Institut Polytechnique des Sciences Avancées
Campus de Paris
63 bis, Boulevard de Brandebourg
94 200 Ivry-sur-Seine

Etablissement Privé d'Enseignement Supérieur
SIRET 433 695632 00011 - APE 803Z

14/10/2021

TP 2 d'analyse linéaire numérique [Ma313]

Décomposition QR, algorithme de Gram-Schmidt

M. AL SAYED

Léa DUPIN
lea.dupin@ipsa.fr

Aéro 3 Groupe F2
IPSA PARIS | 2021 - 2022

Table des matières

Exercice 1 : Décomposition GS

Soit A une matrice carrée. On appelle décomposition QR de A la donnée de Q et R telles que :

- $A = Q * R$
- Q est une matrice orthogonale
- R est une matrice triangulaire supérieure

On souhaite dans un premier temps programmer une fonction *DécompositionGS* qui prend en entrée une matrice A inversible et rend Q et R issues de l'algorithme de Gram-Schmidt.

A partir de l'algorithme fourni dans le TD, nous arrivons définir la fonction telle que :

```
def DecompositionGS(A):

    n, m = A.shape
    Q = np.zeros((n, n))
    R = np.zeros((n, n))
    w = np.zeros(n)

    for j in range (0, n):

        for i in range (0, j):

                if i < j:
                    R[i,j] = np.dot(A[:,j], Q[:,i])
                else:
                    pass
        S = 0
        for k in range (0, j):

            S = S + (R[k, j] * Q[:, k])


        w = A[:, j] - S

        R[j, j] = np.linalg.norm(w)

        Q[:, j] = (1 / R[j, j]) * w

    return Q, R, S
```

En vérifiant avec les exemples vus en cours, nous retrouvons bien les matrices attendues.

Exercice 2 : Résolution GS

Nous écrivons ensuite une fonction *ResolGS* qui résout un système linéaire $Ax = b$ en utilisant la décomposition $Q * R$ de A.

```
def ResolGS(A, b):

    Q = DecompositionGS(A)[0]
    R = DecompositionGS(A)[1]
    Y = np.dot(np.transpose(Q), b)
    T = resoltrisup(R, Y)
    print("Résolution triangulaire supérieure : ", T)
    return(T)
```

Exercice 3 : Comparaisons de temps de calcul

Afin de mesurer le temps de calcul, nous utilisons la fonction *time.process_time()* qui, appelée deux fois successive, permet de mesurer le temps entre deux appels (temps mis pour exécuter les lignes de codes se situant entre les deux appels).

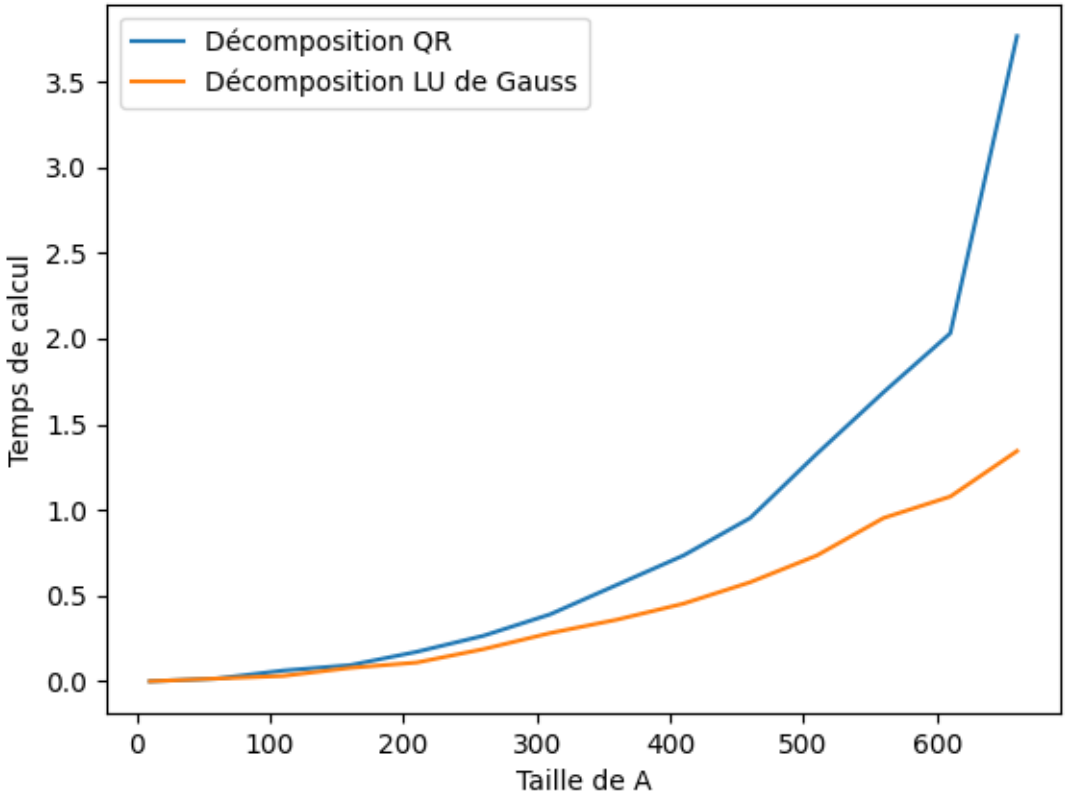
```
def TpsResolQR(n):

    start = time.process_time()
    ResolGS(RandomA(n), Randomb(n))
    end = time.process_time()
    t = end - start

    return t
```

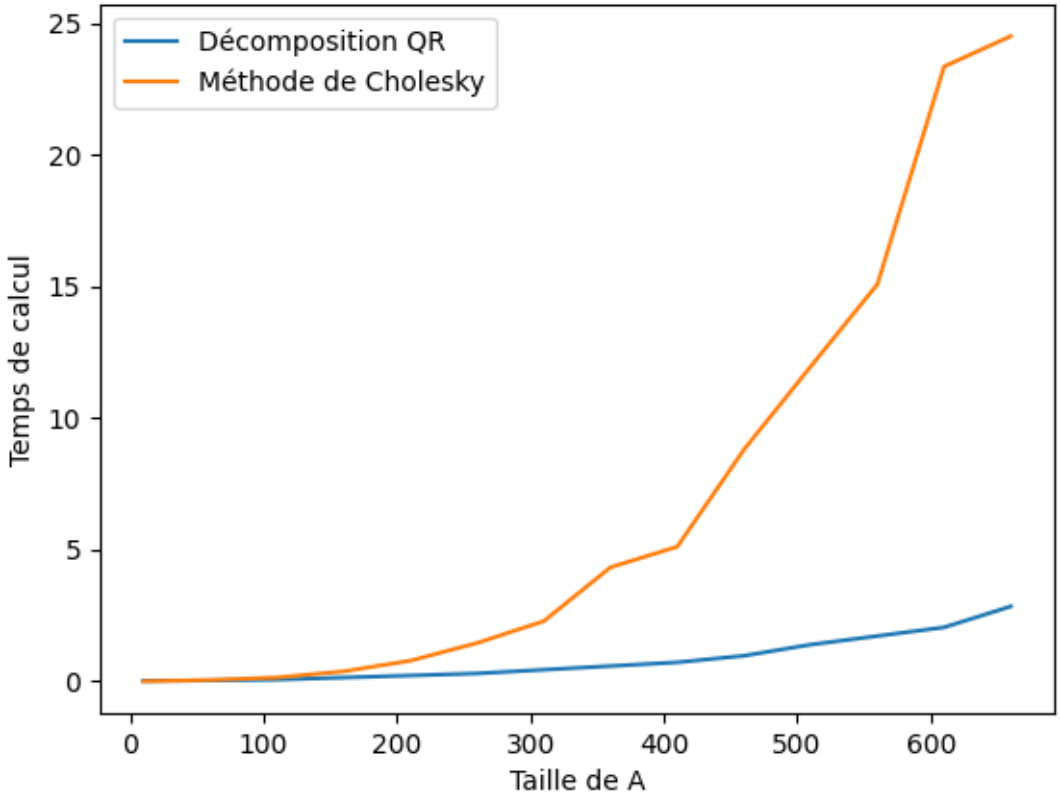
I – Comparaison avec la décomposition LU de la méthode de Gauss

En réutilisant les algorithmes de l'an dernier, nous pouvons comparer le temps d'exécution des deux méthodes :



On remarque que la méthode LU est plus rapide que la décomposition QR que nous avons programmé.

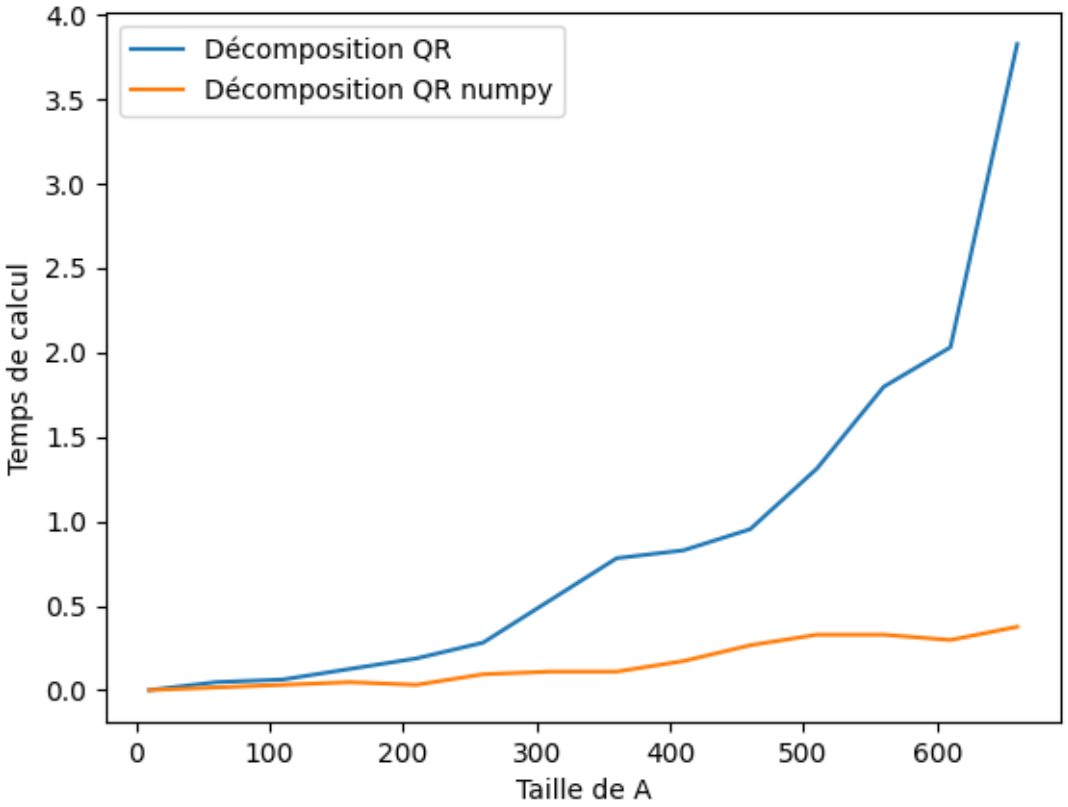
II – Comparaison avec la méthode de Cholesky



La décomposition QR est plus rapide cette fois-ci.

III – Comparaison avec la décomposition QR du module numpy

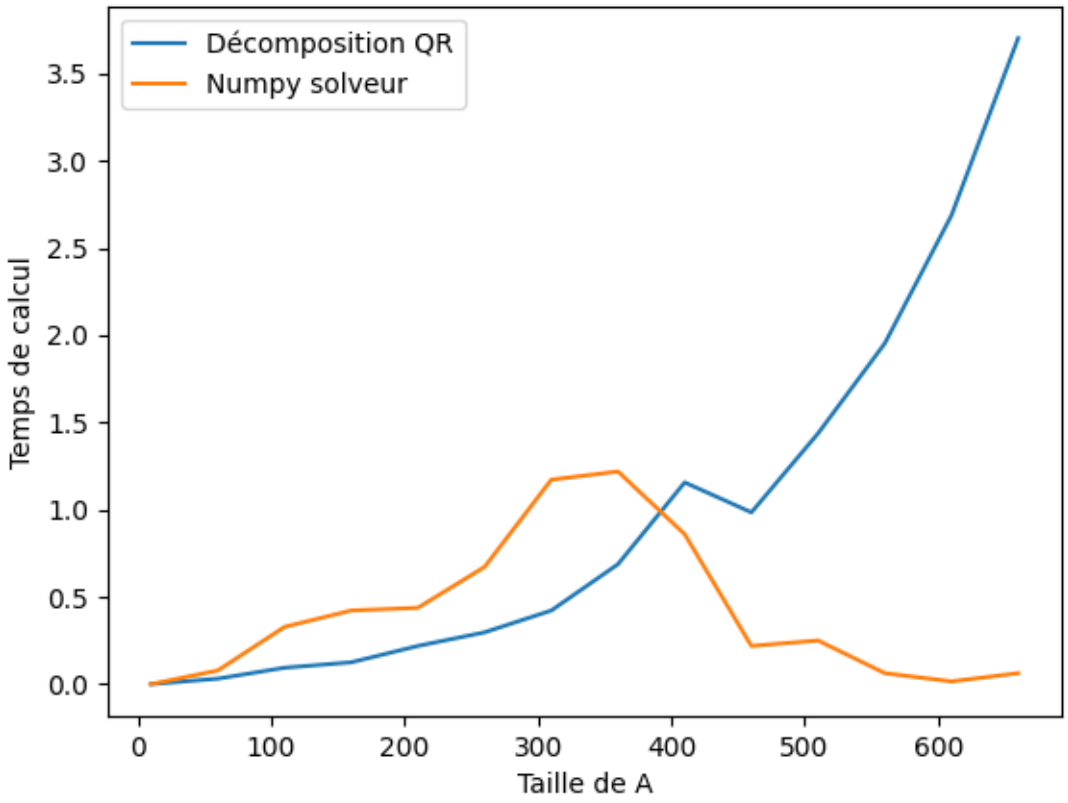
On utilise la fonction *numpy.linalg.qr* afin d'optimiser la décomposition QR de l'algorithme étudié. Cette fonction utilise la méthode de Householder.



La méthode de numpy est bien plus rapide que la nôtre. Deux hypothèses : la nôtre est mal optimisée, ou bien la méthode de Householder est naturellement plus rapide que la nôtre.

IV – Comparaison avec le solveur de numpy

Nous utilisons la fonction *numpy.linalg.solve* qui est connue comme étant la plus précise et la plus rapide :



Mis à part un léger pic au milieu de notre graphique, sur le long terme cette méthode sera effectivement plus rapide. On peut supposer que ce pic est dû à une erreur de programmation de ma part (programme probablement mal optimisé).

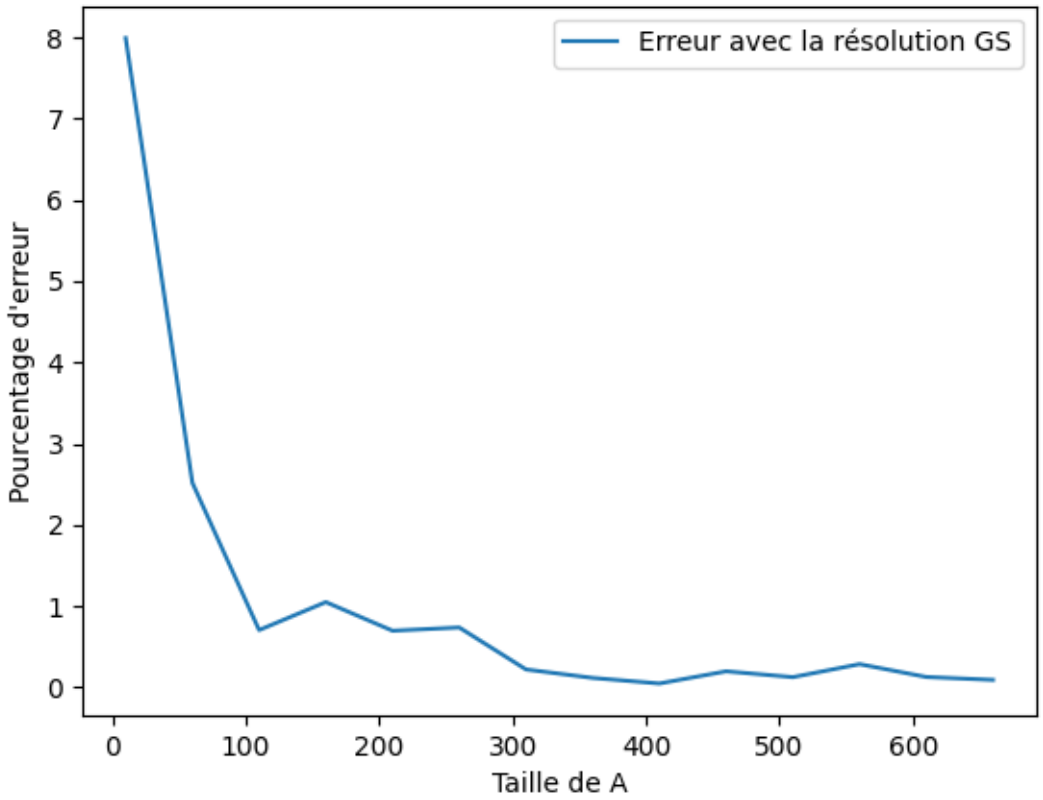
Calcul des erreurs

Afin d'évaluer la précision des différentes méthodes utilisées, nous travaillons également sur l'affichage de l'erreur à travers de nouveaux graphiques. Pour la calculer, nous utilisons le calcul de la norme vectoriel de $\|A * x - b\|$. En effet, plus la méthode est précise, plus cette valeur doit s'approcher de zéro (équation résolue ⇔ norme = 0).

Nous calculons ensuite le pourcentage d'erreur avec la formule suivante, en prenant comme référence la solution donnée par le solveur numpy de la fonction *numpy.linalg.solve* :

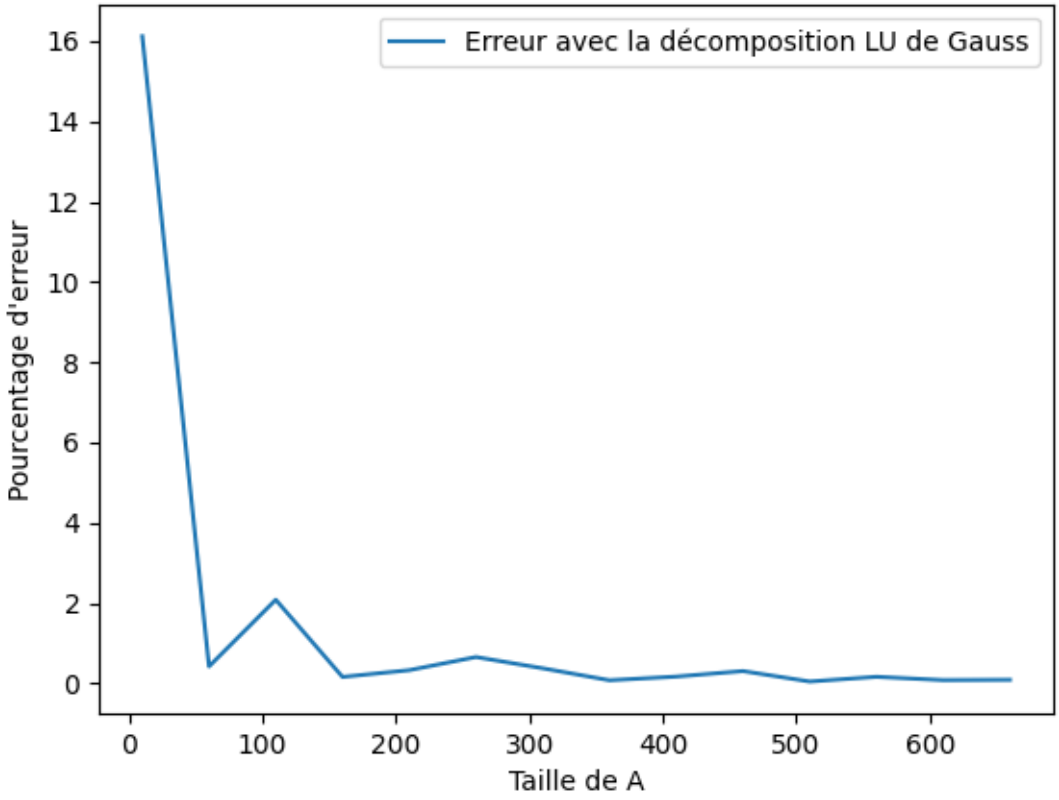
$$Pourcentage = \frac{|norme\ solveur\ numpy - norme\ méthode\ étudiée|}{norme\ solveur\ numpy} * 100$$

Erreur donnée par la décomposition QR de l'algorithme de Gram-Schmidt



On peut dire que l'erreur reste toujours dans le domaine de l'acceptable $(8\% < 10\%)$ avec des erreurs plus prononcées avec des matrices A de petite taille. Au-delà d'une matrice de taille $n = 100$, l'erreur reste en dessous de 1%.

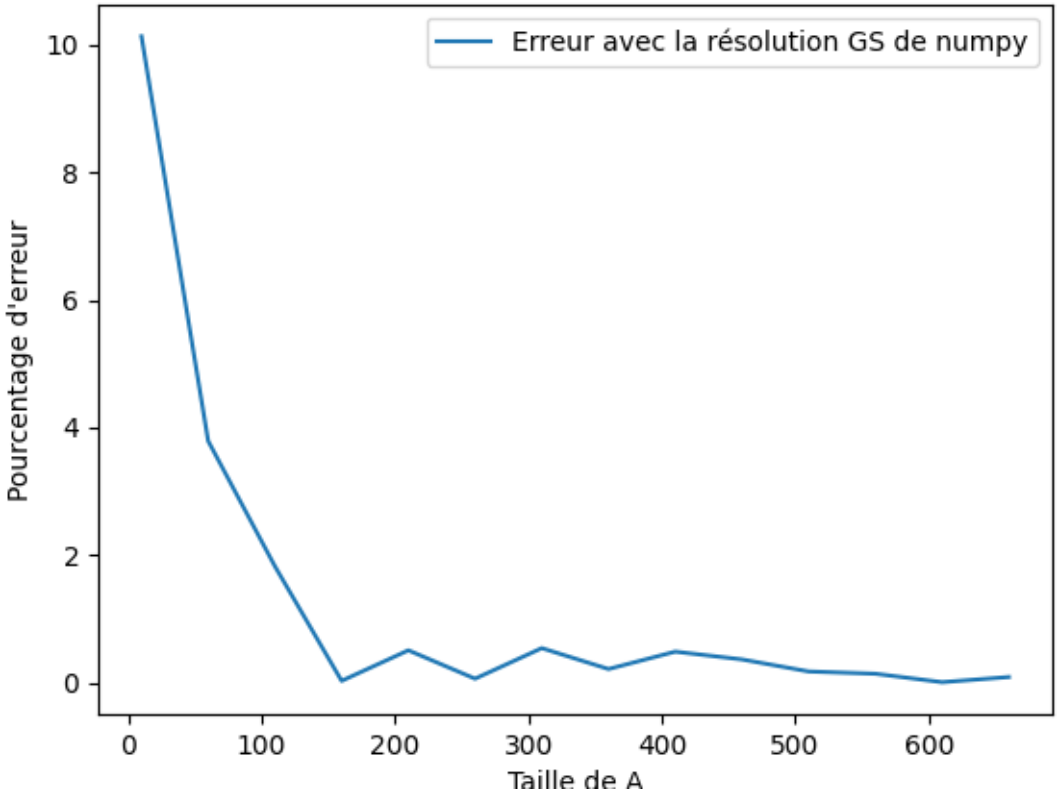
Erreur donnée par la décomposition LU de l'algorithme de Gauss



L'allure de la courbe d'erreur de cette méthode est comparable à celle de la méthode précédente, à la différence d'une pointe à 16% ici et non plus 8%. Cependant, d'une manière générale, on peut dire que les deux méthodes sont comparables en termes de précision de résultats.

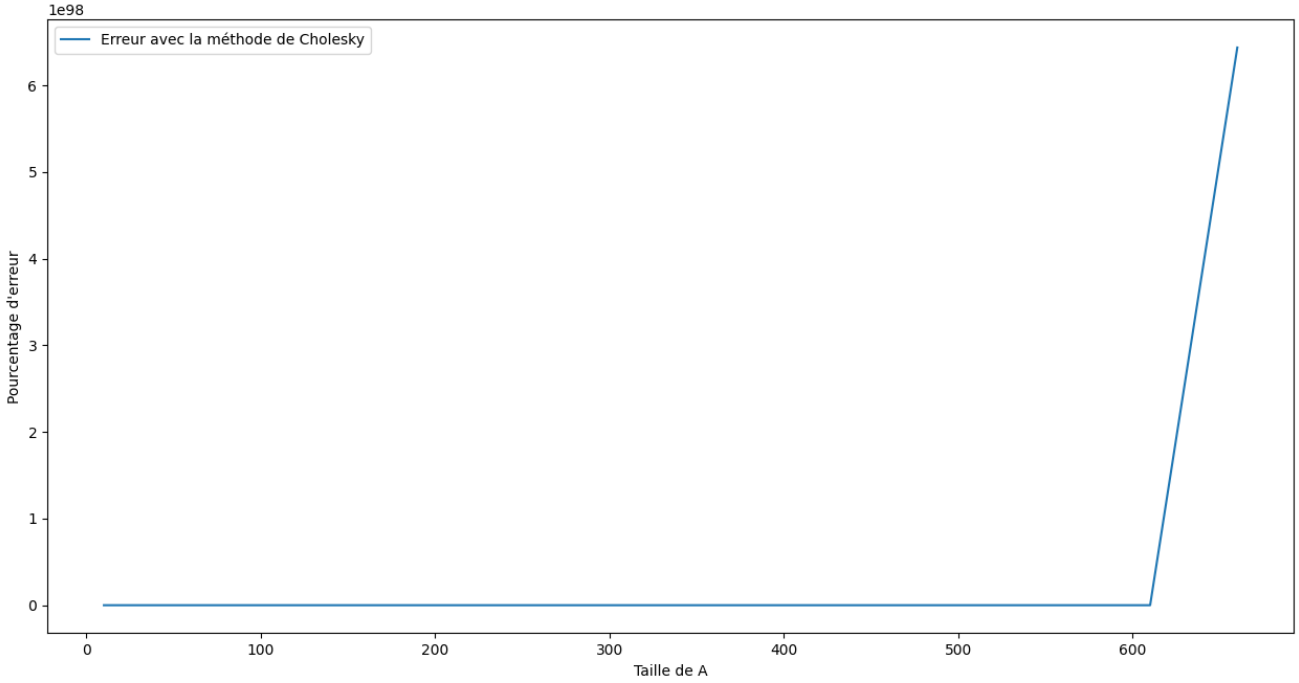
Au-delà d'une matrice de taille $n = 100$ l'erreur ne dépasse pas les 2% et chute même ensuite.

Erreur donnée par la décomposition QR du module numpy



Encore une fois, l'erreur issue de cette méthode est largement comparable aux deux précédentes. Par rapport à la méthode de décomposition QR que nous avons programmé, elle semble cependant plus rapidement fiable, mis à part un pic de 10% au départ, contre 8% pour notre algorithme. La suite de la courbe semble cependant plus constante.

Erreurs obtenues avec la méthode de Cholesky



Pour cet algorithme cependant, il y a de manière très visible un soucis dans notre programme : l'erreur est supérieure à $6 * 10^{98}$ % sur les derniers calculs. Elle semble malgré tout très faible auparavant.

Nous pouvons supposer qu'il s'agisse d'une erreur dans l'enregistrement des résultats de notre calcul d'erreur car en la vérifiant manuellement avec une décomposition connue, elle retourne un résultat juste.

Le test mené avec des matrices de tailles $n < 500$ a donné le même résultat, avec un pic d'erreur à $4 * 10^{68}$ %.

Annexes : codes

Exercice 1 : Décomposition GS

```
def DecompositionGS(A):

    n, m = A.shape
    Q = np.zeros((n, n))
    R = np.zeros((n, n))
    w = np.zeros(n)

    for j in range (0, n):

        for i in range (0, j):

                if i < j:
                    R[i,j] = np.dot(A[:,j], Q[:,i])
                else:
                    pass
        S = 0
        for k in range (0, j):

            S = S + (R[k, j] * Q[:, k])


        w = A[:, j] - S

        R[j, j] = np.linalg.norm(w)

        Q[:, j] = (1 / R[j, j]) * w

    return Q, R, S

A = np.array([[6, 6, 16], [-3, -9, -2], [6, -6, -8]])
print("Décomposition GS : ", DecompositionGS(A))
```

Exercice 2 : Résolution GS

```python
def resoltrisup(T, b):

    n, m = T.shape
    x = np.zeros(n)

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

        S = T[i, i + 1:]@x[i + 1:]

        x[i] = (1 / T[i, i]) * (b[i] - S)

    return x

def ResolGS(A, b):

    Q = DecompositionGS(A)[0]
    R = DecompositionGS(A)[1]
    Y = np.dot(np.transpose(Q), b)
    T = resoltrisup(R, Y)
    print("Résolution triangulaire supérieure : ", T)
    return(T)
```

Exercice 3 : Décomposition de Gauss

```python
def Gauss(A, b):

    A = A.copy()
    b = b.copy()
    n = b.size

    for i in range(n):

        for j in range(i + 1, n):

            g = A[j, i] / A[i, i]
            A[j, :] = A[j, :] - g * A[i, :]
            b[j] = b[j] - g * b[i]

    x = resoltrisup(A, b)

    return x
```

Exercice 3 : Méthode de Cholesky

```python
def Cholesky(A, b):
    n, m = A.shape
    if n != m:
        print ("A n'est pas carrée = problème")
        return

    L = np.zeros((n, n))

    Skk = 0
    Sik = 0

    for k in range(n):

        Skk = 0

        for j in range(0, k):

            Skk += (L[k, j]) ** 2

        L[k, k] = np.sqrt(A[k, k] - Skk)

        for i in range(n):

            Sik = 0

            if i > k:
                for j in range(0, k):
                    Sik += (L[i, j]) * (L[k, j])
                L[i, k] = (A[i, k] - Sik) / L[k, k]
    return L
```

Exercice 3 : Décomposition GS de numpy

```python
def ResolGS_np(A, b):

    Q = np.linalg.qr(A)[0]
    R = np.linalg.qr(A)[1]
    Y = np.dot(np.transpose(Q), b)
    T = resoltrisup(R, Y)
    print("Résolution triangulaire supérieure : ", T)
    return(T)
```

Exemple de calcul d'erreur avec la décomposition GS programmée

```python
def ErreurSolve(n):

    A = Random_2A(n)
    b = Random_2b(n)
    x1 = ResolGS(A, b)
    x2 = np.linalg.solve(A, b)
    exp = np.linalg.norm(A * x1 - b)
    accepted = np.linalg.norm(A * x2 - b)
    erreur = (np.abs(accepted - exp) / accepted) * 100

    return erreur
```

Code complet

Le code complet pour les temps de calcul est disponible à l'adresse GitHub suivante :

https://github.com/lea-dup40/Ma313----TP02.git

Fichiers contenus dans le répertoire :

- TP2_LéaDupin_Aero3F2_Part1.py : Partie du TP où se situent les 3 exercices demandés dans le TP
- TP2_LéaDupin_Aero3F2_Part2.py : Partie du TP où se situent les calculs d'erreurs
- TP2_LéaDupin_Aero3F2_PDF.pdf : Compte rendu du TP (ce document)