Corrigé du TP 11 Pivot de Gauss

Chenevois-Jouhet-Junier

1 Exercices 1,2 et 3:

Voici les résolutions des systèmes 1, 2 et 3 avec le module de calcul formel sympy (il est intégré dans Pyzo). Attention, lancer la commande `x, y, z, t = sympy.symbols('x y z t')` avant chaque résolution de système avec `sympy` car les noms `x y  z t` seront écrasés entre temps par les fonctions de résolution de système linéaire que vous allez écrire.

```
import sympy
x, y, z, t = sympy.symbols('x y z t')
```

- Résolution du système 1

  ```
  In [14]: sympy.solve([2*x + 3*y - 5, 5*x-2*y + 16], [x, y])
  Out [14]: {y: 3, x: -2}
  ```

- Résolution du système 2

  ```
  In [15]: sympy.solve([2*x + 2*y  -3*z - 2, y - 6*z + 3, z -4 ], [x, y, z])
  Out [15]: {z: 4, y: 21, x: -14}
  ```

- Résolution du système 3

  ```
  In [16]: sympy.solve([2*x + 2*y - 3*z - 2, -2*x - 3*z + 5, 6*x + 4*y + 4*z - 16], [x, y, z])
  Out [16]: {z: 1, y: 3/2, x: 1}
  ```

2 Exercice 4

```
def echange_ligne(A, i, j):
    """Il suffit d'échanger les pointeurs"""
    A[i], A[j] = A[j], A[i]

def echange_ligne2(A, i, j):
    """Pour garder les memes pointeurs et copier les elements"""
    for k in range(len(A[i])):
        A[i][k], A[j][k] = A[j][k], A[i][k]

def transvection(A, i, j, mu):
    """Transvection Li <- Li + mu*Lj"""
```

```
    for k in range(len(A[i])):
        A[i][k] += mu*A[j][k]

"""
>>> A = [[4, 5, 6], [1, 2, 3]]
>>> echange_ligne(A, 0, 1)
>>> A
[[1, 2, 3], [4, 5, 6]]
>>> transvection(A, 1, 0, -4)
>>> A
[[1, 2, 3], [0, -3, -6]]
"""
```

3 Exercice 5

```
def pivot_partiel(A, j0):
    """Recherche du pivot de module maximal dans la colonne j0 de
    la matrice A, parmi les lignes d'index >= j0"""
    i = j0 #ligne du maximum provisoire
    modulepivot = abs(A[i][j0])
    for k in range(j0+1, len(A)):
        if abs(A[k][j0]) > modulepivot:
            i, modulepivot = k, abs(A[k][j0])
    return i

"""
>>> A = [[1, 2, 3, 4], [0, 1, 3, 5], [0, -4, 1, 0], [0, 3, 0, 0]]
>>> pivot_partiel(A, 1)
2
"""
```

4 Exercice 6

```
def copie(m):
    """retourne une copie de la matrice m"""
    nlines,ncols = len(m),len(m[0])
    cp = [[0]*ncols for _ in range(nlines)]
    for i in range(nlines):
        for j in range(ncols):
            cp[i][j]=m[i][j]
    return  cp

def resolution_systeme(A0, y0, verbose=False):
    """Resolution d'un systeme de Cramer par la methode du pivot
    de Gauss. Si le booleen verbose vaut True, les etapes intermediaires
    de la mise sous forme triangulaire sont affichees"""
    A = copie(A0) #copie de A
    y = copie(y0) #copie de y
    n = len(A) #nombre de lignes
```

```
    if verbose: #affichage facultatif des etapes
            print('Matrice=', A)
            print('Ordonnees =',y, end='\n\n')
    #phase de mise sous forme triangulaire
    for i in range(n-1):
        #recherche du pivot partiel dans la colonne i
        j = pivot_partiel(A, i)
        #echange de la ligne i  de A et de celle du pivot partiel
        if i != j:
            echange_ligne(A, i, j)
            #idem pour les inconnues
            echange_ligne(y, i, j)
        #transvections dans la colonne i pour mettre des zeros
        pivot = A[i][i]
        for k in range(i+1, n):
            #ATTENTION à bien stocker le coefficient de transvection
            mu = -A[k][i]/float(pivot)
            transvection(A, k, i, mu)
            #si on utilise -A[k][i]/float(pivot) au lieu de mu
            #c'est faux car A[k][i] a été modifié
            transvection(y, k, i, mu)
        if verbose: #affichage facultatif des etapes
            print('Etape %d :'%(i+1), 'pivot = %.3f'%pivot)
            print('Matrice=', A)
            print('Ordonnees =', y, end='\n\n')
    x = [[0] for i in range(n)]
    #phase de remontée
    for i in range(n-1, -1, -1):
        x[i] = [1/float(A[i][i])*(y[i][0] - sum(A[i][k]*x[k][0] for k in range(i+1, n)))]
    return x

"""
In [10]: resolution_systeme([[2, 3], [5, -2]], [[5], [-16]])
Out[10]: [[-2.0], [3.0]]
"""
```

5 Exercice 7

Test de la fonction de résolution maison sur les systemes de l'exercice 2

```
"""
#systeme 1
In [10]: resolution_systeme([[2, 3], [5, -2]], [[5], [-16]])
Out[10]: [[-2.0], [3.0]]

#systeme 2
In [11]: resolution_systeme([[2, 2, -3],[0, 1, -6], [0, 0, 1]],[[2], [-3], [4]])
Out[11]: [[-14.0], [21.0], [4.0]]

#systeme 3

In [12]: resolution_systeme([[2,2,-3],[-2,-1,-3 ],[6,4,4]],[[2],[-5], [16]], verbose=True)
Matrice= [[2, 2, -3], [-2, -1, -3], [6, 4, 4]]
```

```
Ordonnees = [[2], [-5], [16]]

Etape 1 : pivot = 6.000
Matrice= [[6, 4, 4], [0.0, 0.33333333333333326, -1.6666666666666667],
[0.0, 0.6666666666666667, -4.333333333333333]]
Ordonnees = [[16], [0.33333333333333304], [-3.333333333333333]]

Etape 2 : pivot = 0.667
Matrice= [[6, 4, 4], [0.0, 0.6666666666666667, -4.333333333333333], [0.0, 0.0, 0.4999999999999989]]
Ordonnees = [[16], [-3.333333333333333], [1.9999999999999991]]

Out[12]: [[-14.000000000000036], [21.000000000000046], [4.000000000000007]]
"""
```

6 Exercice 8 Comparaison de la fonction de résolution maison et de `numpy.linalg.solve`

```
import numpy as np

"""
>>> import numpy as np
>>> help(np.linalg.solve) #pour obtenir la documentation de cette fonction
...
"""
```

- Résolution du système 1

  ```
  In [10]: np.linalg.solve([[2, 3],[5,-2]],[[5], [-16]])

  Out[10]: array([[-2.],[ 3.]])
  ```

- Résolution du système 2

  ```
  In [11]: np.linalg.solve([[2, 2, -3],[0, 1, -6], [0, 0, 1]],[[2], [-3], [4]])

  Out[11]: array([[-14.],[ 21.],[  4.]])
  ```

- Résolution du système 3

  ```
  In [12]: x = np.linalg.solve([[2,2,-3],[-2,-1,-3 ],[6,4,4]],[[2],[-5], [16]])

  In [13]: x

  Out[13]: array([[-14.], [ 21.], [ 4.]])

  In [14]: x[0][0]

  Out[14]: -14.000000000000023
  ```

La précision avec les fonctions de `numpy` n'est pas meilleure que celle obtenue avec la fonction `resolution_systeme` maison. Les flottants du vecteur solution sont d'un type spécifique à `numpy`.

```
"""
In [15]: type(x[0][0])
Out[15]: numpy.float64
"""
```

7 Exercice 9 Tests avec des systèmes qui ne sont pas de Cramer

- Le systeme $\begin{cases} x + 2y = 1 \\ 2x + 4y = 1 \end{cases}$ n'a pas de solutions

```
"""
In [23]: sympy.solve([x+2*y-1, 2*x + 4*y - 1], [x, y])
Out[23]: []

In [24]: resolution_systeme([[1, 2], [2, 4]], [[1], [1]])
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

In [25]: np.linalg.solve([[1, 2], [2, 4]], [[1], [1]])
---------------------------------------------------------------------------
numpy.linalg.linalg.LinAlgError: Singular matrix
"""
```

Le systeme $\begin{cases} x + 2y = 1 \\ 2x + 4y = 2 \end{cases}$ a une infinité de couples solutions de la forme $(1 - 2y, y)$.

```
"""
In [26]: sympy.solve([x+2*y-1, 2*x + 4*y - 2], [x, y])
Out[26]: {x: -2*y + 1}

In [27]: resolution_systeme([[1, 2], [2, 4]], [[1], [2]])
---------------------------------------------------------------------------
ZeroDivisionError: float division by zerox

In [28]: np.linalg.solve([[1, 2], [2, 4]], [[1], [2]])

LinAlgError: Singular matrix
"""
```

8 Exercice 10 Problemes posés par les flottants

- Premier systeme `A1.X = Y1`

```
A1 = [[1, 1/4., 1], [1, 1/3., 2], [0, 1, 12]]
Y1 = [[0], [0], [1]]
```

```
"""
In [33]: sympy.solve([x+y, x/4 + y/3 + z, x + 2*y + 12*z - 1], [x, y, z])
```

```
Out[33]: []


In [32]: resolution_systeme(A1 ,Y1, verbose=True)
Matrice= [[1, 0.25, 1], [1, 0.3333333333333333, 2], [0, 1, 12]]
Ordonnees = [[0], [0], [1]]

Etape 1 : pivot = 1.000
Matrice= [[1, 0.25, 1], [0.0, 0.08333333333333331, 1.0], [0.0, 1.0, 12.0]]
Ordonnees = [[0], [0.0], [1.0]]

Etape 2 : pivot = 1.000
Matrice= [[1, 0.25, 1], [0.0, 1.0, 12.0], [0.0, 0.0, 2.220446049250313e-16]]
Ordonnees = [[0], [1.0], [-0.08333333333333331]]

Out[32]: [[-750599937895082.8], [4503599627370496.0], [-375299968947541.25]]
"""
```

Le premier système n'a pas de solution (comme on peut le vérifier avec `sympy.solve`).

Et pourtant la fonction de résolution maison retourne un triplet solution : `[-750599937895082.8, 4503599627370496.0, -375299968947541.25]`.

L'erreur se produit lors de la dernière étape de mise sous forme triangulaire : `2.220446049250313e-16` apparaît en `Matrice[2][2]` alors qu'on devrait avoir 0 et donc une équation d'incompatibilité $0 = 1$ pour la dernière ligne du système.

L'erreur est due à la représentation approchée sous forme de flottant de $1/3$ par $0.333\ldots$ puis de $1/3 - 1/4 = 1/12$ par $0.08333333\ldots$ Le coefficient de la dernière ligne devrait être $1 - 1/12 \times 12 = 0$, mais c'est $1 - (1/3. - 1/4.) \times 12$ or en représentation flottante on a :

```
"""
In [34]: 1- (1/3. - 1/4.)*12
Out[34]: 2.220446049250313e-16
"""
```

Ainsi les calculs approchés avec des flottants font apparaître des termes infinitésimaux très petits à la place de zéro et transforment en systèmes de Cramer des systèmes qui n'ont pas cette propriété.

De plus si un de ces infinitésimaux est pris comme pivot, alors des termes très grands (et faux) apparaissent, comme avec la fonction de bibliothèque de `numpy` :

```
"""
In [35]: np.linalg.solve(A1, Y1)
Out[35]:
array([[ -7.50599938e+14],[  4.50359963e+15],[ -3.75299969e+14]])
"""
```

- Deuxième système `A2.X = Y2`

```
A2 = [[1, 10**15, 1], [1, 10**(-2), 2], [0, 10**15, -1]]
Y2 = [[1], [0], [0]]
```

```
"""
In [38]: sympy.solve([x + y - 1, 10**15*x + sympy.Rational(1, 10**2)*y + 10**15*z, x + 2*y - z], [x, 
Out[38]: {y: -200000000000000000, x: 200000000000000001, z: -199999999999999999}

In [39]: resolution_systeme(A2 ,Y2, verbose=True)
```

```
Matrice= [[1, 1000000000000000, 1], [1, 0.01, 2], [0, 1000000000000000, -1]]
Ordonnees = [[1], [0], [0]]

Etape 1 : pivot = 1.000
Matrice= [[1, 1000000000000000, 1], [0.0, -1000000000000000.0, 1.0], [0.0, 1000000000000000.0, -1.0]]
Ordonnees = [[1], [-1.0], [0.0]]

Etape 2 : pivot = -1000000000000000.000
Matrice= [[1, 1000000000000000, 1], [0.0, -1000000000000000.0, 1.0], [0.0, 0.0, 0.0]]
Ordonnees = [[1], [-1.0], [-1.0]]

ZeroDivisionError: float division by zero
"""
```

Dans ce cas, les erreurs d'approximation par des flottants enlèvent la propriété de Cramer au système alors que ce système est de Cramer et a pour unique solution `[1 + 2*10**17, -2*10**17, -2*10**17 + 1]`, comme on peut le vérifier avec `sympy.solve`.

L'erreur de division par 0 s'est produite lors de la première étape de la phase de remontée puisque le système n'est plus de Cramer.

L'erreur d'approximation s'est produite lors de la première étape de la mise sous forme triangulaire au cours de la transvection qui a transformé la deuxième ligne en :

```
[0.0, -1000000000000000.0, 1.0]
```

au lieu de `[0.0, -999999999999999.99, 1.0]`

En effet Python calcule ainsi :

```
"""
In [44]: 10**-2-10**15
Out[44]: -1000000000000000.0 #-> c'est faux phenomene d'absorption
"""
```

Ainsi la deuxième et la troisième ligne de la matrice sont devenues proportionnelles alors que les ordonnées ne le sont pas et le système obtenu par équivalence n'est plus de Cramer.

La fonction de bibliothèque de numpy ne fait pas mieux :

```
"""
In [45]: np.linalg.solve(A2, Y2)
---------------------------------------------------------------------------
...
LinAlgError: Singular matrix
"""
```

9 Exercice 11 Inversion de matrice

On utilise la *méthode du miroir*.

- On applique la phase de mise sous forme triangulaire puis de remontée, en parallèle à la matrice de coefficients `A` et à la matrice identité.

- On part de A.X=Y Lors la phase de mise sous forme triangulaire on multiplie à gauche A par des matrices élémentaires d'opérations sur les lignes : triangulaires inférieures (transvection) ou matrices de permutation (échange de ligne) et on obtient :

  `L(1)..L(N).A.X=L(1)..L(N).Y .`

- Lors la phase de remontée on multiplie à gauche par des matrices de transvection (triangulaires supérieures) ou de dilatation (notées S, elles permutent avec les autres car elles sont diagonales) et on a :

  `S(1)...S(M).L(1)...L(P).A.X = S(1)...S(M).L(1)...L(P).Y`

  où le membre de gauche vaut exactement X et donc `S(1)...S(M)L(1)...L(P)` produit de toutes les matrices élémentaires d'opérations sur les lignes est l'inverse de A..

- Pour obtenir cette matrice inverse, il suffit d'appliquer les mêmes opérations élémentaires sur les lignes à partir de la matrice identité :

  `inv(A).I= inv(A)`

```
def identite1(n):
    """Retourne une matrice carree identite de dimensions n*n"""
    return [[0]*i + [1] + [0]*(n - i -1 ) for i in range(n)]

def dilatation(A, i, mu):
    """Li <- Li*mu si mu non nul"""
    assert mu != 0
    #A[i] = [A[i][k]*mu for k in range(len(A[i])]
    for k in range(len(A[i])):
        A[i][k] *= mu

def inversion(A0):
    """Inversion de la matrice A0 par la methode du miroir"""
    A = copie(A0) #copie de A
    n = len(A) #nombre de lignes
    I = identite1(n)
    for i in range(n-1):
        j = pivot_partiel(A, i)
        if i != j:
            echange_ligne(A, i, j)
            echange_ligne(I, i, j)
        #transvections dans la colonne i pour mettre des zeros
        pivot = A[i][i]
        for k in range(i+1, n):
            mu = -A[k][i]/float(pivot)
            transvection(A, k, i, mu)
            transvection(I, k, i, mu)
    #phase de remontée
    for i in range(n-1, -1, -1):
        #transvections
        for k in range(i+1, n):
            mu = -A[i][k]
            transvection(I, i, k, mu)
        #x[i] = 1/float(A[i][i])
        dilatation(I, i, 1/float(A[i][i]))
    return I
"""
>>> inversion([[1,2],[0,3]])
```

```
[[1.0, -0.6666666666666666], [0.0, 0.3333333333333333]]
>>> inversion([[1,2],[3,5]])
[[-4.999999999999997, 1.9999999999999993], [2.9999999999999987, -0.9999999999999996]]
>>> inversion([[1,2],[3,4]])
[[-1.9999999999999996, 0.9999999999999998], [1.4999999999999998, -0.4999999999999999]]
>>> inversion([[2,2,-3],[0,1,-6],[0,0,1]])
[[0.5, -1.0, -4.5], [0.0, 1.0, 6.0], [0.0, 0.0, 1.0]]
>>> inversion([[2,2,-3],[-2,-1,-3],[6,4,4]])
[[4.000000000000008, -10.000000000000021, -4.500000000000009],
[-5.00000000000011, 13.000000000000027, 6.000000000000012],
 [-1.000000000000018, 2.000000000000044, 1.000000000000002]]
"""
```

- On peut aussi utiliser la fonction `inv` de la bibliotheque `numpy.linalg` :

```
"""
>>> import numpy as np
>>> numpy.linalg.inv([[2,2,-3],[-2,-1,-3],[6,4,4]])
array([[  4. , -10. ,  -4.5],
       [ -5. ,  13. ,   6. ],
       [ -1. ,   2. ,   1. ]])
"""
```

- Et on peut vérifier les méthodes numériques avec le module de calcul formel `sympy` :

```
"""
In [49]:  A = sympy.Matrix([[2,2,-3],[-2,-1,-3],[6,4,4]])

In [50]: A**(-1)
Out[50]:
Matrix([[ 4, -10, -9/2],[-5,  13,    6],[-1,   2,    1]])

In [51]: A.det()
Out[51]: 2

"""
```

Plus généralement on peut programmer une fonction `pivot_gauss(B)` qui applique l'algorithme du pivot de Gauss à une matrice, puis envelopper cette fonction dans des fonctions de résolution de système ou d'inversion.

```
def pivot_gauss(B):
    """Pivot de Gauss appliqué à une matrice B"""
    n = len(B)
    A = [ B[k][:]  for k in range(n)]  #Copie de B
    for i in range(n - 1):
        j = pivot_partiel(A, i)  #recherche de la ligne du pivot partiel
        echange_ligne(A, i, j)
        pivot = A[i][i]
        for k in range(i + 1, n):
            transvection(A, k, i, - A[k][i]/pivot)
    for i in range(n - 1, -1, -1):
        #l'ordre des transvections n'a pas d'importance
        # car il y a des zeros après les pivots dans les lignes déjà traitées
        for j in range(i + 1, n, 1):
```

```
            transvection(A, i, j, -A[i][j]/A[j][j])
        dilatation(A, i, 1/A[i][i])
    return A

def resolution_systeme2(A0, y0):
    matricePlus = [ A0[k][:] + y0[k][:] for k in range(len(A0))]
    A = pivot_gauss(matricePlus)
    return [Ligne[-1] for Ligne in A]

def inversion2(A0):
    n = len(A0)
    matricePlus = [ A0[k][:] + [ 1 if j == k else 0 for j in range(n)] for k in range(n)]
    A = pivot_gauss(matricePlus)
    return [Ligne[n:] for Ligne in A]
```

10 Exercice 13 Inversion de matrice et problèmes avec les flottants

```
A3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
A4 = [[1, 1./4, 1], [1, 1./3, 2], [0, 1, 12]]
A5 = [[1,1.+10**5,1],[1,1+10**(-5),2],[0,10**5,-1]]
A6 = [[1,10**15,1],[1,10**(-2),2],[0,10**15,-1]]
```

- La matrice `A3` est non inversible alors que les fonctions d'inversion numérique (maison et de bibliothèque) la considèrent comme inversible:

  `A3[2] - A3[1] = A3[1] - A3[0] = [3, 3, 3]`

donc `A3[2] = 2*A3[1] - A3[0]` donc les lignes de `A3` sont liées donc A3 pas inversible

ou encore `det(A3) = 1*5*9 + 2*6*7 + 4*8*3 - 3*5*7 - 2*4*9 - 6*8*1 = 0`

```
"""
In [57]: inversion(A3)
Out[57]:
[[-4503599627370498.0, 9007199254740992.0, -4503599627370494.5],
 [9007199254740996.0, -1.8014398509481984e+16, 9007199254740990.0],
 [-4503599627370498.0, 9007199254740992.0, -4503599627370495.5]]

In [58]: A3bis = sympy.Matrix(A3)

In [59]: A3bis.det()
Out[59]: 0

In [60]: A3bis**(-1)
---------------------------------------------------------------------------

ValueError: Matrix det == 0; not invertible.

In [63]: np.linalg.inv(A3)
Out[63]:
array([[  3.15221191e+15,  -6.30442381e+15,   3.15221191e+15],
       [ -6.30442381e+15,   1.26088476e+16,  -6.30442381e+15],
       [  3.15221191e+15,  -6.30442381e+15,   3.15221191e+15]])
"""
```

Explication : à l'étape 2 de la phase de mise sous forme triangulaire la dernière ligne comporte deux infinitésimaux au lieu de zéros :

```
"""
>>> resolution_systeme(A3, [[1],[0], [0]], verbose=True)
Matrice= [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Ordonnees = [1, 0, 0]

Etape 1 : pivot = 7.000
Matrice= [[7, 8, 9], [0.0, 0.4285714285714288, 0.8571428571428577],
 [0.0, 0.8571428571428572, 1.7142857142857144]]
Ordonnees = [0, 0.0, 1.0]

Etape 2 : pivot = 0.857
Matrice= [[7, 8, 9], [0.0, 0.8571428571428572, 1.7142857142857144],
[0.0, 5.551115123125783e-17, 1.1102230246251565e-16]]
Ordonnees = [0, 1.0, -0.5000000000000002]

"""
```

- La matrice `A4` est non inversible alors que les fonctions numériques la considèrent comme inversible (voir l'exo 10 pour l'explication par erreurs d'approximations).

```
"""
In [65]: inversion(A4)
Out[65]: [[-9007199254740992.0, 9007199254740992.0, -750599937895082.8],
[5.404319552844595e+16, -5.404319552844595e+16, 4503599627370496.0],
[-4503599627370496.0, 4503599627370496.0, -375299968947541.25]]

In [66]: np.linalg.inv(A4)
Out[66]: array([[ -9.00719925e+15,   9.00719925e+15,  -7.50599938e+14],
       [  5.40431955e+16,  -5.40431955e+16,   4.50359963e+15],
       [ -4.50359963e+15,   4.50359963e+15,  -3.75299969e+14]])

In [67]: A4bis = sympy.Matrix([[1,sympy.Rational(1,4), 1], [1, sympy.Rational(1,3), 2], [0, 1, 12]])

In [68]: A4bis**(-1)

ValueError: Matrix det == 0; not invertible.

In [69]: A4bis.det()
Out[69]: 0

"""
```

- La matrice `A5` est inversible :

  `det(A5) = -(1+10**(-5)) + 10**5 - 2*10**5 + (1 + 10**5) = -10**(-5)`

```
"""
>>> inversion(A5)
[[20000098346.184444, -20000098345.184444, -20000098344.184444],
 [-99999.9917259636, 99999.9917259636, 99999.9917259636],
  [-9999999172.59636, 9999999172.59636, 9999999171.59636]]

>>> np.linalg.inv(A5)
```

```
array([[  2.00001206e+10,  -2.00001205e+10,  -2.00001205e+10],
[ -1.00000103e+05,   1.00000103e+05,   1.00000103e+05],
[ -1.00000103e+10,   1.00000103e+10,   1.00000103e+10]])

In [74]: A5bis = sympy.Matrix([[1,1+10**5,1],[1,1+Rational(1,10**5),2],[0,10**5,-1]])

In [75]: A5bis.det()
Out[75]: -1/100000

In [76]: A5bis**(-1)
Out[76]:
Matrix([[ 20000100001, -20000100000, -20000099999],[-100000, 100000, 100000],
 [-10000000000,  10000000000,   9999999999]])
"""
```

- La matrice `A6` est inversible alors que les fonctions numériques la considerent comme non inversible. (voir l'exo 10 pour l'explication par erreurs d'approximations)

```
"""
In [85]: inversion(A6)
Traceback (most recent call last):
  .....
ZeroDivisionError: float division by zero

In [86]: np.linalg.inv(A6)
Traceback (most recent call last):
numpy.linalg.linalg.LinAlgError: Singular matrix

In [87]: A6bis = sympy.Matrix([[1,10**15,1],[1,Rational(1,10**2),2],[0,10**15,-1]])

In [88]: A6bis.det()
Out[88]: -1/100

In [89]: A6bis**(-1)
Out[89]: Matrix([[ 200000000000000001, -200000000000000000, -199999999999999999],
[ -100, 100,100],[-100000000000000000,  100000000000000000,   99999999999999999]])
"""
```

11 Exercice 14 Comparaison entre fonctions maisons et fonctions de bibliothèques sur les matrices de Virginie

```
def virginie1(n):
    """Retourne une matrice de Virginie de dimensions  nxn"""
    assert str(type(n))=="<class 'int'>" and n>=2,"n doit etre un entier >=2"
    V = []
    for i in range(n):
        ligne = []
        for j in range(n):
            if j < i-1 or j > i+1:
                ligne.append(0)
            elif j == i-1 or j == i+1:
```

```
                ligne.append(-1)
            else:
                ligne.append(2)
        V.append(ligne)
    return V

def virginie2(n):
    """Retourne une matrice de Virginie de dimensions  nxn
    Voir le fichier cadeau.py pour une explication sur l'expression
    booleenne utilisee"""
    assert str(type(n))=="<class 'int'>" and n>=2,"n doit etre un entier >=2"
    return [[(i == j-1 and -1) or (i == j and 2) or (i == j+1 and -1) or 0
            for j in range(n)] for i in range(n)]

"""
>>> virginie1(4)
[[2, -1, 0, 0], [-1, 2, -1, 0], [0, -1, 2, -1], [0, 0, -1, 2]]
>>> virginie2(4)
[[2, -1, 0, 0], [-1, 2, -1, 0], [0, -1, 2, -1], [0, 0, -1, 2]]
"""

def exo14():
    import time, numpy
    tabvirginie = [(n,virginie1(n)) for  n in [50,100,200,400]]
    print("Temps pour inverser des matrices de Virginie avec la fonction inversion maison : ")
    for v in tabvirginie:
        t0 = time.time()
        inversion(v[1])
        print('Pour la matrice de Virginie de taille %s : %s'%(v[0],time.time()-t0))
    print("Temps pour inverser des matrices de Virginie avec \
            la fonction inversion inv de numpy.linalg : ")
    for v in tabvirginie:
        t0 = time.time()
        numpy.linalg.inv(v[1])
        print('Pour la matrice de Virginie de taille %s : %s'%(v[0],time.time()-t0))
```

Les fonctions de `numpy` sont beaucoup plus rapides (facteur 100) !

```
"""
>>> exo14()
Temps pour inverser des matrices de Virginie avec la fonction inversion maison :
Pour la matrice de Virginie de taille 50 : 0.04155135154724121
Pour la matrice de Virginie de taille 100 : 0.2840964794158935 5
Pour la matrice de Virginie de taille 200 : 2.1560568809509277
Pour la matrice de Virginie de taille 400 : 18.00588893890381
Temps pour inverser des matrices de Virginie avec la fonction inversion inv de numpy.linalg :
Pour la matrice de Virginie de taille 50 : 0.0004217624664306640 6
Pour la matrice de Virginie de taille 100 : 0.0014603137969970703
Pour la matrice de Virginie de taille 200 : 0.006672382354736328
Pour la matrice de Virginie de taille 400 : 0.04255509376525879
"""
```