de la réduction progressive de S. L'idée est alors de trigonaliser de bas en haut puis de haut en bas pour arriver à une matrice diagonale ce qui sera plus efficace qu'un traitement de tout le tableau en continu.
On peut utiliser la fonction `det` qui est rapide et permet de ne pas prendre trop de précautions ensuite sachant que la matrice est inversible.

```
def inverse(self):
    return self.cat_carre_droite().triangle().diago_triangle().extrait_carre_droite()
```

Le tout est de construire cette méthode `diago_triangle` qui va trigonaliser le triangle en le parcourant de bas en haut.
On transforme légèrement les fonctions intermédiaires précedentes pour les adapter au nouveau parcours :

```
    def decoupe_bas(self):
        """
        Fonction interne à diago_triangle qui retire la dernière ligne et la
        colonne de même numéro de S

        """
        [lig, col], f = self.D, self.F
        #g = lambda r,c: f(lig-1,c) if r == lig else f(i,c) if r == lig-1 else f(r,c)
```

```
9          g = lambda r,c: f(r,c) if c < lig - 1 else f(r,c+1)
10         return Mat([lig-1,col-1],lambda r,c : g(r,c))
11
12     def remplace_ligneg(self,i,g):
13         """
14         Fonction interne à diago_triangle qui remplace dans la ligne i
15         les coefficients à partir de la colonne i par ceux du tableau S
16         """
17         [lig, col], f = self.D, self.F
18         h = lambda r,c: g(r,c - (lig - 1) + i) if r == i and c >= i else  f(r,c)
19         return Mat([lig,col],h)
```

Recherche

Construisez `diago_triangle`

```
1  def diago_triangle(self):
2      ????
```

Recherche

On considère la matrice

$$A = \begin{pmatrix} 10^{-20} & 0 & 1 \\ 1 & 10^{20} & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

Quel est son rang ? Quel est son déterminant ? Utilisez les fonctions précédentes puis du papier et un crayon...

# 4 Manipulation d'images

Dans cette section, des mathématiques concrètes vont nous permettre de réduire, agrandir, assombrir, éclaircir, compresser, bruiter, quantifier,... une photo. Pour cela, il existe des méthodes provenant de la théorie du signal et des mathématiques continues. Nous nous pencherons plutôt sur des méthodes plus légères basées sur l'algèbre linéaire et l'analyse matricielle. Une image sera pour nous une matrice carrée de taille $2^9$ à coefficients dans $[\![0, 2^9 - 1]\!]$. Cela manque de charme ? C'est sans compter sur Lena qui depuis quarante ans rend les maths sexy (la population hantant les laboratoires mathématiques et surtout informatiques est plutôt masculine ...).
Nous étudierons pour cela la « Décomposition en Valeurs Singulières » qui apparaît de nos jours comme un couteau suisse des problèmes linéaires : en traitement de l'image et de tout signal en général, en reconnaissance des formes, en robotique, en statistique, en étude du langage naturel, en géologie, en météorologie, en dynamique des structures, etc.
Dans quel domaine travaille-t-on alors : algèbre linéaire, analyse, probabilités, topologie,... ? Un peu de tout cela et d'autres choses encore : cela s'appelle...la mathématique.

## 4 1 Lena

Nous allons travailler avec des images qui sont des matrices de niveaux de gris. Notre belle amie Léna sera représentée par une matrice carrée de taille $2^9$ ce qui permet de reproduire Léna à l'aide de $2^{18} = 262\,144$ pixels. Léna prend alors beaucoup de place. Nous allons tenter de compresser la pauvre Léna sans pour cela qu'elle ne perde sa qualité graphique. Une des méthodes les plus abordables est d'utiliser la décomposition d'une matrice en valeurs singulières.
C'est un sujet extrêmement riche qui a de nombreuses applications. L'algorithme que nous utiliserons (mais que nous ne détaillerons pas) a été mis au point par deux très éminents chercheurs en 1965 (Gene Golub, états-unien et William Kahan, canadien, père de la norme IEEE-754). Il s'agit donc de mathématiques assez récentes, au moins en comparaison avec votre programme... La petite histoire dit que des chercheurs américains de l'University of Southern California étaient pressés de trouver une image de taille $2^{18}$ pixels pour leur conférence. Passe alors un de leurs

collègues avec, en bon informaticien, le dernier Playboy sous le bras. Ils décidèrent alors d'utiliser le poster central de la Playmate comme support...
La photo originale est ici : http://www.lenna.org/full/len_full.html mais nous n'utiliserons que la partie scannée par les chercheurs, de taille 5.12in × 5.12in...

## 4 2 Environnement de travail

Nous travaillerons dans Pylab avec l'option pylab qui charge les bibliothèques nécessaires à la manipulation d'images (en fait, pylab charge les bibliothèques pour se retrouver dans un environnement proche de celui de logiciels comme MatLab ou Scilab).
$ipython --pylab$
Les images sont alors sous forme d'array de la bibliothèque numpy.
Nous allons créer quelques outils pour continuer à travailler malgré tout avec notre classe `Mat`.

```
def array2mat(tab):
    """ convertit un array de numpy en objet de type Mat"""
    dim = list(tab.shape)
    return Mat(dim,  lambda i,j : tab[i,j])

def mat2array(mat):
    """ convertir un objet de type Mat en array de numpy """
    return np.fromfunction(mat.F,tuple(mat.D),dtype = int)

def montre(mat):
    """ permet de visualiser les images dans un terminal """
    return imshow(mat2array(mat), cmap = cm.gray)
```

Python contient un tableau carré de taille 512 contenant les niveaux de gris représentant Lena. Il suffit de cherger les bonnes bibliothèques :

Haskell
```
from scipy import misc

# lena sous forme d'un objet de type Mat
matlena = array2mat(misc.lena())
```

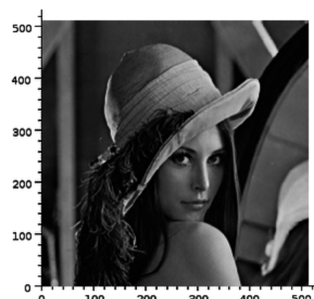
Lena est bien un carré de 512 × 512 pixels :

Haskell
```
In [9]: matlena.D
Out[9]: [512, 512]
```

On peut voir Lena :

```
In [3]: montre(matlena)
Out[3]: <matplotlib.image.AxesImage at 0x7f06c6d0c7b8>
```

et on obtient :




On peut préférer travailler sur une photo de format jpg (qui sera à l'envers) ou png (qui sera à l'endroit). Python la transforme en matrice avec la fonction imread. La matrice obtenue est en

RVB : chaque coefficient est un vecteur (R,V,B). On la convertit en niveau de gris par exemple en calculant la moyenne des trois couleurs.
On récupère d'abord une image :

Haskell
```
In [4]: from urllib.request import urlretrieve

In [5]: urlretrieve('addresse de la photo en ligne','photo.png')
Out[5]: ('photo.png', <http.client.HTTPMessage at 0x7f3aeeb5fa20>)
```

et on la transforme en matrice :

Haskell
```
lena_face_couleur = imread('lena_face.jpeg')

def rvb_to_gray(m):
    r = len(m)
    c = len(m[0])
    return(np.array([[m[i,j][0] for j in range(c)] for i in range(r)]))

lena_face_array = rvb_to_gray(lena_face_couleur)

lena_face = array2mat(lena_face_array)
```

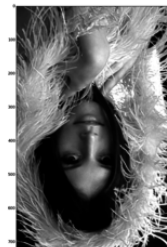
En exclusivité mondiale, voici Lena de face :

```
1 In [3]: montre(lena_face)
2 Out[3]: <matplotlib.image.AxesImage at 0x7f6d8c68cc88>
```

et on obtient :

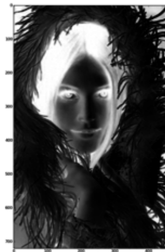
## 4 3 Manipulations basiques

Comment obtenir les deux images de gauche sachant que l'image originale est à droite :



Sachant que l'échelle de gris varie entre 0 et 255, comment obtenir les images suivantes :

On peut modifier le contraste en « mappant » par une fonction croissant plus rapidement ou plus lentement que la fonction identité sur [0 ;255] vers [0 ;255].
Comment obtenir l'image en négatif de Lena ?



## 4 4 Résolution

Une matrice de taille $2^9$ contient $2^{18}$ entiers codés entre 0 et $2^8 - 1$ ce qui prend pas mal de place en mémoire. Peut-on être plus économique sans pour cela diminuer la qualité esthétique de la photo ?
La première idée est de prendre de plus gros pixels, c'est-à-dire une matrice plus petite : on extrait par exemple régulièrement un pixel sur $k$ (en choisissant $k$ parmi une puissance de 2 inférieure à $2^9$).
Comment obtenir par exemple ces images ?

## 4 5 Quantification

On peut également réduire le nombre de niveaux de gris en regroupant par exemple tous les niveaux entre 0 et 63 en un seul niveau 0, puis 64 à 127 en 64, 128 à 191 en 128, 192 à 256 en 192.
Par exemple, avec 4, 8 puis 16 niveaux :

Pour cela, on divise chaque coefficient de la matrice par une puissance de 2.
Mais une division d'un nombre par une puissance de 2 revient à faire un décalage de la chaîne de bits représentant ce nombre vers la droite Ce décalage vers la droite est effectué en Python par

l'opérateur >>xcom]decalage !gauche@>>. L'image précedente peut donc être obtenue par ? ? ?

### 4 6 Enlever le bruit

La transmission d'informations a toujours posé des problèmes : voleurs de grands chemins, poteaux télégraphiques sciés, attaques d'indiens, etc.
Claude Elwood SHANNON (1916 - 2001) est un mathématicien-inventeur-jongleur américain qui, suite à son article « *A mathematical theory of communications* » paru en 1948, est considéré comme le fondateur de la *théorie de l'information* qui est bien sûr une des bases de... l'*informatique*.
L'idée est d'étudier et de quantifier l'« information » émise et reçue : quelle est la compression maximale de données digitales ? Quel débit choisir pour transmettre un message dans un canal « bruité » ? Quel est le niveau de sûreté d'un chiffrement ?...
La théorie de l'information de SHANNON est fondée sur des modèles probabilistes : leur étude est donc un préalable à l'étude de problèmes de réseaux, d'intelligence artificielle, de systèmes complexes.
Par exemple, dans le schéma de communication présenté par SHANNON, la source et le destinataire d'une information étant séparés, des perturbations peuvent créer une différence entre le message émis et le message reçu. Ces perturbations (bruit de fond thermique ou accoustique, erreurs d'écriture ou de lecture, etc.) sont de nature *aléatoire* : il n'est pas possible de prévoir leur effet. De plus, le message source est par nature *imprévisible* du point de vue du destinataire (sinon, à quoi bon le transmettre).
SHANNON a également emprunté à la physique la notion d'entropie pour mesurer le désordre de cette transmission d'information, cette même notion d'entropie qui a inspiré notre héros national, Cédric VILLANI...
Mais revenons à Lena. Nous allons simuler une Lena bruitée. Pour cela, nous allons ajouter ajouter des pixels aléatoirement selon une loi de Bernoulli de paramètre $p$.

```
from random import randint

def ber(p):
    return 1 if random() < p else 0

def rand_image(dim,p):
    return Mat(dim, lambda i,j : randint(0,255)*ber(p))

def bruit(mat,p):
     return (mat + rand_image(mat.D,p)).map(lambda x : x % 256)
```

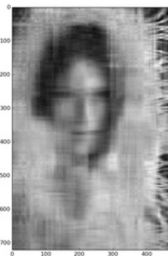
Par exemple, voici Lena bruitée à 0.1 :

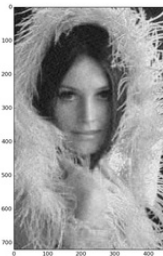
Il existe de nombreuses méthodes, certaines très élaborées, permettant de minimiser ce bruit. Une des plus simples est de remplacer chaque pixel par la moyenne de lui-même et de ses 8 autres voisins directs, voire aussi ses 24 voisins directs et indirect, voire plus, ou de la médiane de ces séries de « cercles » concentriques de pixels.
Voici les résultats avec respectivement la moyenne sur des 9 pixels puis 25 pixels et à doite la médiane de carrés de 9 pixels. La médiane est plus efficace que la moyenne !

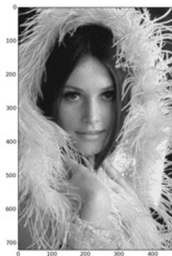
Do it ! !

### 4 7 Compression par SVD

On en a déjà parlé.
Le problème est d'obtenir cette décomposition. Python heureusement s'en charge grâce à la fonction `linalg.svd(mat)` qui renvoie la décomposition en valeurs singulières sous la forme de trois « array » de numpy $U$, $S$ et ${}^{t}V$.
Par souci d'efficacité, nous travaillerons ici uniquement avec les « array » de numpy.
Voici les trois niveaux de compression 10, 50 et 100 :





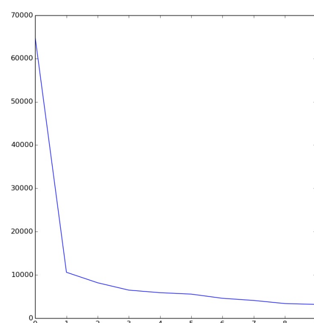


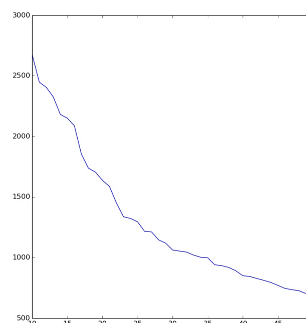
obtenus par :

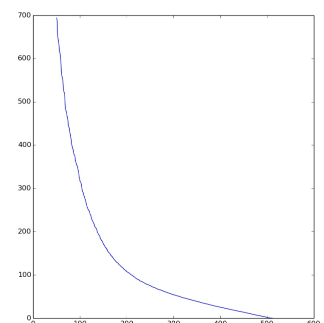
```
In [144]: imshow(c_svd(lena_face_array,100),cmap = cm.gray)
```

On peut observer la décroissance extrêmement rapide des valeurs singulières :

```
In [1]: U,s,tV = linalg.svd(misc.lena())

In [2]: x = np.arange(len(s))

In [3]: plot(x[:10],s[:10])

In [4]: plot(x[10:50],s[10:50])

In [5]: plot(x[50:],s[50:])
```




On peut également faire une petite animation des images compressées :

```python
def anim(im,n):
    for k in range(n):
        plt.imsave("lena_face_svd" + str(1000 + k), c_svd(lena_face_array,k), cmap =
            ↪ cm.gray)
```

Puis ensuite dans un terminal :
$apngasmanim_lena_face.pnglena_face_svd * .png15$
Il faut installer au préalable le petit logiciel apngasm (disponible dans les dépôts Linux ou sur http://sourceforge.net/projects/apngasm/).
On obtient l'image animée suivante :
http://download.tuxfamily.org/tehessinmath/les_images/anim_lena_face.png

Recherche

**Détection des bords**

Pour sélectionner des objets sur une image, on peut essayer de détecter leurs bords, i.e. des zones de brusque changement de niveaux de gris.
On remplace alors un pixel par une mesure des écarts des voisins immédiats donnée par exemple par :

$$\sqrt{\left(m_{i,j+1} - m_{i,j-1}\right)^2 + \left(m_{i+1,j} - m_{i-1,j}\right)^2}$$

Écrivez une fonction qui trace le contour défini ci-dessus.