El Zenary Ramel Aéro 2 C2

Aéro 2 — Analyse numérique (Ma 223, 2019-2020)

TP 1 — Méthode de Gauss pour la résolution de systèmes linéaires

1 Introduction

Dans ce TP, on va écrire des fonctions simples permettant d'appliquer l'algorithme de Gauss pour résoudre des systèmes.Pour ce faire, on utilisera Python et en particulier la librairie numpy.

2 L'algorithme de Gauss

Dans un premier temps, on va programmer la version simple de la méthode de Gauss, sans interversion de lignes, ni interversion de colonnes, autrement dit, sans choix de pivot.

Question 1

Programmer une fonction ReductionGauss(Aaug) qui rend la matrice obtenue après l'application de la méthode de Gauss à $\tilde{A}$, une matrice augmentée de format $(n,n+1)$.

Voici la capture du code qui correspond à cette question :

```
def ReductionGauss ( mat ) :

    li,co = np.shape( mat )

    for k in range ( 0 , li-1 ) :

        for i in range ( k+1 , li ) :

            g = mat [ i , k ] / mat[ k , k]

            mat[ i , : ] = mat [ i , : ] - g*mat[ k , : ]

    return mat
```

Question 2

Programmer une fonction ResolutionSystTriSup(Taug) qui rend la solution d'un système *TX* = *B*, où *T* est triangulaire supérieure. L'argument fourni à la fonction est la matrice augmentée de ce système Taug, de format (*n*,*n* + 1).

Voici la capture du code qui correspond à cette question :

```
def resolutionSysTriSup ( mat ) :

    P = np.array(mat)

    li , co = np.shape( P )

    global x

    x = np.zeros( li )

    for i in range ( li-1 , -1 , -1 ) :

        somme = 0

        for  k  in  range  ( li-1,i, -1 ) :

            somme = somme + P [ i , k ] * x[ k ]

        x [ i ] =( P [ i , li ] - somme ) / P [ i , i ]

    print ( 'Les solutions sont' , x )
    return x
```

Question 3

Programmer une fonction Gauss(A,B) qui rend la solution d'un système *AX* = *B* (*B* un vecteur colonne) en utilisant les fonctions programmées précédemment.

Voici la capture du code qui correspond à cette question :

```
def Gauss ( a , b ) :
    global x
    li,co = np.shape( a )
##    print(a)
##    a=np.ravel(a)
##    print(a)
##    print( b)
##    b=np.ravel(b)
##    V=[]
##    X=['a']
##    for x in a :
##        X.append(x)
##    for x in b :
##        V.append(x)
##    print(X)
##    print(V)
##    T=[]
##    print(li)
##
##    c=0
##    for  x in range (1,li*co+1) :
##
##        if x%co!=0:
##            T.append(X[x])
##
##        elif x%co==0:
##            T.append(X[x])
##            T.append(V[c])
##            c=c+1
##
##    a = np.array ([T])
##    print(T)
##    print(a)
##    a.resize(co,co+1)
##    print(a)
##    Ag=a
    Ag = np.concatenate ( ( a , b.T ) , axis = 1 )
    ReductionGauss (  Ag )
    resolutionSysTriSup ( Ag )
    print('Les resultat pour x avec Gauss sont:',x)
    Erreur = np.linalg.norm(Aaug@x-np.ravel(B))
    print(Erreur)
    return x
```

Il y a ici en commentaire une alternative à la fonction « concatenate », je me suis servis de liste dans lesquels j'ai introduis les valeur des matrice A et B, puis d'une liste final où, selon le rang de la liste dans lequel on se trouvait, un valeur de le matrice A ou de la matrice B est inséré.

Ce faisant, en transformant cette liste en matrice, puis en la redimensionnant, on trouve le même résultat qu'en utilisant la fonction « concatenate »

Question 4

Tester la résolution de *AX* = *B*, en prenant des matrices *A* et *B* aléatoires. On pourra par exemple, tester pour des matrices de taille 10, 100, 500.

Grace au code suivant :

```
def Q4 ( n ) :
    global Rd1
    global Bd1

    p = 100

    if n >= p :

        p = n

    print ( 'La taille de la matrice est' , n )

    Rd = np.random.randint ( 1 , p , size = ( n , n ) )
    Rd1=np.array(Rd,dtype=float)

    Bd = np.random.randint ( 1 , p , size = ( 1 , n ) )
    Bd1=np.array(Bd,dtype=float)

    print ( '""""""""""""""""""' )

    Gauss( Rd , Bd )



Q4(10)
Q4(100)
Q4(500)
```

Nous pouvons obtenir la solution de système de la forme AX=B pour des matrices de tailles 10, 100 et 500 comme nous pouvons le voir sur l'image ci-dessous :

(les resultats de la matrice de taille 500 ont volontairement été enlevé du compte rendu car trop volumineux).

```
La taille de la matrice est 10
""""""""""""""""""
Les solutions sont [ 0.35413631  1.03141635 -2.00797055 -0.2641753   2.19263592 -1.0153289
 -0.49785836 -0.60978213  2.4422188  -0.3220339 ]
Les resultat pour x avec Gauss sont: [ 0.35413631  1.03141635 -2.00797055 -0.2641753   2.19263592 -1.0153289
 -0.49785836 -0.60978213  2.4422188  -0.3220339 ]
10.205527261521377
La taille de la matrice est 100
""""""""""""""""""
Les solutions sont [-1.61247912  3.07030504 -3.09686334  1.0402887  -1.04234602  0.69448069
 -0.60681383  3.70177482 -0.64018935 -1.63868217 -2.07625424 -2.02949669
 -0.81637316  1.66061639  0.75384648  1.05398309  1.51992062 -2.69385079
  3.99034443 -0.37258594  0.97364899  1.44499016  3.39848877  0.915288
 -1.2921085  -1.48851889 -0.44361101  0.23667199  1.66659551  0.69539302
  0.33279836  1.50704841 -1.26216585  0.20196598 -1.6977845  -1.41062361
 -0.63258314 -0.83986785 -1.4894957   2.64792151  1.41472082 -0.63341319
  2.05548318 -3.47688033  1.31374708  3.55000113  2.69597004 -1.46371165
  1.76921477 -0.38757961 -1.72139649 -1.51484491  0.08159688  2.62237757
  3.48348649  1.44288707  1.50578273 -1.2457745  -0.09854644 -1.1083948
 -1.74958505 -1.46466264  2.96362395  0.0712322   0.11083614  0.54416794
  0.5128647   2.06566646 -1.76604389 -0.11305089  2.7345335  -4.51917333
  2.35746313  2.44906529 -4.88263555 -2.84482911  2.72588832 -0.92513119
 -0.22771164  2.1361037  -2.12850402 -0.84849926  1.03625131 -2.32185946
 -1.2054281  -2.11086336 -1.42419791 -0.19083279  1.39835475 -1.09055957
 -4.35528078  0.27673918 -0.38721268 -0.2555889   0.98866814 -1.19937415
  0.2024866  -0.21694632 -1.45864662  1.28853047]
Les resultat pour x avec Gauss sont: [-1.61247912  3.07030504 -3.09686334  1.0402887  -1.04234602  0.69448069
 -0.60681383  3.70177482 -0.64018935 -1.63868217 -2.07625424 -2.02949669
 -0.81637316  1.66061639  0.75384648  1.05398309  1.51992062 -2.69385079
  3.99034443 -0.37258594  0.97364899  1.44499016  3.39848877  0.915288
 -1.2921085  -1.48851889 -0.44361101  0.23667199  1.66659551  0.69539302
  0.33279836  1.50704841 -1.26216585  0.20196598 -1.6977845  -1.41062361
 -0.63258314 -0.83986785 -1.4894957   2.64792151  1.41472082 -0.63341319
  2.05548318 -3.47688033  1.31374708  3.55000113  2.69597004 -1.46371165
  1.76921477 -0.38757961 -1.72139649 -1.51484491  0.08159688  2.62237757
  3.48348649  1.44288707  1.50578273 -1.2457745  -0.09854644 -1.1083948
 -1.74958505 -1.46466264  2.96362395  0.0712322   0.11083614  0.54416794
  0.5128647   2.06566646 -1.76604389 -0.11305089  2.7345335  -4.51917333
  2.35746313  2.44906529 -4.88263555 -2.84482911  2.72588832 -0.92513119
 -0.22771164  2.1361037  -2.12850402 -0.84849926  1.03625131 -2.32185946
 -1.2054281  -2.11086336 -1.42419791 -0.19083279  1.39835475 -1.09055957
 -4.35528078  0.27673918 -0.38721268 -0.2555889   0.98866814 -1.19937415
  0.2024866  -0.21694632 -1.45864662  1.28853047]
508.32402608714733
La taille de la matrice est 500
""""""""""""""""""
Les solutions sont [ 2.52392686e-01  2.27834675e-01 -2.82504265e-03 -5.04368162e-01
```

On pourra présenter un tableau (voire une courbe) des temps de calcul pour différentes valeurs de *n*.

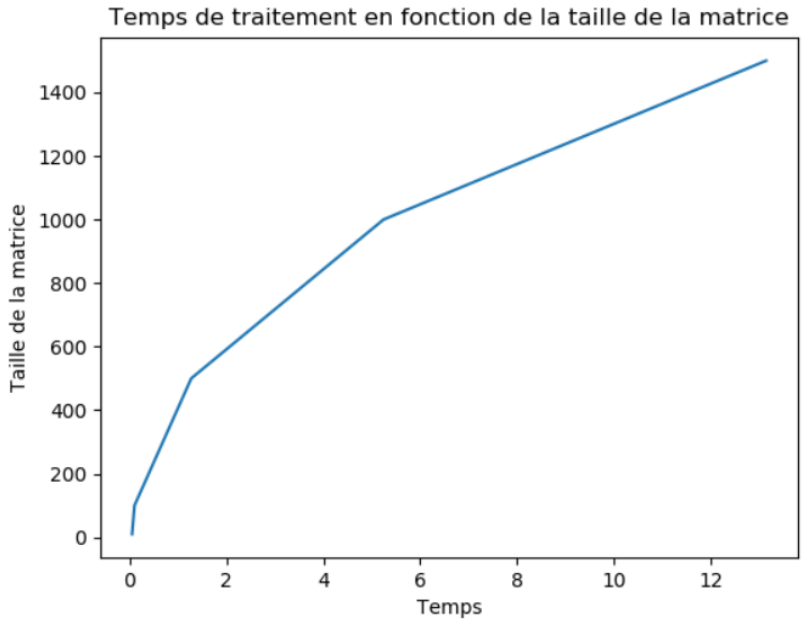
On pourra aussi donner une estimation de l'erreur commise, en considérant par exemple la valeur de *(AX – B)* pour la solution *X* calculée.

Pour pouvoir générer la courbe du temps de traitement j'ai utilisé ces 3 fonctions :

```
def TQ( n ) :
        global TC
        TC = []
        for i in range ( 10 ) :
                TM=[]
                T0=time.time()
                Q4( n )
                T1 = time.time()
                T = round ( T1-T0 , 5 )
                print( T )
                TM.append( n )
                TC.append( T )
                TT=[TM,TC]
                print('La matrice de taille' , TT[[0][0]] , 'prend' , TT[[1][0]] , 'seconde à etre calculé' )
def FMoyenneTemps () :
        global TC
        global Ttotal
        Ttot = 0
        for i in TC :
                Ttot = Ttot + i
        Ttotal = Ttot / len ( TC )
def TempsMoyen(n):
        global TC
        global Ttotal
        TQ(n)
        FMoyenneTemps()
        print(Ttotal)
        return Ttotal
```

La première fonction génère 10 matrices de tailles n pour en stockés le temps de traitement, la deuxième fonction en fait la moyenne, la dernière fonction compile les 2 précédentes pour afficher un temps de traitement moyen par rapport à 10 matrices de tailles n calculé.

Voici la courbe du temps de traitement en fonction de la taille de la matrice :

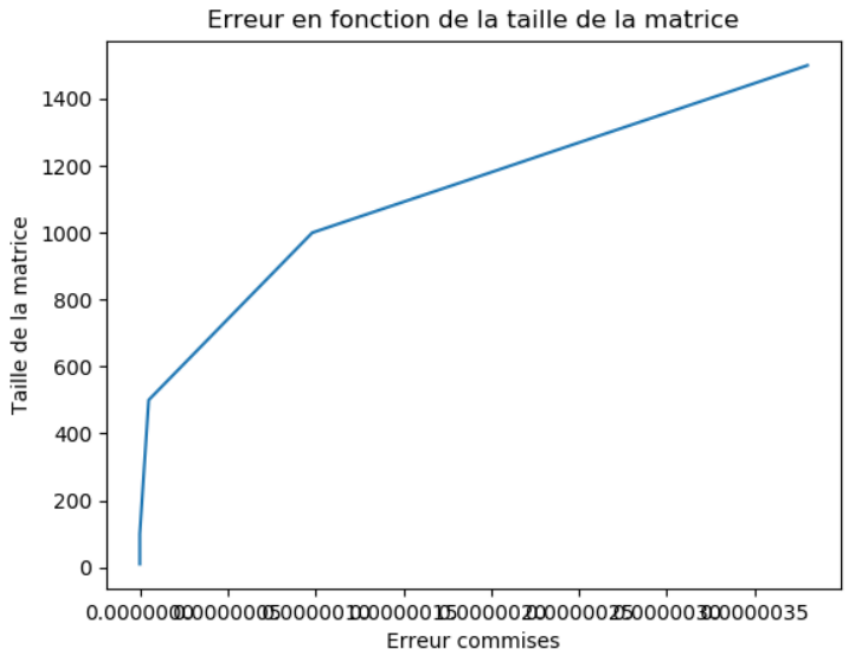


J'ai élaboré ce code pour gagner en précision quant à l'allure de la courbe.

Cette fois ci pour élaborer la courbe de l'erreur commise la fonction utilisée a été celle-ci :

```
def erreur ( n ) :
    global Rd1
    global Bd1
    global x
    p = 100
    if n >= p :
        p = n
    print('La taille de la matrice est',n)
    Q4(n)
    Erreur = np.linalg.norm(Rd1@Gauss(Rd1,Bd1)-np.ravel(Bd1))

    return Erreur
```

Voici la courbe de l'erreur commise en fonction de la taille de la matrice :



3 Décomposition LU

Question 1

Programmer une fonction [L,U]=DecompositionLU(A) qui rend la décomposition *LU* d'une matrice carrée *A* (on adaptera pour ce faire la fonction ReductionGauss(A).

Le code pour répondre à cette question est semblable à celui de la Question 1, 2eme partie, cependant une modification y a été ajouté afin de nous fournir la matrice Lower.

Nous avons simplement stocker les coefficient dans une liste, crée une matrice identité de taille semblable à la matrice fournis en argument, et nous avons distribuer les argument de la liste dans la matrices, dans le bon ordre en jouant sur les indices de boucles for et en créant les variables nécessaire.

Code servant à décomposer une matrice sous sa forme L x U :

```
def DecompositionLU ( mat ) :
    li , co = np.shape ( mat )
    L0=  []
    for k in range ( 0 , li-1 ) :
        for i in range ( k+1 , li ) :
            g = mat [ i , k ] / mat[ k , k]
            mat[ i , : ] = mat [ i , : ] - g*mat [ k , : ]
            L0.append ( g )
    I = np.identity ( li )
    d = 0
    m = 0
    for i in range ( co ) :
        m = m + 1
        for k in range ( li-1-i ) :
            I [ k + m , i ] = float ( L0 [ d ] )
            d = d + 1
    mat = np.array(mat,dtype=float)
    return mat , I
```

Question 2

Programmer une fonction ResolutionLU(L,U,B) qui rend la solution d'un système *AX* = *B* (avec la décomposition de *A* = *LU* fourni en argument).

Pour crée le code répondant à cette question, il suffisait dans crée une fonction semblable a « ResolutionSysTriSup » mais adapter pour résoudre des systèmes avec une matrice diagonale inférieur.

On appelle cette fonction « ResolutionSysTriInf » :

```
def resolutionSysTriInf ( mat ) :
    P = np.array ( mat )
    li , co = np.shape( P )
    global y
    y = np.zeros((li))
    for i in range ( li ) :
        somme = 0
        for k in range ( 0 , i+1 ) :
            somme = somme + y[k]*P[i,k]
        y[i]=(P[i,li]-somme)/P[i,i]
    print(y)
    return y
```

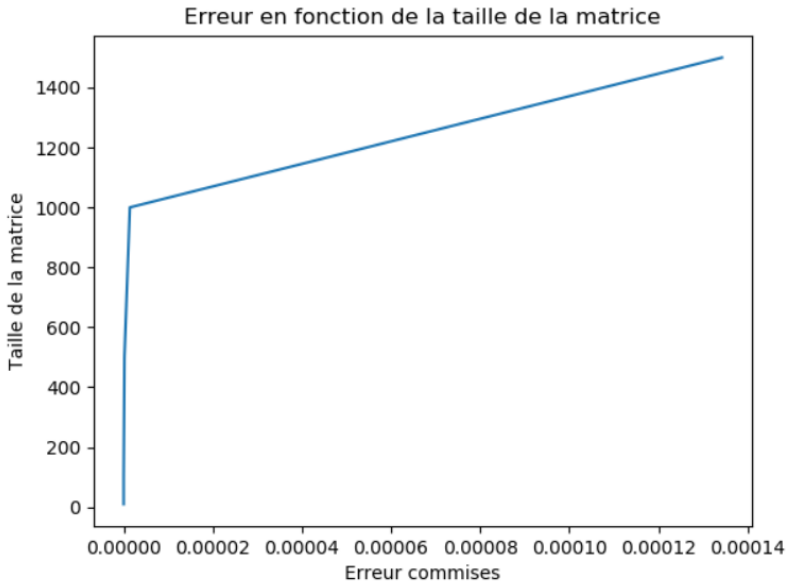
Avec y comme nouvel inconnu on résout l'équation de la même manière que dans la partie 2.

Voici le code nous permettant de résoudre des équations AX=B avec la méthode LU :

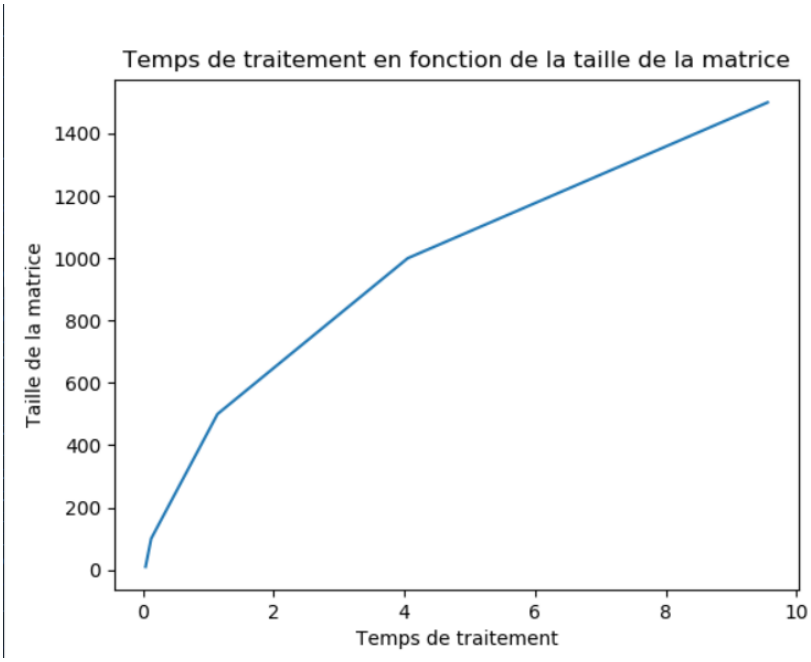
```
def ResolutionLU ( L , U , B ) :
    global y
    global x
    Ag = np.concatenate ( ( L , B.T ) , axis = 1 )
    li,co = np.shape(L)
    y = resolutionSysTriInf ( Ag )
    Ag1 = concate ( U , y )
    resolutionSysTriSup ( Ag1 )
    print('Les resultat pour x avec LU sont:',x)

def Resolution_LU(n) :
    global x
    global y
    p = 100
    if n >= p :
        p = n
    print ( 'La taille de la matrice est' , n )
    Rd = np.random.randint ( 1 , p , size = ( n , n ) )
    Rd1=np.array(Rd,dtype=float)
    Bd = np.random.randint ( 1 , p , size = ( 1 , n ) )
    Bd1=np.array(Bd,dtype=float)

    U1 , L1 = DecompositionLU ( Rd1 )

    ResolutionLU ( L1 , U1 , Bd1 )
    Erreur = np.linalg.norm(L1@U1@x-np.ravel(Bd1))


    print('x:',x)
```

(Pour le code ci-dessus je me suis servis de mon alternative à la fonction concatenate que j'ai mis en définiction et nomé « concate »)

Voici la courbe de l'erreur commise en fonction de la taille de la matrice correspondante :



Voici la courbe du temps de traitement fonction de la taille de la matrice correspondante :



On observe ici que l'erreur commise et le train de traitement en fonction de la matrice avec la méthode, sont moins important qu'avec la résolution du pivot de Gauss traditionnelle.

4 Variantes de l'algorithme de Gauss

Question 1

Programmer une fonction GaussChoixPivotPartiel(A,B) pour résoudre $AX = B$ avec choix de pivot partiel. C'est-à-dire que l'on utilise des échanges de lignes, de sorte que le pivot soit choisi de plus grand module possible au sein de la colonne.

Pour répondre à cette question, c'est la partie réduction de la matrice qu'il à fallu modifier, en effet pour cette fonction avant chaque pivot de Gauss effectué, un tri des lignes a été élaboré afin d'avoir en première position la ligne présentant à sa tête le plus grand coefficient en valeur absolue.

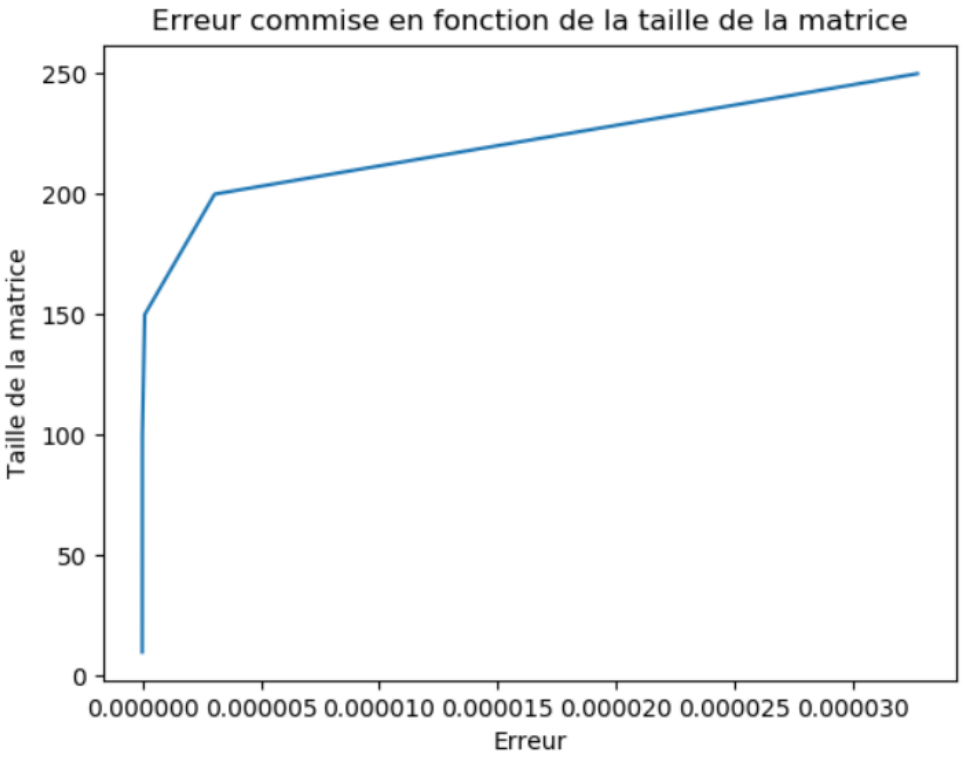
Ainsi la fonction ReductionGaussPartiel fut élaboré.

Cependant pour ce qui est de la fonction suivante, elle est similaire à la fonction du pivot de gauss dans la partie 2 à la simple est unique différence que nous avons utilisé la réduction en pivot partiel.

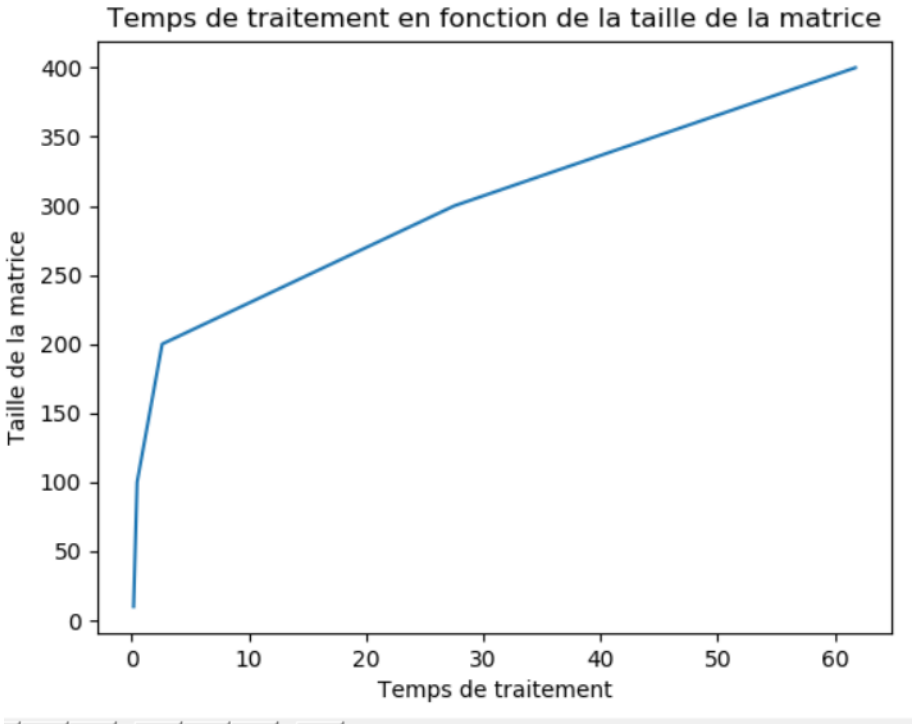
Voici le code :

```
def ReductionGaussPartiel ( mat ) :
    print(mat)

    li,co = np.shape( mat )

    for k in range ( 0 , li - 1 ) :

        for i in range ( k + 1 , li ) :

            for q in range (k,li-1) :
                sup = k
                if abs(mat[q,k]) < abs(mat [q+1,k]):
                    sup = q+1
            if sup!=k :
                Nmat=np.array(mat)
                Nmat1=np.array(mat)

                mat [k,:]=Nmat[sup,:]
                mat[sup,:]=Nmat1[k,:]
                sup=k



            g = mat [ i , k ] / mat[ k , k]

            mat[ i , : ] = mat [ i , : ] - g*mat[ k , : ]
    print(mat)

    return mat


def GaussChoixPivotPartiel ( A , B ) :
    global x
    Ag = np.concatenate ( ( A , B.T ) , axis = 1 )
    Ag=np.array(Ag,dtype=float)
    ReductionGaussPartiel (  Ag )
    print(Ag)
    resolutionSysTriSup ( Ag )

    print('Les resultat pour x avec Gausspartiel sont:',x)
    Erreur = np.linalg.norm(Aaug@x-np.ravel(B))
    print(Erreur)
    return x
```

Voici la courbe de l'erreur commise correspondante :



Voici la courbe du temps de traitement correspondante :



On observe ici que le temps de traitement est lourd même pour des matrices de petites tailles, il en est de même pour l'erreur commise.

Question 2

Programmer une fonction GaussChoixPivotTotal(A,B) qui rend la solution d'un système *AX* = *B* (*B* un vecteur colonne) avec la méthode de Gauss avec choix de pivot total.

Cette fois ci pour le pivot Total, le triage de la matrice ne s'est pas fait qu'en intervertissant les lignes mais également les colonnes.

En effet le coefficient est d'abord comparé avec les éléments de la ligne, ainsi les colonnes sont interverties, puis il est comparé avec les éléments de la colonne et donc les lignes sont interverties.

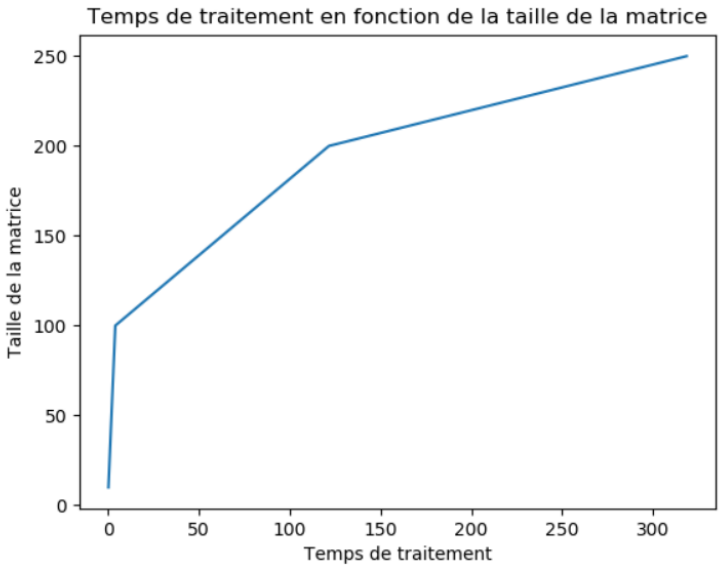
Cependant lors de cette manipulation, le vecteur colonnes qui nous est retourné à la fin n'est pas fournis dans le bon ordre.

Une liste a donc été fournis afin de surveiller les échanges de colonnes pour pouvoir remettre les coefficients de la matrice solution dans le bon ordre.

Voici le code :

```
def ReductionGaussTotal ( mat ) :
    global Ini
    Ini=[]
    li,co = np.shape( mat )
    for k in range ( 0 , li - 1 ) :
        for i in range ( k +1 , li ) :
            for q in range (k,li-1) :
                sup1=q
                if abs(mat[k,q]) < abs(mat [k,q+1]):
                    sup1 = q+1
                if sup1 !=q :
                    Ini.append([q,sup1])
                    Nmat1=np.array(mat)
                    Nmatt1=np.array(mat)
                    mat [:,q]=Nmat1[:,sup1]
                    mat[:,sup1]=Nmatt1[:,q]
                    sup1=q
                sup = k
                if abs(mat[q,k]) < abs(mat [q+1,k]):
                    sup = q+1
            if sup!=k :
                Nmat=np.array(mat)
                Nmatt=np.array(mat)
                mat [k,:]=Nmat[sup,:]
                mat[sup,:]=Nmatt[k,:]
                sup=k
            g = mat [ i , k ] / mat[ k , k]
            mat[ i , : ] = mat [ i , : ] - g*mat[ k , : ]
    return mat
def GaussChoixPivotTotal ( A , B ) :
    global x
    global Ini
    global Fin
    Ag = np.concatenate ( ( A , B.T ) , axis = 1 )
    Ag=np.array(Ag,dtype=float)
    ReductionGaussTotal (  Ag )
    print(Ag)
    resolutionSysTriSup ( Ag )
    for e in reversed(Ini):
        h=copy.copy(x[e[0]])
        x[e[0]]=x[e[1]]
        x[e[1]]=h
    print('Les resultat pour x avec Gausstotal sont:',x)
    Erreur = np.linalg.norm(Aaug@x-np.ravel(B))
    print(Erreur)
    return x
```

Voici la courbe du temps de traitement correspondante :



On observe qu'avec la méthode pivot total le temps de traitement est très élevé même pour des matrices de petites tailles.

Annexe :

Voici les codes utilisés pour générer les courbes d'erreur et de temps pour :

La résolution LU :

```
def Resolution_LU(n) :
    global x
    global y
    p = 100
    if n >= p :
        p = n
    print ( 'La taille de la matrice est' , n )
    Rd = np.random.randint ( 1 , p , size = ( n , n ) )
    Rd1=np.array(Rd,dtype=float)
    Bd = np.random.randint ( 1 , p , size = ( 1 , n ) )
    Bd1=np.array(Bd,dtype=float)

    U1 , L1 = DecompositionLU ( Rd1 )

    ResolutionLU ( L1 , U1 , Bd1 )
    Erreur = np.linalg.norm(L1@U1@x-np.ravel(Bd1))

    print(Erreur)
    print('x:',x)
    return (Erreur)

def TLU( n ) :
    T0=time.time()
    Resolution_LU( n )
    T1 = time.time()
    T = round ( T1-T0 , 5 )
    return T
```

La résolution en Pivot Partiel :

```
def Resolution_PIVOTPARTIEL(n) :
    global x
    global y
    p = 100
    if n >= p :
        p = n
    print ( 'La taille de la matrice est' , n )
    Rd = np.random.randint ( 1 , p , size = ( n , n ) )
    Rd1=np.array(Rd,dtype=float)
    Bd = np.random.randint ( 1 , p , size = ( 1 , n ) )
    Bd1=np.array(Bd,dtype=float)

    GaussChoixPivotPartiel(Rd1 , Bd1 )
    Erreur = np.linalg.norm(Rd1@x-np.ravel(Bd1))

    print(Erreur)
    print('x:',x)
    return (Erreur)

def TPP( n ) :
    T0=time.time()
    Resolution_PIVOTPARTIEL( n )
    T1 = time.time()
    T = round ( T1-T0 , 5 )
    return T
```

La résolution en Pivot Total :

```
def Resolution_PIVOTTOTAL(n) :
    global x
    global y
    p = 100
    if n >= p :
        p = n
    print ( 'La taille de la matrice est' , n )
    Rd = np.random.randint ( 1 , p , size = ( n , n ) )
    Rd1=np.array(Rd,dtype=float)
    Bd = np.random.randint ( 1 , p , size = ( 1 , n ) )
    Bd1=np.array(Bd,dtype=float)

    GaussChoixPivotTotal(Rd1 , Bd1 )
    Erreur = np.linalg.norm(Rd1@x-np.ravel(Bd1))

    print(Erreur)
    print('x:',x)
    return (Erreur)


def TPP( n ) :
    T0=time.time()
    Resolution_PIVOTTOTAL( n )
    T1 = time.time()
    T = round ( T1-T0 , 5 )
    return T
```