```
# ======================== #
# ======================== #
# IPT MATHS SPE            #
# Calcul Matriciel Partie I #
# Novembre 2017            #
# ======================== #
# ======================== #

## Importations
# Importations de numpy
from numpy import *
# Autres importations
from math import log, floor

## CONSTRUCTIONS DE MATRICES

# Exercice N°1
def Mat1(n):
    M = np.zeros((n,n))
    for i in range(n):
        M[i,n-1-i] = 1
    return M

# Exercice N°2
def Mat2(n):
    M = np.zeros((n,n))
    # Diagonale principâle (n coefficients)
    for i in range(n):
        M[i,i] = 1
    # Les 2 diagonales secondaires (n-1 coefficients)
    for i in range(n-1):
        M[i,i+1] = 1
        M[i+1,i] = 1
    return M

# Exercice N°3
def Mat3(n):
    M = np.ones((n,n))
    for i in range(1,n):
        for j in range(n-i):
            M[j,j+i] = i+1
    return M

# Une autre possibilité en utilisant les sous-tableau et la fonction arange
def Mat3_bis(n):
    M = np.ones((n,n))
    for i in range(n):
        M[i,i:] = np.arange(1,n-i+1)
    return M

# Exercice N°4
def MatArith1(n,u0,r):
    M = np.arange(u0,u0+n**2*r,r).reshape(n,n)
    return M

def MatArith2(n,u0,r):
    M = zeros((n,n))
    for i in range(n):
        M[i,:] = np.arange(u0+i*n*r,u0+(i+1)*n*r,r)
    return M
```

```
def MatArith3(n,u0,r):
    M = zeros((n,n))
    for i in range(n):
        for j in range(n):
            M[i,j] = u0 + (n * i + j) * r
    return M

## DECOMPOSITIONS DE MATRICES EN SOMMES

# Exercice N°5
def SAS(M):
    T = np.transpose(M)
    return(0.5 * (M + T),0.5 * (M - T))

# Exercice N°6
def INV2(M):
    n = M.shape[0]
    # LI sera une matrice triangulaire inférieure inversible
    # UI sera une matrice triangulaire supérieure inversible
    LI, UI = zeros((n,n)), zeros((n,n))
    # Coefficients sous la diagonale de LI
    for i in range(1,n):
        for j in range(i):
            LI[i,j] = M[i,j]
    # Coefficients au-dessus de la diagonale de UI
    for i in range(n-1):
        for j in range(i+1,n):
            UI[i,j] = M[i,j]
    # Coefficients diagonaux
    for i in range(n):
        if M[i,i] == 0:
            LI[i,i], UI[i,i] = 1, -1
        else:
            LI[i,i], UI[i,i] = 0.5*M[i,i], 0.5*M[i,i]
    # Renvoi du tuple
    return(LI,UI)

## NATURE

# Exercice N°7
def IsDiag(M):
    n = M.shape[0]
    for i in range(n-1):
        for j in range(i+1,n):
            if M[i,j] != 0 or M[j,i] != 0:
                return(False)
    return(True)

# Exercice N°8
def IsTriang(M):
    # Principe général : on comptabilise les 0 au dessus et en dessous de la
    # diagonale.
    # Initialisations
    n = M.shape[0]
    m = (n*(n-1))//2
    nzsup, nzinf = 0, 0
    # Comptage
    for i in range(n-1):
        for j in range(i+1,n):
```

```
            if M[i,j] == 0:
                nzsup += 1
            if M[j,i] == 0:
                nzinf += 1
    print(nzsup,nzinf)
    # Bilan
    if nzsup != m and nzinf != m:
        return(False)
    elif nzsup == m and nzinf != m:
        return(True,'L')
    elif nzsup != m and nzinf == m:
        return(True,'U')
    else:
        return(True,'D')

## OPERATIONS ELEMENTAIRES SUR LES MATRICES

# Exercice N°9
# ATTENTION ! Ici l1 et l2 désignent des indices MATHEMATIQUES !
def LS_build(n,l1,l2):
    M = np.identity(n)
    M[l1-1,l1-1], M[l2-1,l2-1] = 0, 0
    M[l1-1,l2-1], M[l2-1,l1-1] = 1, 1
    return(M)

def LineSwap1(M,l1,l2):
    return(np.dot(LS_build(M.shape[0],l1,l2),M))

# Exercice N°10
# ATTENTION ! Ici l1 et l2 désignent des indices MATHEMATIQUES !
def LineSwap2(M,l1,l2):
    N = array(M[l2-1])
    M[l2-1] = array(M[l1-1])
    M[l1-1] = N

# Exercice N°11
# ATTENTION ! Ici l désigne un indice MATHEMATIQUE !
def A_build(n,l,Lambda):
    M = np.identity(n)
    M[l-1,l-1] = Lambda
    return(M)

def LineMult(M,l,Lambda):
    return(np.dot(A_build(M.shape[0],l,Lambda),M))

# Exercice N°12
# ATTENTION ! Ici l désigne un indice MATHEMATIQUE !
def LineMult2(M,l,Lambda):
    M[l-1] *= Lambda

# Exercice N°13
# ATTENTION ! Ici l1 et l2 désignent des indices MATHEMATIQUES !
def TV_build(n,l1,l2,Lambda):
    M = np.identity(n)
    M[l1-1,l2-1] = Lambda
    return(M)

def Transvec(M,l1,l2,Lambda):
    return(np.dot(TV_build(M.shape[0],l1,l2,Lambda),M))
```

```
# Exercice N°14
# ATTENTION ! Ici l1 et l2 désignent des indices MATHEMATIQUES !
def Transvec2(M,l1,l2,Lambda):
    M[l1-1] += Lambda * M[l2-1]

## RETOUR SUR LE PRODUIT MATRICIEL

# Exercice N°15
def MulStrassen(A,B):
    # On vérifie que les dimensions des matrices A et B sont adéquates...
    (nA,mA) = A.shape
    (nB,mB) = B.shape
    e = log(nA,2)
    if nA != mA or nB != mB:
        return("Au moins l'une des matrices n'est pas carrée !")
    elif nA != nB:
        return("Les matrices ne sont pas de même dimension !")
    elif e - floor(e) != 0:
        return("La dimension de la matrice n'est pas une puissance de 2 !")
    else:
        # Cas de base
        if nA == 1:
            P = np.zeros((1,1))
            P[0,0] = A[0,0] * B[0,0]
            return(P)
        # Cas général
        else:
            # Les blocs de la matrice A
            A11 = A[:nA//2,:nA//2]
            A12 = A[:nA//2,nA//2:]
            A21 = A[nA//2:,:nA//2]
            A22 = A[nA//2:,nA//2:]
            # Les blocs de la matrice B
            B11 = B[:nA//2,:nA//2]
            B12 = B[:nA//2,nA//2:]
            B21 = B[nA//2:,:nA//2]
            B22 = B[nA//2:,nA//2:]
            # Les produits de Strassen (appels récursifs)
            M1 = MulStrassen(A11 + A22,B11 + B22)
            M2 = MulStrassen(A21 + A22,B11)
            M3 = MulStrassen(A11,B12 - B22)
            M4 = MulStrassen(A22,B21 - B11)
            M5 = MulStrassen(A11 + A12,B22)
            M6 = MulStrassen(A21 - A11,B11 + B12)
            M7 = MulStrassen(A12 - A22,B21 + B22)
            # Construction du produit
            P = np.zeros((nA,nA))
            P[:nA//2,:nA//2] = M1 + M4 - M5 + M7
            P[:nA//2,nA//2:] = M3 + M5
            P[nA//2:,:nA//2] = M2 + M4
            P[nA//2:,nA//2:] = M1 - M2 + M3 + M6
            return(P)

## ECHELONNAGE

# Exercice N°16
def Echelon_base(M):
    """Cette fonction échelonne la matrice M en lignes SUR PLACE et utilise
des
    transvections pour les mises à 0 sous les pivots.
```

```
    PAS DE REDUCTION
    """
    (n,p) = M.shape
    # Cas général : M comporte au moins deux lignes (sinon, on ne fait rien
!)
    if n > 1:
        LC, CC = 0, 0
        while LC < n and CC < p:
            # détermination du pivot.
            # Stratégie de choix : on retient le pivot de plus grande valeur
            # absolue.
            pivot = M[LC,CC]
            ipiv = LC
            for i in range(LC,n):
                if abs(M[i,CC]) > abs(pivot):
                    pivot = M[i,CC]
                    ipiv = i
            # Si le pivot est non nul...
            if pivot != 0:
                # si la ligne du pivot n'est pas la ligne courante on
échange
                # la ligne du pivot et la ligne courante
                if ipiv != LC:
                    LineSwap2(M,LC+1,ipiv+1)
                # Mise à 0 de la sous-colonne
                for i in range(LC + 1,n):
                    Transvec2(M,i+1,LC+1,-M[i,CC]/pivot)
                # On passe à la ligne suivante
                LC += 1
            # On passe à la colonne suivante
            CC += 1
    return M

# Exercice N°17
def Echelon_Etendu(M):
    """Cette fonction échelonne la matrice M en lignes SUR PLACE.
    Elle renvoie un tuple de trois éléments :
        (*) La matrice M échelonnée
        (*) Le produit des matrices de transvection/transposition
        (*) Un booléen valant True/False suivant que M est ou non inversible
    """
    (n,p) = M.shape
    # Cas particulier : M ne comporte qu'une seule ligne.
    if n == 1:
        P = identity(1)
        # Il y a au moins deux colonnes.
        if p > 1:
            return(M,P,False)
        # Il y a une seule colonne.
        else:
            if M[0,0] == 0:
                return(M,P,False)
            else:
                return(M,P,True)
    # Cas général : M comporte au moins deux lignes.
    else:
        P = identity(n)
        Inversible = True
        LC, CC = 0, 0
        while LC < n and CC < p:
```

```
            # détermination du pivot.
            pivot = M[LC,CC]
            ipiv = LC
            for i in range(LC+1,n):
                if abs(M[i,CC]) > abs(pivot):
                    pivot = M[i,CC]
                    ipiv = i
            # Si le pivot est non nul.
            if pivot != 0:
                # si la ligne du pivot n'est pas la ligne courante on
échange
                # la ligne du pivot et la ligne courante.
                if ipiv != LC:
                    P = dot(LS_build(n,LC+1,ipiv+1),P)
                    LineSwap2(M,LC+1,ipiv+1)
                # Mise à 0 de la sous-colonne.
                for i in range(LC + 1,n):
                    Lambda = -M[i,CC]/pivot
                    P = dot(TV_build(n,i+1,LC+1,-Lambda),P)
                    Transvec2(M,i+1,LC+1,Lambda)
                # On incrément l'indice de la ligne courante.
                LC += 1
            else:
                # Dès que l'un des pivot est nul, la matrice M n'est pas
                # inversible.
                Inversible = False
            # On passe à la colonne suivante.
            CC += 1
        return(M,P,Inversible)

# Exercice N°18
def EchelonReduit_base(M):
    """Cette fonction échelonne et réduit la matrice M en lignes SUR PLACE.
    """
    (n,p) = M.shape
    # Cas particulier : M comporte une seule ligne
    if n == 1:
        # Détermination du premier coefficient non nul
        pivot, k = M[0,0], 0
        while M[0,k] == 0 and k < p:
            pivot = M[0,k]
            k += 1
        pivot = M[0,k]
        if pivot != 0:
            print(pivot)
            LineMult2(M,1,1/pivot)
        return M
    # Cas général : M comporte au moins deux lignes
    else:
        LC, CC = 0, 0
        while LC < n and CC < p:
            # détermination du pivot
            pivot = M[LC,CC]
            ipiv = LC
            for i in range(LC+1,n):
                if abs(M[i,CC]) > abs(pivot):
                    pivot = M[i,CC]
                    ipiv = i
            # Si le pivot est non nul...
            if pivot != 0:
```

```
                # si la ligne du pivot n'est pas la ligne courante on
échange
                # la ligne du pivot et la ligne courante.
                if ipiv != LC:
                    LineSwap2(M,LC+1,ipiv+1)
                # Réduction (on normalise la ligne du pivot en la divisant
par
                # le pivot !
                LineMult2(M,LC+1,1/pivot)
                # Mise à 0 des coefficients de la colonne CC grâce à des
                # transvections sur les lignes d'indices i différents de LC.
                for i in range(n):
                    if i != LC:
                        Transvec2(M,i+1,LC+1,-M[i,CC])
                # On passe à la ligne suivante.
                LC += 1
            # On passe à la colonne suivante.
            CC += 1
        return M

# Exercice N°21
def GaussJordan(A,B):
    # PREMIERE ETAPE : VALIDATION DES DIMENSIONS DES MATRICES
    (nA,pA) = A.shape
    if nA != pA:
        return('La matrice n\'est pas carrée !')
    # Test relatif au nombre de lignes de la matrice B
    # (il doit être égal au nombre de lignes de la matrice A)
    (nB, pB) = B.shape
    if nB != nA:
        return('Nombre de lignes de B incorrect !')
    # DEUXIEME ETAPE : AUGMENTATION DE A (CONCATENATION DE A ET B).
    # (En procédant de la sorte, on remarque que l'on conserve les matrices
A
    # et B initiales, la fonction concatenate créant un nouveau tableau)
    A2 = concatenate((A,B),axis=1)
    # TROISIEME ETAPE : ECHELONNAGE REDUIT DE A2
    EchelonReduit_base(A2)
    # RENVOI DE LA SOLUTION
    return(A2[:,pA:])

## Pour tester les exercices 1 à 18 (hors Strassen)
"""Quelques matrices pour tester, en particulier, les programmes
d'échelonnage.
Les calculs se font systématiquement avec des coefficients matriciels de
type
"flottant".
"""

A = array([[1,2,3],[4,6,8],[11,20,56]],dtype=float)

B = array([[1,2,3],[0,6,8],[0,0,56]],dtype=float)

C = array([[1,2,2,-3,2],[2,4,1,0,-5],[4,8,5,-6,-1],
[-1,-2,-1,1,1]],dtype=float)

D = array([[2,-1,0],[-1,2,-1],[0,-1,2]],dtype=float)

E = array([[2,-3,1,-1],[-2,2,-3,2],[4,-9,-2,3],[-2,5,5,-4]],dtype=float)
```

```
F = array([[1,2],[4,6]],dtype=float)

G = array([[3,2,-1,15],[2,3,1,10],[1,1,1,6]],dtype=float)

H = array([[0,0,0,0,4],[0,0,0,1,0],[0,0,1,0,0],[0,5,0,0,0],
[1,0,0,0,0]],dtype=float)

I = array([[0,0,0,0,2,-4,11,5,2,0,2,55]],dtype=float)

J = array([[5]],dtype=float)

K = array([[0]],dtype=float)

## Pour tester la résolution de systèmes
# La matrice carrée A
# ===================
# A = array([[2,-1,0],[-1,2,-1],[0,-1,2]],dtype=float)
# A = array([[2,3,-1],[1,-1,0],[-3,4,6]],dtype=float)
# A = array([[1,2,3],[4,6,8],[11,20,56]],dtype=float)
A = array([[2,0,1,-3],[-1,1,5,0],[0,2,3,1],[1,-2,1,4]],dtype=float)


# La matrice B (comportant autant de lignes que A)
# ================================================
# Si l'on souhaite obtenir l'inverse de la matrice A, on saisit pour B la
# matrice identité.
# B = array([[1,0,0],[0,1,0],[0,0,1]],dtype=float)
# La 2ème matrice B correspond à la 3ème matrice A ci-dessus.
# B = array([[-1],[0],[-34]],dtype=float)
# Cette 3ème matrice B correspond à la 4ème matrice A ci-dessus.
B = array([[-5],[-16],[-5],[5]],dtype=float)

# SOL = PivotGauss2(A,B)
SOL = GaussJordan(A,B)
print("Matrice A :")
print(A)
print("Matrice B :")
print(B)
print("Solution :")
print(SOL)
```