TP5

October 5, 2022

1 TP 5 (matrices)

1.0.1 Exo 1

[1]:
```
A=matrix(2,3)
print(A)
print()
A=zero_matrix(2,3)
print(A)
```

```
[0 0 0]
[0 0 0]

[0 0 0]
[0 0 0]
```

[2]:
```
A=identity_matrix(3)
A
```

[2]:
```
[1 0 0]
[0 1 0]
[0 0 1]
```

[3]:
```
A=diagonal_matrix([1,2,3])
A
```

[3]:
```
[1 0 0]
[0 2 0]
[0 0 3]
```

[4]:
```
A=matrix.random(GF(3),4,5)
A
```

[4]:
```
[0 2 2 0 1]
[2 1 0 1 2]
[2 2 2 2 1]
[2 2 0 1 2]
```

[5]:
```
A[1][2]==A[1,2]
```

[5]:
```
True
```

Attention, il est cependant préférable d'utiliser A[i,j] pour rester dans le monde des matrices.

[6]:
```
A.row(2)
```

[6]:
```
(2, 2, 2, 2, 1)
```

[7]:
```
A.column(0)
```

[7]:
```
(0, 2, 2, 2)
```

[8]:
```
A.rows()
```

[8]:
```
[(0, 2, 2, 0, 1), (2, 1, 0, 1, 2), (2, 2, 2, 2, 1), (2, 2, 0, 1, 2)]
```

[9]:
```
A.ncols()
```

[9]:
```
5
```

[10]:
```
C=A.submatrix(0,2,2,3)
C
```

[10]:
```
[2 0 1]
[0 1 2]
```

[11]:
```
B=A
B[1,1]=B[1,1]+1
A==B
```

[11]:
```
True
```

[12]:
```
B=deepcopy(A)
B[1,1]=B[1,1]+1
A==B
```

[12]:
```
False
```

[13]:
```
M=MatrixSpace(QQ,4,5)
A=M(A)
A.parent()
```

[13]:
```
Full MatrixSpace of 4 by 5 dense matrices over Rational Field
```

[14]:
```
A.rank()
```

[14]:
```
3
```

[15]:
```
A.left_kernel()
```

[15]:
```
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[ 0  1  0 -1]
```

[16]:
```
A.right_kernel()
```

[16]:
```
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[   1    0  1/4   -1 -1/2]
[   0    1 -1/2    0   -1]
```

[17]:
```
print(A.column_space())
print(A.row_space())
```

```
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[1 0 0 0]
[0 1 0 1]
[0 0 1 0]
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[   1    0    0    1    0]
[   0    1    0 -1/2    1]
[   0    0    1  1/2 -1/2]
```

[18]:
```
A=random_matrix(ZZ,4)
A
```

[18]:
```
[-3 -1  1 -2]
[ 2  0 -2 -1]
[ 3  5  7 -3]
[-1 -1  6  3]
```

[19]:
```
A.det()
```

[19]:
```
266
```

[20]:
```
A.inverse()
```

[20]:
```
[ -13/266  125/266    5/266      1/7]
[ -65/266 -173/266   25/266     -2/7]
[  11/133   17/133    6/133      1/7]
[   -5/19    -6/19    -1/19        0]
```

[21]:
```
A**(-1)
```

[21]:
```
[ -13/266  125/266    5/266      1/7]
[ -65/266 -173/266   25/266     -2/7]
[  11/133   17/133    6/133      1/7]
[   -5/19    -6/19    -1/19        0]
```

[22]:
```
B=A.LU()
pretty_print(B)
B[0]*B[1]*B[2]==A
```

```
(
[1 0 0 0]  [   1    0    0    0]  [   -3    -1     1    -2]
[0 0 0 1]  [  -1    1    0    0]  [    0     4     8    -5]
[0 1 0 0]  [ 1/3 -1/6    1    0]  [    0     0     7  17/6]
[0 0 1 0], [-2/3 -1/6    0    1], [    0     0     0 -19/6]
)
```

[22]:
```
True
```

[23]:
```
A.characteristic_polynomial()
```

[23]:
```
x^4 - 7*x^3 + 15*x^2 + 174*x + 266
```

[24]:
```
A.minimal_polynomial()
```

[24]:
```
x^4 - 7*x^3 + 15*x^2 + 174*x + 266
```

[25]:
```
A.eigenvalues()
```

[25]:
```
[-2.044243469722945? - 0.7809372787342890?*I, -2.044243469722945? +
0.7809372787342890?*I, 5.544243469722945? - 4.980733277515822?*I,
5.544243469722945? + 4.980733277515822?*I]
```

[26]:
```
(A*transpose(A))**2
```

[26]:
```
[  278  -207    23   325]
[ -207   431  -924 -1101]
[   23  -924  9115  3556]
[  325 -1101  3556  3139]
```

[27]:
```
K.<X>=FractionField(QQ['X'])
f=(5*X**5+3)/X
f(A)
```

[27]:
```
[ -196879/266  -340105/266  -688925/266       7843/7]
[ -634605/266  -861029/266  -304495/266      27189/7]
[  -93732/133  -294544/133 -1798807/133     -28802/7]
[   124720/19    172787/19     47307/19       -12625]
```

1.0.2 Exo 2

[28]:
```
A=matrix(ZZ,3,3,[1..9])
B=vector([1,1,1])
X=A.solve_right(B)
print(A,B,X)
print(A*X==B)
```

```
[1 2 3]
[4 5 6]
[7 8 9] (1, 1, 1) (-1, 1, 0)
True
```

[29]:
```
v=A.right_kernel().basis()
v
```

[29]:
```
[
(1, -2, 1)
]
```

[30]:
```
A*(X+3*v[0])==B
```

[30]:
```
True
```

1.0.3 Exo 3 (Gauss-Jordan)

[31]:
```
def GaussJordan(A):
    M=deepcopy(A)
    m,n=A.nrows(),A.ncols()
    r=-1  # indice du dernier pivot trouvé
    for j in range(n):
        # cherche premier pivot non nul dans la colonne j
        k=r+1
        while k<m and M[k,j]==0:
            k=k+1
        # si k=m, rien à faire, on passe à la colonne suivant
        if k<m:
            r=r+1    # on actualise le dernier pivot trouvé
            M[k]=M[k]/M[k,j]  # normalise pour avoir pivot = 1
            if r!=k:
                M[k],M[r]=M[r],M[k] # permute ligne r et k
            for i in range(m):
```

```
                if i!=r:
                    M[i]=M[i]-M[i,j]*M[r]   # annule M[i,j]
    return M
```

[32]:
```
A=random_matrix(GF(3),6,9)
M=GaussJordan(A)
N=A.rref()
print(A)
print()
print(M)
print()
print(N)
print()
print('rang de A:', A.rank())
```

```
[2 0 2 0 0 0 2 1 0]
[0 2 1 0 0 0 1 0 2]
[1 0 0 2 1 1 1 2 1]
[2 1 2 0 1 1 2 1 2]
[0 2 2 0 1 1 0 1 2]
[1 2 2 2 2 1 2 1 2]

[1 0 0 0 2 0 0 2 2]
[0 1 0 0 1 0 0 0 2]
[0 0 1 0 1 0 0 2 2]
[0 0 0 1 1 0 0 1 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 1 2]

[1 0 0 0 2 0 0 2 2]
[0 1 0 0 1 0 0 0 2]
[0 0 1 0 1 0 0 2 2]
[0 0 0 1 1 0 0 1 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 1 2]

rang de A: 6
```

[33]:
```
A=random_matrix(GF(3),5,3)
M=GaussJordan(A)
N=A.rref()
print(A)
print()
print(M)
print()
print(N)
print()
```

```
print('rang de A:', A.rank())
```

```
[0 1 1]
[2 0 1]
[0 2 0]
[1 1 0]
[2 2 0]

[1 0 0]
[0 1 0]
[0 0 1]
[0 0 0]
[0 0 0]

[1 0 0]
[0 1 0]
[0 0 1]
[0 0 0]
[0 0 0]

rang de A: 3
```

1.0.4 Exo 4 (algorithme de Keller-Gehrig)

[34]:
```
def FabriqueP(A,v):
    P=[v]
    n=A.nrows()
    for i in range(n-1):
        v=A*v
        P.append(v)
    return matrix(P).transpose()
```

[35]:
```
A=random_matrix(ZZ,4)
v=vector([1,0,0,0])
print(A)
print()
P=FabriqueP(A,v)
print(P)
print()
print(P**(-1)*A*P)
```

```
[ -4  -2   1   0]
[-86   1  -1   3]
[  1   3   0   2]
[ -4  -6  -2   1]

[     1     -4    189  -1516]
```

```
[     0    -86    245 -14161]
[     0      1   -270   1976]
[     0     -4    526  -1160]

[   0    0    0 1967]
[   1    0    0 -517]
[   0    1    0  155]
[   0    0    1   -2]
```

On peut faire du diviser pour régner. Insérer des colonnes serait pratique, mais on ne peut que insérer des lignes, ce qui est bien con, avec A.insert_row. Du coup, soit on fait A.transpose().insert_row(L,k).transpose(), soit on représente P par une liste de colonnes et on la transforme en matrice par matrix(P).transpose() si besoin (ce que l'on fera ici).

[36]:
```
## suppose n=puissance de 2

def FabriquePvite(A,v):
    M=deepcopy(A)
    n=M.nrows()
    P=[v] # représentée par liste de colonne pour pouvoir ajouter des colonnes␣
 ↪(aux chiottes Sage !)
    i=1
    while 2**i<=n:
        MP=M*matrix(P).transpose()
        P.extend(MP.columns())
        M=M**2
        i=i+1
    return matrix(P).transpose()
```

[37]:
```
P=FabriquePvite(A,v)
print(P)
print()
print(P**(-1)*A*P)
```

```
[     1     -4    189  -1516]
[     0    -86    245 -14161]
[     0      1   -270   1976]
[     0     -4    526  -1160]

[   0    0    0 1967]
[   1    0    0 -517]
[   0    1    0  155]
[   0    0    1   -2]
```

[38]:
```
K.<X>=A.base_ring()['X']

def PolChar(A,v):
    n=A.nrows()
```

```
    P=FabriqueP(A,v)
    C=P**(-1)*A*P
    Q=X**n-sum(C[i,n-1]*X**(i) for i in range(n))
    return(Q)
```

[39]:
```
A=random_matrix(ZZ,8)
v=random_vector(ZZ,8)
Chi1=PolChar(A,v)
Chi2=A.characteristic_polynomial()
print(Chi1)
print(Chi2)
```

```
X^8 - X^7 + 322*X^6 - 6607*X^5 - 60737*X^4 + 1367129*X^3 - 10376842*X^2 +
43526438*X - 187154198
x^8 - x^7 + 322*x^6 - 6607*x^5 - 60737*x^4 + 1367129*x^3 - 10376842*x^2 +
43526438*x - 187154198
```

[ ]: