Pivot_Gauss

April 3, 2023

1 Le pivot de Gauss

Marc Lorenzi

21 février 2020

[1]:
```
import random
import math
import time
import matplotlib.pyplot as plt
```

[60]:
```
plt.rcParams['figure.figsize'] = (8, 4)
```

L'algorithme du pivot de Gauss est un vaste sujet. Nous allons dans ce notebook nous intéresser à cet algorithme dans un cas particulier, celui des matrices inversibles.

Soit $A \in GL_n(\mathbb{R})$ une matrice inversible. Soit $B \in \mathcal{M}_{n,q}(\mathbb{R})$. Considérons l'équation d'inconnue $X \in \mathcal{M}_{nq}(\mathbb{R})$

$$AX = B$$

Cette équation admet une unique solution, à savoir $X = A^{-1}B$.

Nous allons voir comment l'algorithme du pivot permet d'obtenir X efficacement. Nous verrons également deux effets secondaires de cet algorithme : - Le calcul de A^{-1}. - Le calcul de $\det A$.

Dans ce qui va suivre nous travaillerons sur des matrices de flottants, représentées par des listes de listes en Python. **Pour nous conformer aux conventions du langage Python, les indices de lignes et de colonnes des matrices commenceront systématiquement à 0.**

Avant toute chose, mettons en place quelques fonctions qui nous seront fort utiles.

1.1 1. Quelques fonctions utiles

1.1.1 1.1 Nombre de lignes, de colonnes

Les fonctions `nb_lig` et `nb_col` renvoient le nombre de lignes et colonnes de la matrice A.

[3]:
```
def nb_lig(A): return len(A)
def nb_col(A): return len(A[0])
```

[4]:
```
A = [[1, 2, 3], [4, 5, 6]]
print(nb_lig(A), nb_col(A))
```

```
2 3
```

1.1.2 1.2 Afficher une matrice

Voici une fonction qui affiche de façon lisible une matrice de taille raisonnable. Par défaut, les coefficients de A sont affichés avec 5 chiffres après la virgule. Changez ce nombre si cela vous fait plaisir.

[23]:
```
def prnt(A):
    p = nb_lig(A)
    q = nb_col(A)
    s = q * ('+' + 12 * '-') + '+'
    for i in range(p):
        print(s)
        for j in range(q):
            print('|%+10.5f  ' % A[i][j], end='')
        print('|')
    print(s)
```

[24]:
```
prnt(A)
```

```
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
|  +4.00000  |  +5.00000  |  +6.00000  |
+------------+------------+------------+
```

1.1.3 1.3 Matrices aléatoires

Il nous sera fort utile de disposer de matrices "aléatoires" pour tester nos fonctions. La fonction `randmat` renvoie une matrice de taille $p \times q$ dont les coefficients sont des réels suivant une loi uniforme sur l'intervalle $[-r, r]$ (où, par défaut, $r = 1$).

[25]:
```
def randmat(p, q, r=1):
    A = [q * [0] for i in range(p)]
    for i in range(p):
        for j in range(q):
            A[i][j] = random.uniform(-r, r)
    return A
```

[26]:
```
prnt(randmat(3, 5))
```

```
+------------+------------+------------+------------+------------+
|  +0.68922  |  +0.45985  |  -0.22195  |  -0.86794  |  +0.79359  |
+------------+------------+------------+------------+------------+
|  +0.00591  |  +0.05023  |  -0.14001  |  +0.74599  |  -0.20269  |
+------------+------------+------------+------------+------------+
|  +0.12358  |  -0.92569  |  -0.52789  |  -0.31744  |  -0.77628  |
+------------+------------+------------+------------+------------+
```

1.1.4 1.4 Opérations matricielles

La fonction `eye` renvoie la matrice identité d'ordre n.

[27]:
```
def eye(n):
    A = [n * [0] for i in range(n)]
    for i in range(n): A[i][i] = 1
    return A
```

[28]:
```
prnt(eye(4))
```

```
+------------+------------+------------+------------+
|  +1.00000  |  +0.00000  |  +0.00000  |  +0.00000  |
+------------+------------+------------+------------+
|  +0.00000  |  +1.00000  |  +0.00000  |  +0.00000  |
+------------+------------+------------+------------+
|  +0.00000  |  +0.00000  |  +1.00000  |  +0.00000  |
+------------+------------+------------+------------+
|  +0.00000  |  +0.00000  |  +0.00000  |  +1.00000  |
+------------+------------+------------+------------+
```

La fonction `prodmat` renvoie le produit des matrices A et B. Si $A \in \mathcal{M}_{pq}(\mathbb{R})$ et $b \in \mathcal{M}_{qr}(\mathbb{R})$, la fonction `prodmat` effectue $2pqr$ opérations sur des flottants. En particulier, lorsque $p = q = r = n$, la complexité de cette fonction est $2n^3$. Cette complexité sera notre valeur de référence : nous verrons que l'algorithme du pivot possède une complexité comparable. Résoudre un système, calculer un déterminant, inverser une matrice, sont des opérations dont le coût est analogue à celui de la multiplication matricielle.

[29]:
```
def prodmat(A, B):
    if nb_col(A) != nb_lig(B):
        raise Exception('Matrices incompatibles')
    p = nb_lig(A)
    q = nb_col(A)
    r = nb_col(B)
    C = [r * [0] for i in range(p)]
    for i in range(p):
        for j in range(r):
```

```
            s = 0
            for k in range(q):
                s += A[i][k] * B[k][j]
            C[i][j] = s
    return C
```

[31]:
```
prnt(prodmat([[1, 3], [2, 4]], [[5, 7], [6, 8]]))
```

```
+------------+------------+
| +23.00000  | +31.00000  |
+------------+------------+
| +34.00000  | +46.00000  |
+------------+------------+
```

La fonction `submat` renvoie la différence des matrices A et B. Elle effectue pq soustractions pour soustraire deux matrices de taille $p \times q$.

[32]:
```
def submat(A, B):
    p = nb_lig(A)
    q = nb_col(A)
    if (p != nb_lig(B)) or (q != nb_col(B)):
        raise Exception('Matrices incompatibles')
    C = [q * [0] for i in range(p)]
    for i in range(p):
        for j in range(q):
            C[i][j] = A[i][j] - B[i][j]
    return C
```

[33]:
```
prnt(submat([[1, 2, 3], [4, 5, 6]], [[2, 1, 3], [5, 4, 6]]))
```

```
+------------+------------+------------+
|  -1.00000  |  +1.00000  |  +0.00000  |
+------------+------------+------------+
|  -1.00000  |  +1.00000  |  +0.00000  |
+------------+------------+------------+
```

1.2 2. Opérations sur les lignes d'une matrice

1.2.1 2.1 Introduction

Le principe de l'algorithme du pivot est d'effectuer des opérations sur les lignes d'une matrice. Ces opérations sont au nombre de trois :

1. Échanger deux lignes.
2. Multiplier une ligne par un réel non nul.
3. Ajouter à une ligne un multiple d'une autre ligne.

Pourquoi ce opérations ?

Proposition : Soient $A \in \mathcal{M}_n(\mathbb{R})$ $X \in \mathcal{M}_{n,q}(\mathbb{R})$ et $B \in \mathcal{M}_{n,q}(\mathbb{R})$. Soit A' la matrice obtenue à partir de A en effectuant l'une des opérations ci-dessus. Soit B' la matrice obtenue en effectuant sur B la même opération que celle faite sur A. On a alors

- $\mathrm{rg}A' = \mathrm{rg}A$
- $AX = B \Leftrightarrow A'X = B'$

Démonstration : Faite dans tous les bon cours d'algèbre linéaire.

On peut de plus remarquer que

1. Pour l'opération d'échange de lignes, $\det A' = \det A$.
2. Pour l'opération multipliant une ligne de A par le réel λ, $\det A' = \lambda \det A$
3. Pour l'opération d'ajout à une ligne de A d'un multiple d'une autre ligne de A, on a $\det A' = \det A$.

1.2.2 2.2 Echange des lignes i et j : $L_i \leftrightarrow L_j$

Soit $A \in \mathcal{M}_{pq}(\mathbb{R})$. Échanger les lignes i et j de la matrice A nécessite q échanges de coefficients, où q est le nombre de colonnes de A. Elle ne fait en revanche aucune opération sur des flottants.

[34]:
```
def echanger_lignes(A, i, j):
    q = nb_col(A)
    for k in range(q):
        A[i][k], A[j][k] = A[j][k], A[i][k]
```

[35]:
```
A = [[1, 2, 3], [4, 5, 6]]
prnt(A)
```

```
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
|  +4.00000  |  +5.00000  |  +6.00000  |
+------------+------------+------------+
```

[36]:
```
echanger_lignes(A, 0, 1)
prnt(A)
```

```
+------------+------------+------------+
|  +4.00000  |  +5.00000  |  +6.00000  |
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
```

Remarque : Si l'on connaît un peu la façon dont Python implémente les listes, il s'avère que cette opération s'exécute en complexité $O(1)$. Voici la fonction magique `echanger_lignes` :

[37]:
```
def echanger_lignes(A, i, j):
    A[i], A[j] = A[j], A[i]
```

1.2.3 2.3 Multiplier la ligne i par le réel t : $L_i \leftarrow tL_i$

Multiplier la ligne i de $A \in \mathcal{M}_{pq}(\mathbb{R})$ par le réel t nécessite q multiplications de flottants.

[38]:
```
def multiplier_ligne(A, i, t):
    q = nb_col(A)
    for j in range(q): A[i][j] *= t
```

[39]:
```
multiplier_ligne(A, 0, 4)
prnt(A)
```

```
+------------+------------+------------+
| +16.00000  | +20.00000  | +24.00000  |
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
```

1.2.4 2.4 Ajouter à la ligne k t fois la ligne i : $L_k \leftarrow L_k + tL_i$

Ajouter à la ligne k de $A \in \mathcal{M}_{pq}(\mathbb{R})$ t fois la ligne i nécessite $2q$ opérations sur des flottants.

[40]:
```
def combiner_lignes(A, k, i, t):
    q = nb_col(A)
    for j in range(q): A[k][j] += t * A[i][j]
```

Sur notre exemple, si nous voulons annuler A_{00}, il nous suffit d'effectuer l'opération $L_0 \leftarrow L_0 - 16L_1$.

[41]:
```
combiner_lignes(A, 0, 1, -16)
prnt(A)
```

```
+------------+------------+------------+
|  +0.00000  | -12.00000  | -24.00000  |
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
```

1.3 3. L'algorithme

On s'intéresse ici au cas où la matrice A est inversible (et donc carrée). La version que nous allons étudier est appelée **pivot total**. Elle est environ trois fois moins efficace que celle du **pivot partiel** mais elle est un peu plus simple à décrire. Nous examinerons l'algorithme du pivot partiel dans la dernière partie du notebook.

1.3.1 3.1 Introduction

Voici un petit exemple qui va permettre de comprendre l'algorithme du pivot de Gauss. Résolvons le système

$$\begin{cases} x + 2y + 3z &= 8 \\ 4x + 5y + 6z &= 17 \\ 7x + 8y + 8z &= 24 \end{cases}$$

L'unique solution de ce système est $(0, 1, 2)$. Ce système s'écrit de façon plus compacte $AX = B$, où $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{pmatrix}$ et $B = \begin{pmatrix} 8 \\ 17 \\ 24 \end{pmatrix}$.

[42]:
```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 8]]
B = [[8], [17], [24]]
prnt(A)
print()
prnt(B)
```

```
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
|  +4.00000  |  +5.00000  |  +6.00000  |
+------------+------------+------------+
|  +7.00000  |  +8.00000  |  +8.00000  |
+------------+------------+------------+

+------------+
|  +8.00000  |
+------------+
| +17.00000  |
+------------+
| +24.00000  |
+------------+
```

Effectuons les opérations $L_1 \leftarrow L_1 - 4L_0$ et $L_2 \leftarrow L_2 - 7L_0$ sur les matrices A et B. Rappelons que les indices commencent à 0 !

[43]:
```
combiner_lignes(A, 1, 0, -4)
combiner_lignes(B, 1, 0, -4)
combiner_lignes(A, 2, 0, -7)
combiner_lignes(B, 2, 0, -7)
prnt(A)
print()
prnt(B)
```

```
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
```

```
+------------+------------+------------+
|  +0.00000  |  -3.00000  |  -6.00000  |
+------------+------------+------------+
|  +0.00000  |  -6.00000  | -13.00000  |
+------------+------------+------------+

+------------+
|  +8.00000  |
+------------+
| -15.00000  |
+------------+
| -32.00000  |
+------------+
```

Nous obtenons un nouveau système $A'X = B'$ où la matrice A' possède des zéros sur la première colonne. La clé de tout est le fait suivant : A' **est inversible et les systèmes** $AX = B$ **et** $A'X = B'$ **ont les mêmes solutions.**

Maintenant, effectuons l' opération $L_1 \leftarrow -\frac{1}{3}L_1$ sur A' et B'. Ceci place un 1 ligne 1, colonne 1.

[44]:
```
multiplier_ligne(A, 1, -1/3)
multiplier_ligne(B, 1, -1/3)
prnt(A)
print()
prnt(B)
```

```
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
|  -0.00000  |  +1.00000  |  +2.00000  |
+------------+------------+------------+
|  +0.00000  |  -6.00000  | -13.00000  |
+------------+------------+------------+

+------------+
|  +8.00000  |
+------------+
|  +5.00000  |
+------------+
| -32.00000  |
+------------+
```

Combinons maintenant les lignes 0 et 2 des matrices A' et B' avec leur ligne 1 pour faire apparaître des 0 sur la colonne 1 de la matrice A'.

[45]:
```
combiner_lignes(A, 0, 1, -2)
combiner_lignes(B, 0, 1, -2)
combiner_lignes(A, 2, 1, 6)
combiner_lignes(B, 2, 1, 6)
```

```
prnt(A)
print()
prnt(B)
```

```
+------------+------------+------------+
|  +1.00000  |  +0.00000  |  -1.00000  |
+------------+------------+------------+
|  -0.00000  |  +1.00000  |  +2.00000  |
+------------+------------+------------+
|  +0.00000  |  +0.00000  |  -1.00000  |
+------------+------------+------------+

+------------+
|  -2.00000  |
+------------+
|  +5.00000  |
+------------+
|  -2.00000  |
+------------+
```

Nous obtenons un nouveau système $A''X = B''$ où A'' est inversible, et qui a toujours les mêmes solutions que le système d'origine. Un dernier effort : Opérons maintenant pour faire apparaître des zéros dans la colonne 2 de la matrice A''.

[46]:
```
multiplier_ligne(A, 2, -1)
multiplier_ligne(B, 2, -1)
combiner_lignes(A, 0, 2, 1)
combiner_lignes(B, 0, 2, 1)
combiner_lignes(A, 1, 2, -2)
combiner_lignes(B, 1, 2, -2)
prnt(A)
print()
prnt(B)
```

```
+------------+------------+------------+
|  +1.00000  |  +0.00000  |  +0.00000  |
+------------+------------+------------+
|  +0.00000  |  +1.00000  |  +0.00000  |
+------------+------------+------------+
|  -0.00000  |  -0.00000  |  +1.00000  |
+------------+------------+------------+

+------------+
|  +0.00000  |
+------------+
|  +1.00000  |
+------------+
|  +2.00000  |
```

```
+------------+
```

Nous voici avec un système $A'''X = B'''$ ayant les mêmes solutions que le système d'origine. Comme $A''' = I_3$, ce dernier système est donc extrêmement simple : il s'agit de $X = B'''$. Nous avons résolu notre système d'origine.

Ce que nous venons de faire sur un exemple peut être fait pour n'importe quel système linéaire $AX = B$ lorsque $A \in GL_n(\mathbb{R})$ est une matrice inversible. L'algorithme est le suivant :

Pour k variant de 0 à $n-1$: 1. Trouver un indice $k \leq \ell \leq n-1$ tel que $A_{\ell k} \neq 0$ 2. Si $\ell \neq k$ échanger les lignes ℓ et k des matrices A et B. Maintenant, $A_{kk} \neq 0$. 3. Diviser la ligne k de A et B par $\frac{1}{A_{kk}}$. Maintenant, $A_{kk} = 1$. 4. Pour i variant de 0 à $n-1$, $i \neq k$, combiner la ligne i de A et B avec leur ligne k de sorte que A_{ik} devienne nul. La combinaison est clairement $L_i \leftarrow L_i - A_{ik}L_k$.

Problème : **Le** point problématique est l'étape 1. Un tel indice ℓ existe-t-il ? Nous verrons que oui, grâce à l'inversibilité de A. Admettons pour l'instant l'existence de cet indice ℓ et lançons-nous dans l'écriture du code.

1.3.2 3.2 Recherche du pivot

Soit $A \in \mathcal{M}_n(\mathbb{R})$. Soit $k \in [0, n-1]$. On cherche un indice $\ell \in [|k, n-1|]$ tel que $A_{\ell k} \neq 0$. Si on le trouve, on le renvoie. Sinon, on lève une exception.

[47]:
```
def chercher_pivot(A, k):
    n = nb_lig(A)
    l = k
    while l < n and A[l][k] == 0: l = l + 1
    if l == n: raise Exception('Pivot non trouvé')
    return l
```

[48]:
```
A = [[0, 2, 3, 4], [1, 5, 1, 1], [4, 2, 2, 0]]
prnt(A)
```

```
+------------+------------+------------+------------+
|  +0.00000  |  +2.00000  |  +3.00000  |  +4.00000  |
+------------+------------+------------+------------+
|  +1.00000  |  +5.00000  |  +1.00000  |  +1.00000  |
+------------+------------+------------+------------+
|  +4.00000  |  +2.00000  |  +2.00000  |  +0.00000  |
+------------+------------+------------+------------+
```

[49]:
```
chercher_pivot(A, 0)
```

[49]: 1

Revenons à notre question existentielle : un tel indice ℓ existe-t-il ? Nous allons montrer que si la matrice A possède une certaine forme la réponse est oui. Plus loin, lors de la preuve de la correction de l'algorithme du pivot de Gauss, il nous suffira de montrer que A possède effectivement la dite forme.

Proposition : Soit $A \in GL_n(\mathbb{R})$. Soit $0 \leq k \leq n-1$. On suppose que A s'écrit par blocs

$$A = \begin{pmatrix} I_k & U \\ 0 & V \end{pmatrix}$$

où $U \in \mathcal{M}_{k(n-k)}(\mathbb{R})$, $V \in \mathcal{M}_{(n-k)(n-k)}(\mathbb{R})$ et $0 \in \mathcal{M}_{(n-k)k}(\mathbb{R})$. Alors il existe $\ell \in [0, n-1-k]$ tel que $V_{\ell k} \neq 0$.

Démonstration : Si l'on suppose que la colonne 0 de V est nulle, alors la colonne k de A est combinaison linéaire des colonnes $0, 1, \ldots, k-1$. Ce n'est pas possible puisque A est inversible.

Remarque : Au lecteur dérouté par le cas $k = 0$, on convient qu'une matrice ayant 0 ligne ou 0 colonne est vide. On suppose donc dans ce cas que $A = V$. La colonne 0 de A n'est pas la colonne nulle puisque A est inversible.

1.3.3 3.3 Pivoter

La fonction `pivoter` ci-dessous effectue les opérations suivantes :

- On cherche un coefficient non nul sur la $k$ème colonne de A, à partir de la ligne k. Mettons que ce coefficient soit trouvé à la ligne ℓ. Un tel coefficient existe si la matrice est inversible.
- On échange les lignes k et ℓ de A.
- On divise la ligne k de A par A_{kk} qui est maintenant non nul.
- Pour $i \neq k$, on soustrait à la $i$ème ligne de A A_{ik} fois la $k$ème ligne.
- On effectue les mêmes opérations sur le second membre B

Le résultat de cette fonction est d'annuler tous les coefficients de la colonne k de A, sauf celui situé à la ligne k qui devient égal à 1.

La fonction `pivoter` prend également un paramètre D qui est un flottant dont nous reparlerons plus loin.

[50]:
```
def pivoter(A, B, k, D):
    n = nb_lig(A)
    l = chercher_pivot(A, k)
    if l != k:
        D = -D
        echanger_lignes(A, k, l)
        echanger_lignes(B, k, l)
    P = A[k][k]
    D = D * P
    multiplier_ligne(B, k, 1 / P)
    multiplier_ligne(A, k, 1 / P)
    for i in range(n):
        if i != k:
            Aik = A[i][k]
            combiner_lignes(A, i, k, -Aik)
```

```
            combiner_lignes(B, i, k, -Aik)
    return D
```

Pour les amateurs de théorie, voici la

Proposition : Soit $A \in GL_n(\mathbb{R})$. Soit $0 \leq k \leq n-1$. On suppose que A s'écrit par blocs

$$A = \begin{pmatrix} I_k & U \\ 0 & V \end{pmatrix}$$

où $U \in \mathcal{M}_{k(n-k)}(\mathbb{R})$, $V \in \mathcal{M}_{(n-k)(n-k)}(\mathbb{R})$ et $0 \in \mathcal{M}_{(n-k)k}(\mathbb{R})$. Soient A' et B' les valeurs des matrices A et B après un appel à `pivoter(A, B, k, D)`. - On a $A' = \begin{pmatrix} I_{k+1} & U' \\ 0 & V' \end{pmatrix}$ où $U' \in \mathcal{M}_{(k+1)(n-k-1)}(\mathbb{R})$, $V' \in \mathcal{M}_{(n-k-1)(n-k-1)}(\mathbb{R})$ et $0 \in \mathcal{M}_{(n-k-1)(k+1)}(\mathbb{R})$. - La matrice A' est inversible. - On a $AX = B$ si et seulement si $A'X = B'$.

Reprenons nos matrices d'exemple.

[51]:
```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 8]]
B = [[8], [17], [24]]
prnt(A)
print()
prnt(B)
```

```
+------------+------------+------------+
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
|  +4.00000  |  +5.00000  |  +6.00000  |
+------------+------------+------------+
|  +7.00000  |  +8.00000  |  +8.00000  |
+------------+------------+------------+

+------------+
|  +8.00000  |
+------------+
| +17.00000  |
+------------+
| +24.00000  |
+------------+
```

[52]:
```
D = 1
D = pivoter(A, B, 0, D)
print('D=', D)
prnt(A)
print()
prnt(B)
```

```
D= 1
+------------+------------+------------+
```

```
|  +1.00000  |  +2.00000  |  +3.00000  |
+------------+------------+------------+
|  +0.00000  |  -3.00000  |  -6.00000  |
+------------+------------+------------+
|  +0.00000  |  -6.00000  | -13.00000  |
+------------+------------+------------+

+------------+
|  +8.00000  |
+------------+
| -15.00000  |
+------------+
| -32.00000  |
+------------+
```

[53]:
```
D = pivoter(A, B, 1, D)
print('D=', D)
prnt(A)
print()
prnt(B)
```

```
D= -3.0
+------------+------------+------------+
|  +1.00000  |  +0.00000  |  -1.00000  |
+------------+------------+------------+
|  -0.00000  |  +1.00000  |  +2.00000  |
+------------+------------+------------+
|  +0.00000  |  +0.00000  |  -1.00000  |
+------------+------------+------------+

+------------+
|  -2.00000  |
+------------+
|  +5.00000  |
+------------+
|  -2.00000  |
+------------+
```

[54]:
```
D = pivoter(A, B, 2, D)
print('D=', D)
prnt(A)
print()
prnt(B)
```

```
D= 3.0
+------------+------------+------------+
|  +1.00000  |  +0.00000  |  +0.00000  |
+------------+------------+------------+
```

```
|  +0.00000  |  +1.00000  |  +0.00000  |
+------------+------------+------------+
|  -0.00000  |  -0.00000  |  +1.00000  |
+------------+------------+------------+

+------------+
|  +0.00000  |
+------------+
|  +1.00000  |
+------------+
|  +2.00000  |
+------------+
```

Nous avons automatisé ce que nous avions fait manuellement au paragraphe 3.1. La dernière automatisation consiste en une simple boucle.

1.3.4 3.4 Une simple boucle

La fonction `pivot_gauss` est maintenant immédiate. Elle prend en paramètres deux matrices A et B. On suppose que - A est inversible. - Le nombre de lignes de B est le même que celui de A.

Soit n le nombre de lignes de A. Pour k variant de 0 à $n-1$, on applique la fonction `pivoter` à la matrice A et à la matrice B.

Proposition : Soient $A \in GL_n(\mathbb{R})$ et $B \in \mathcal{M}_{nq}(\mathbb{R})$. Un appel à `pivot_gauss(A, B)` renvoie un triplet (A', X, D) où - $A' = I_n$. - $X \in \mathcal{M}_{nq}(\mathbb{R})$ est l'unique solution de l'équation $AX = B$. - $D = \det A$ est le déterminant de la matrice A.

Démonstration : Tout a déjà été (presque) fait. Le lecteur consciencieux reprendra les propositions précédentes pour s'en convaincre.

[55]:
```
def pivot_gauss(A, B):
    n = nb_lig(A)
    A = [A[i].copy() for i in range(n)]
    B = [B[i].copy() for i in range(n)]
    D = 1
    for i in range(n):
        D = pivoter(A, B, i, D)
    return (A, B, D)
```

Reprenons une dernière fois notre petit exemple

[56]:
```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 8]]
B = [[8], [17], [24]]
A1, B1, D = pivot_gauss(A, B)
print("D =", D)
prnt(A1)
print()
prnt(B1)
```

```
D = 3.0
+------------+------------+------------+
|  +1.00000  |  +0.00000  |  +0.00000  |
+------------+------------+------------+
|  +0.00000  |  +1.00000  |  +0.00000  |
+------------+------------+------------+
|  -0.00000  |  -0.00000  |  +1.00000  |
+------------+------------+------------+

+------------+
|  +0.00000  |
+------------+
|  +1.00000  |
+------------+
|  +2.00000  |
+------------+
```

1.3.5 3.5 Complexité

Combien d'opérations sur des flottants la fonction `pivot_gauss` effectue-t-elle lorsqu'elle est appelée sur une matrice $A \in GL_n(\mathbb{R})$ et une matrice $B \in \mathcal{M}_{n,q}(\mathbb{R})$?

La fonction `pivoter` est appelée n fois. À chaque appel de celle-ci :

- On cherche la valeur du pivot par un appel à `chercher_pivot` : cette fonction ne fait aucune opération sur des flottants.
- On échange éventuellement deux lignes de A et B : toujours aucune opération.
- On multiplie une ligne de A et une ligne de B par un réel : $n + q$ opérations.
- On appelle $n - 1$ fois la fonction `combiner_lignes` sur A et B : $2(n + q)$ opérations.

Notons $C_{n,q}$ le nombre d'opérations total. On a donc

$$C_{n,q} = n(n + q + 2(n - 1)(n + q) = n(n + q)(2n - 1)$$

Deux cas nous intéresseront dans la suite :

- On résout un système $AX = B$ où $B \in \mathcal{M}_{n1}(\mathbb{R})$. Dans ce cas $q = 1$ et

$$C_{n1} = n(n + 1)(2n - 1) = 2n^3 + n^2 - n \sim 2n^3$$

- On inverse une matrice A. Nous verrons que dans ce cas $B = I_n \in \mathcal{M}_n(\mathbb{R})$. Ainsi $q = n$ et

$$C_{nn} = n(2n)(2n - 1) = 4n^3 - 2n^2 \sim 4n^3$$

Remarque : Le nombre d'opérations nécessaire pour multiplier deux matrices $n \times n$ avec notre fonction `prodmat` est $2n^3$. Ainsi, résoudre un système de Cramer coûte exactement la même chose (en termes d'opérations sur des flottants) que multiplier deux matrices. Et contrairement aux idées reçues, inverser une matrice ne coûte que deux fois plus cher !

Tentons une expérience numérique. Résolvons un système de 200 équations à 200 inconnues. Quel est le temps nécessaire à sa résolution ? Comment ce temps ce compare-t-il au temps mis pour élever une matrice de taille 200 au carré ?

[57]:
```
n = 200
A = randmat(n, n)
B = randmat(n, 1)
t1 = time.time()
pivot_gauss(A, B)
t2 = time.time()
print('Temps pivot  : %.1fs' % (t2 - t1))
t3 = time.time()
prodmat(A, A)
t4 = time.time()
print('Temps produit: %.1fs' % (t4 - t3))
```

```
Temps pivot  : 2.4s
Temps produit: 2.3s
```

Les deux temps sont comparables, et nous sommes même plutôt agréablement surpris. Rappelons que notre complexité théorique ne compte que les opérations sur des flottants. Il s'avère que des instructions du type `A[i][j] = ...` n'ont pas un coût négligeable en Python. Nous ne creuserons pas plus avant pour ne pas rendre ce notebook incommensurablement long.

Disons que notre complexité « pratique » est $C_n \sim Cn^3$, exprimée en secondes. Que vaut C ? Cela dépend de la machine utilisée, bien évidemment.

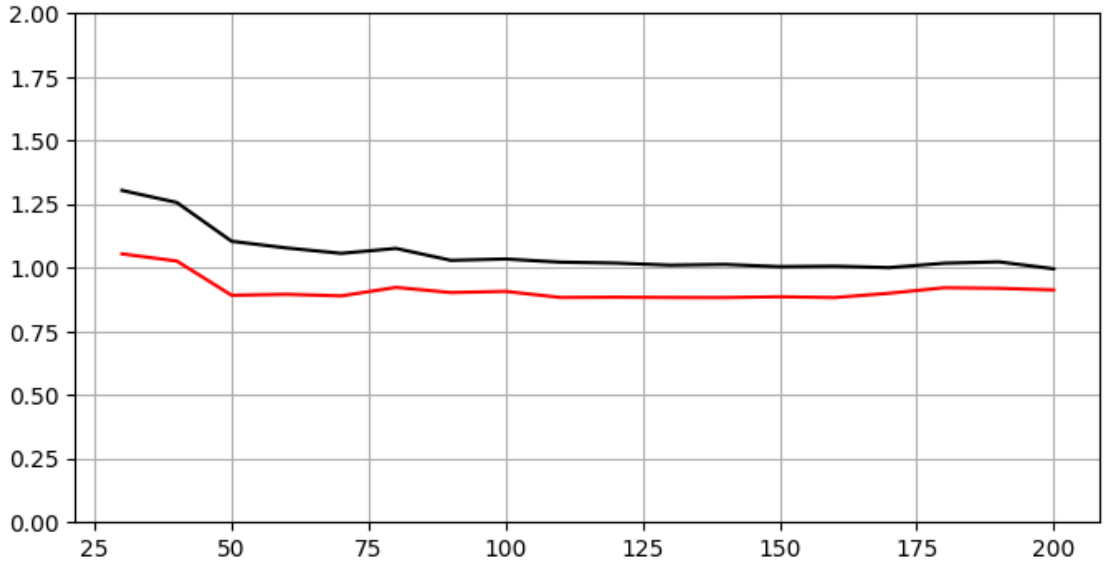
[58]:
```
n = 200
A = randmat(n, n)
B = randmat(n, 1)
t1 = time.time()
pivot_gauss(A, B)
t2 = time.time()
C = (t2 - t1) / n ** 3
print('C=%.2e' % C)
```

```
C=2.97e-07
```

Traçons la courbe de la fonction $n \longmapsto \frac{t}{Cn^3}$ où t est le temps mis pour résoudre un système $n \times n$. Traçons également la courbe de la fonction $n \longmapsto \frac{t'}{Cn^3}$ où t' est le temps mis pour multiplier deux matrices $n \times n$.

[61]:
```
s1 = []
s2 = []
rng = range(30, 201, 10)
for n in rng:
    A = randmat(n, n)
    B = randmat(n, 1)
    t1 = time.time()
    pivot_gauss(A, B)
```

```
    t2 = time.time()
    s1.append((t2 - t1) / (C * n ** 3))
    t3 = time.time()
    prodmat(A, A)
    t4 = time.time()
    s2.append((t4 - t3) / (C * n ** 3))
plt.ylim(0, 2)
plt.plot(rng, s1, 'k')
plt.plot(rng, s2, 'r')
plt.grid()
```



Nous obtenons ce que nous pensions. La courbe noire, celle qui correspond à l'algorithme du pivot, est quasi-constante, égale à 1. La courbe rouge (produit matriciel) est elle aussi-quasi constante, avec une valeur plus faible que 1, environ 0.8.

1.3.6 3.6 Quelques justifications sur la valeur de D

Soit D' la valeur de D après l'appel à `pivoter`. Soit A' la valeur de la matrice A après ce même appel. - Un éventuel échange de lignes change le signe de $\det A$. - La multiplication d'une ligne de A par $\frac{1}{P}$ divise le déterminant de A par P. - L'ajout à une ligne de A d'un multiple d'une autre ligne de A ne change pas son déterminant.

Ainsi, en posant $K = \pm P$,

$$\det A' = \frac{1}{K} \det A, \quad D' = KD$$

d'où

$$D' \det A' = D \det A$$

Ainsi, la quantité $D \det A$ est un invariant de boucle. Après n itérations de `pivoter`, la matrice A devient la matrice identité. Soit D la valeur finale. On a donc $D \det I_n = 1 \det A$, ou encore

$$D = \det A$$

1.4 4. Inversion d'une matrice

L'algorithme du pivot permet d'inverser une matrice $A \in \mathcal{M}_n(\mathbb{R})$: il suffit de mettre pour second membre du système la matrice I_n.

[62]:
```
def inverse(A, dbg=False):
    p = nb_lig(A)
    q = nb_col(A)
    if p != q:
        raise Exception('inverser: matrice pas carrée')
    B = eye(p)
    A1, B1, D = pivot_gauss(A, B)
    if dbg: print('Déterminant: %.5e' % D)
    return B1
```

Inversons une matrice aléatoire 4×4.

[63]:
```
A = randmat(4, 4)
prnt(A)
print()
B = inverse(A)
prnt(B)
```

```
+------------+------------+------------+------------+
|  -0.18828  |  +0.63477  |  +0.43689  |  -0.94777  |
+------------+------------+------------+------------+
|  +0.94170  |  -0.39208  |  -0.77402  |  -0.21631  |
+------------+------------+------------+------------+
|  -0.23130  |  +0.38631  |  -0.20643  |  +0.37933  |
+------------+------------+------------+------------+
|  +0.95840  |  +0.10295  |  +0.99896  |  -0.79604  |
+------------+------------+------------+------------+

+------------+------------+------------+------------+
|  -0.35479  |  +0.48321  |  +0.88361  |  +0.71217  |
+------------+------------+------------+------------+
|  +0.43510  |  +0.19589  |  +1.97401  |  +0.36941  |
+------------+------------+------------+------------+
|  -0.40600  |  -0.72033  |  -0.21733  |  +0.57557  |
```

```
+------------+------------+------------+------------+
|  -0.88038  |  -0.29685  |  +1.04639  |  +0.37126  |
+------------+------------+------------+------------+
```

Faisons le produit AB, cela devrait donner l'identité. En fin presque : je rappelle que nous faisons des calcul en flottants, et donc des calculs approchés.

[64]:
```
prnt(prodmat(A, B))
```

```
+------------+------------+------------+------------+
|  +1.00000  |  -0.00000  |  -0.00000  |  +0.00000  |
+------------+------------+------------+------------+
|  +0.00000  |  +1.00000  |  +0.00000  |  +0.00000  |
+------------+------------+------------+------------+
|  -0.00000  |  -0.00000  |  +1.00000  |  +0.00000  |
+------------+------------+------------+------------+
|  +0.00000  |  -0.00000  |  +0.00000  |  +1.00000  |
+------------+------------+------------+------------+
```

Faisons la même expérience qu'avec le pivot de Gauss : Comparons le temps mis pour inverser une matrice avec celui mis pour l'élever au carré.

[65]:
```
n = 200
A = randmat(n, n)
t1 = time.time()
B = inverse(A)
t2 = time.time()
print('Temps inversion   : %.1fs' % (t2 - t1))
t3 = time.time()
B = prodmat(A, A)
t4 = time.time()
print('Temps produit     : %.1fs' % (t4 - t3))
print('Quotient des temps: %.1f' % ((t2 - t1) / (t4 - t3)))
```

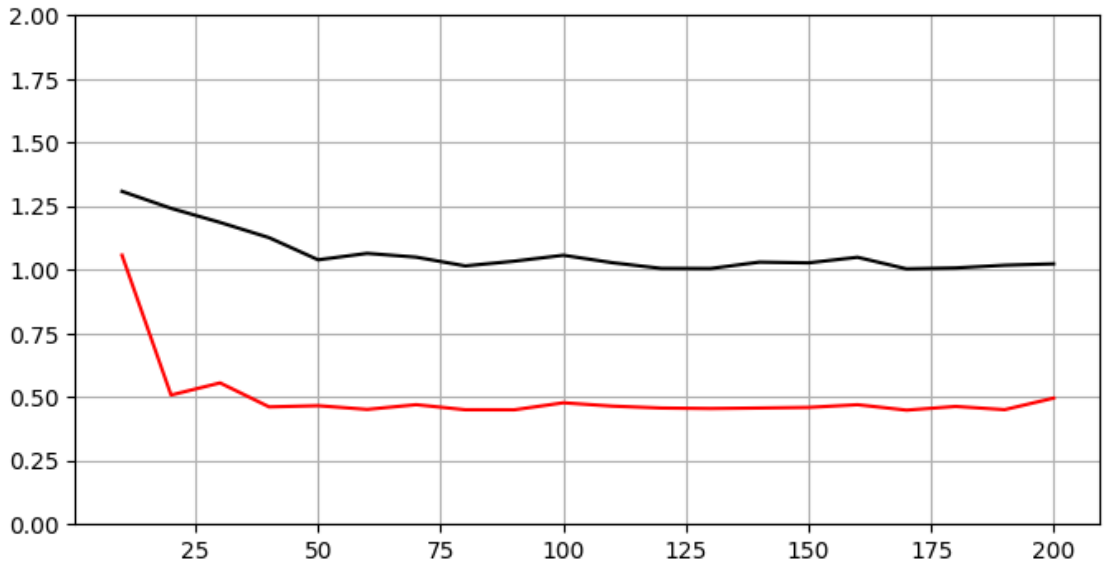
```
Temps inversion   : 4.7s
Temps produit     : 2.2s
Quotient des temps: 2.1
```

Notre théorie prévoyait un temps d'inversion double du temps de multiplication. La pratique nous conforte dans l'idée que contrairement à ce que disent les mauvaises langues, la théorie, ça marche :-).

[66]:
```
n = 200
A = randmat(n, n)
t1 = time.time()
B = inverse(A)
t2 = time.time()
C = (t2 - t1) / n ** 3
print('C = %.2e' % C)
```

```
C = 5.83e-07
```

[67]:
```
s1 = []
s2 = []
rng = range(10, 201, 10)
for n in rng:
    A = randmat(n, n)
    B = randmat(n, 1)
    t1 = time.time()
    inverse(A)
    t2 = time.time()
    s1.append((t2 - t1) / (C * n ** 3))
    t3 = time.time()
    prodmat(A, A)
    t4 = time.time()
    s2.append((t4 - t3) / (C * n ** 3))
plt.ylim(0, 2)
plt.plot(rng, s1, 'k')
plt.plot(rng, s2, 'r')
plt.grid()
```



1.5 5. Stabilité de l'algorithme du pivot

Cette section mériterait un long développement … que nous ne ferons pas.

1.5.1 5.1 Norme de Frobenius

Nous avons eu, lors de la vérification de nos algorithmes, à nous poser des questions du genre : A-t-on $AX = B$? Le souci est que, opérant sur des flottants, la vraie question serait plutôt : a-t-on $AX \simeq B$? Ou encore : a-t-on $AX - B$ "petit" ? La notion de petitesse est très relative. Introduisons, pour toute matrice A à coefficients réels, la quantité

$$\|A\| = \left(\sum_{i,j} A_{ij}^2 \right)^{1/2}$$

Le nombre $\|A\|$ est la **norme de Frobenius** de la matrice A. On montre aisément que la fonction $A \mapsto \|A\|$ est une norme sur l'espace vectoriel $\mathcal{M}_{pq}(\mathbb{R})$. Une "petite" matrice sera donc, dans la discussion qui va suivre, une matrice de "petite norme".

[68]:
```
def norme_frob(A):
    s = 0
    p = nb_lig(A)
    q = nb_col(A)
    for i in range(p):
        for j in range(q):
            s += A[i][j] ** 2
    return math.sqrt(s)
```

[69]:
```
print(norme_frob([[1, 2], [3, 4]]))
```

```
5.477225575051661
```

La fonction `check_prod` prend en paramètres trois matrices A, X, B. Elle renvoie $\|AX - B\|$. Si X est solution du système $AX = B$, le valeur renvoyée par la fonction `check_prod` est donc 0. Dans la pratique, à cause des erreurs d'arrondis, la valeur devrait être petite quoique non nulle.

[70]:
```
def check_prod(A, X, B):
    return norme_frob(submat(prodmat(A, X), B))
```

[71]:
```
A = randmat(100, 100)
B = randmat(100, 1)
A1, X, D = pivot_gauss(A, B)
print(check_prod(A, X, B))
```

```
3.867914080153951e-12
```

Il semble bien que $AX \simeq B$. Tentons autre chose.

1.5.2 5.2 Les matrices de Hilbert

Les matrices de Hilbert sont connues pour poser de gros problèmes aux algorithmes numériques. Pour tout $n \geq 1$, posons

$$H_n = \begin{pmatrix} \frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{n+2} \\ \vdots & \vdots & \vdots & & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n-1} \end{pmatrix}$$

On peut montrer que H_n est inversible, quoique son déterminant soit très petit, vraiment TRÈS petit.

[72]:
```
def hilbert(n):
    return [[1 / (i + j + 1) for i in range(n)] for j in range(n)]
```

[73]:
```
prnt(hilbert(8))
```

```
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +1.00000  |  +0.50000  |  +0.33333  |  +0.25000  |  +0.20000  |  +0.16667  |
+0.14286  |  +0.12500  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +0.50000  |  +0.33333  |  +0.25000  |  +0.20000  |  +0.16667  |  +0.14286  |
+0.12500  |  +0.11111  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +0.33333  |  +0.25000  |  +0.20000  |  +0.16667  |  +0.14286  |  +0.12500  |
+0.11111  |  +0.10000  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +0.25000  |  +0.20000  |  +0.16667  |  +0.14286  |  +0.12500  |  +0.11111  |
+0.10000  |  +0.09091  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +0.20000  |  +0.16667  |  +0.14286  |  +0.12500  |  +0.11111  |  +0.10000  |
+0.09091  |  +0.08333  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +0.16667  |  +0.14286  |  +0.12500  |  +0.11111  |  +0.10000  |  +0.09091  |
+0.08333  |  +0.07692  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +0.14286  |  +0.12500  |  +0.11111  |  +0.10000  |  +0.09091  |  +0.08333  |
+0.07692  |  +0.07143  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
----------+-----------+
|  +0.12500  |  +0.11111  |  +0.10000  |  +0.09091  |  +0.08333  |  +0.07692  |
+0.07143  |  +0.06667  |
+-----------+-----------+-----------+-----------+-----------+-----------+-
```

```
----------+-----------+
```

Prenons une matrice de Hilbert A de taille 20, calculons son inverse B avec notre fonction `inverse`, puis vérifions si $AB = I_{20}$.

[74]:
```
n = 20
A = hilbert(n)
B = inverse(A, dbg=True)
print(check_prod(A, B, eye(n)))
```

```
Déterminant: 1.82581e-196
1466265.670936713
```

Le résultat est catastrophique. Nous ne creuserons pas plus avant ce sujet, il nécessiterait un notebook à lui tout seul. Mais il est bon de garder à l'esprit que sur certaines matrices l'algorithme du pivot peut tout à fait donner des résultats **désastreux**.

1.6 6. Le pivot partiel

Nous allons dans cette section améliorer la complexité de l'algorithme du pivot en étudiant la métode du **pivot partiel**. Nous verrons que par cette méthode les temps de calcul sont divisés par 3.

1.6.1 6.1 Résolution d'un système triangulaire

Soit $A \in \mathcal{M}_n(\mathbb{R})$ une matrice triangulaire supérieure. Supposons que les coefficients diagonaux de A sont non nuls. La matrice A est alors inversible. Soit $B \in \mathcal{M}_{n1}(\mathbb{R})$. Il est alors aisé de résoudre le système $AX = B$, d'inconnue $X \in \mathcal{M}_{n1}(\mathbb{R})$. En effet, $AX = B$ si et seulement si

$$A_{(n-1)(n-1)}x_{n-1} = b_{n-1}$$

et pour tout $i \in [0, n-2]$,

$$A_{ii}x_i + \sum_{j=i+1}^{n-1} A_{ij}x_j = b_i$$

ou encore

$$x_i = \frac{1}{A_{ii}}\left(b_i - \sum_{j=i+1}^{n-1} A_{ij}x_j\right)$$

On obtient donc immédiatement x_{n-1}. Puis, supposant calculés $x_{n-1}, \dots, x_{i+1}$, on en déduit x_i en fonction de ceux-ci.

La fonction `solve_triang` prend en paramètres une matrice triangulaire inversible A et une matrice colonne B. Elle modifie B sur place. En sortie de fonction, B contient l'unique solution du système $AX = B$.

[75]:
```
def solve_triang(A, B):
    n = nb_lig(A)
    for i in range(n - 1, -1, -1):
        s = B[i][0]
        for j in range(i + 1, n):
            s -= A[i][j] * B[j][0]
        B[i][0] = s / A[i][i]
```

Quelle est la complexité C_n de `solve_triang` en nombre d'opérations sur des flottants ? Pour chaque i compris entre 0 et $n-1$, la fonction effectue $2(n-i-1)+1$ opérations. Ainsi,

$$C_n = \sum_{i=0}^{n-1} (2(n-i-1)+1) = \sum_{k=0}^{n-1} (2k+1) = n^2$$

Résoudre un système triangulaire a donc un coût négligeable par rapport à celui de la résolution d'un système quelconque.

1.6.2 6.2 L'idée du pivot partiel

Le faible coût de la résolution d'un système triangulaire suggère de modifier le pivot de Gauss de la façon suivante : au lieu d'opérer sur les lignes de la matrice A pour obtenir la matrice identité, - On amène A à une forme triangulaire supérieure. - On résout le système obtenu par l'algorithme du paragraphe précédent. Nous allons voir que le gain de temps est conséquent.

1.6.3 6.3 Nouvelles opérations sur les lignes

Les nouvelles fonctions sur les lignes se passent de commentaires. - La fonction `echanger_lignes` est gardée telle quelle, son coût est négligeable par rapport au coût total. - La fonction `multiplier_ligne`, appelée sur la ligne i d'une matrice, ne modifie les colonnes de la matrice qu'à partir de la colonne i. L'idée est que les colonnes précédentes contiendront des zéros lorsque cette fonction sera appelée. - Même remarque pour la fonction `combiner_lignes`.

[76]:
```
def multiplier_ligne_partiel(A, i, t):
    q = nb_col(A)
    for j in range(i, q): A[i][j] *= t
```

[77]:
```
def combiner_lignes_partiel(A, k, i, t):
    q = nb_col(A)
    for j in range(i, q): A[k][j] += t * A[i][j]
```

1.6.4 6.4 La fonction de pivot

La nouvelle fonction de pivot est un quasi copier-coller de l'ancienne. Les seules différences sont : - Des appels à des opérations partielles sur les lignes de la matrice A. - Et surtout une modification

des lignes de A uniquement à partir de la ligne $k+1$: les lignes 0 à $k-1$ sont laissées inchangées, c'est ce point précis qui amène A à une forme triangualire, et pas diagonale.

[78]:
```
def pivoter_partiel(A, B, k, D):
    n = nb_lig(A)
    l = chercher_pivot(A, k)
    if l != k:
        D = -D
        echanger_lignes(A, k, l)
        echanger_lignes(B, k, l)
    P = A[k][k]
    D = D * P
    multiplier_ligne(B, k, 1 / P)
    multiplier_ligne_partiel(A, k, 1 / P)
    for i in range(k + 1, n):
        Aik = A[i][k]
        combiner_lignes(B, i, k, -Aik)
        combiner_lignes_partiel(A, i, k, -Aik)
    return D
```

Notre nouvelle fonction `pivot_gauss` appelle `pivoter_partiel` au lieu de `pivoter` et n'oublie pas de résoudre le système triangulaire obtenu.

[79]:
```
def pivot_gauss_partiel(A, B):
    n = nb_lig(A)
    A = [A[i].copy() for i in range(n)]
    B = [B[i].copy() for i in range(n)]
    D = 1
    for k in range(n):
        D = pivoter_partiel(A, B, k, D)
    solve_triang(A, B)
    return (A, B, D)
```

Testons.

[80]:
```
A = randmat(100, 100)
B = randmat(100, 1)
A1, B1, D = pivot_gauss_partiel(A, B)
print(check_prod(A, B1, B))
```

```
4.044729556805806e-12
```

Tout a l'air de fonctionner.

1.6.5 6.5 Complexité

Quelle est la complexité C_n de `pivot_gauss_partiel` en termes d'opérations sur des flottants ? Supposons la matrice A carrée de taille n, le second membre B possédant n lignes et 1 colonne.

Pour $k = 0, \dots, n-1$, on appelle `pivoter_partiel(A, B, k, D)`. Cette fonction effectue : - 1 multiplication sur un coefficient de B - $n-k$ multiplications sur la ligne k de A. - Pour $i = k+1, \dots, n-1$, $2(n-i)$ opérations sur la ligne i de A et 2 opérations sur le $i$ième coefficient de B.

Il faut ajouter à cela la résolution du système triangulaire, qui nécessite n^2 opérations. Ainsi,

$$\begin{array}{rcl} C_n & = & \sum_{i=0}^{n-1} \left(1 + (n-i) + \sum_{k=i+1}^{n-1} (2(n-i)+2)\right) + n^2 \\ & = & \sum_{i=0}^{n-1} (n-i+1+(n-1-i)(2(n-i)+2)) + n^2 \end{array}$$

Posons $k = n-i-1$. Il vient

$$\begin{array}{rcl} C_n & = & \sum_{k=0}^{n-1} (k+2+k(2k+4)) + n^2 \\ & = & \sum_{k=0}^{n-1} (2k^2+5k+2)) + n^2 \\ & = & \frac{1}{3}n(n-1)(2n-1) + \frac{5}{2}n(n-1) + n^2 + 2n \\ & = & \frac{1}{6}n(n-1)(4n+13) + n^2 + 2n \\ & = & \frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{1}{6}n \\ & \sim & \frac{2}{3}n^3 \end{array}$$

Rappelons-nous que par la méthode du pivot total, la complexité était $2n^3$. Nous devrions donc obtenir un gain d'un facteur 3 avec l'algorithme du pivot partiel. Quelques vérifications s'imposent …

1.6.6 6.5 Quelques tests

Commençons par vérifier que nos fonctions fonctionnent. Résolvons un système de 6 équtaions à 6 inconnues.

[81]:
```
A = randmat(6, 6)
B = randmat(6, 1)
prnt(A)
print()
A1, B1, D = pivot_gauss_partiel(A, B)
prnt(A1)
```

```
+-----------+-----------+-----------+-----------+-----------+-----------+
|  -0.88431 |  +0.14258 |  +0.06001 |  -0.97422 |  -0.20884 |  +0.07581 |
+-----------+-----------+-----------+-----------+-----------+-----------+
|  -0.40361 |  +0.42173 |  +0.97150 |  -0.95554 |  +0.77423 |  -0.29805 |
+-----------+-----------+-----------+-----------+-----------+-----------+
|  +0.88867 |  +0.81939 |  +0.98036 |  +0.81476 |  +0.16997 |  -0.38153 |
+-----------+-----------+-----------+-----------+-----------+-----------+
|  +0.50000 |  -0.89329 |  +0.02803 |  -0.63405 |  -0.92089 |  -0.85544 |
+-----------+-----------+-----------+-----------+-----------+-----------+
|  -0.22434 |  -0.87722 |  -0.32921 |  +0.66588 |  -0.03604 |  -0.67577 |
+-----------+-----------+-----------+-----------+-----------+-----------+
|  +0.44274 |  +0.92691 |  +0.22233 |  -0.88090 |  -0.89110 |  +0.82560 |
+-----------+-----------+-----------+-----------+-----------+-----------+
```

```
+------------+------------+------------+------------+------------+------------+
|  +1.00000  |  -0.16123  |  -0.06786  |  +1.10168  |  +0.23616  |  -0.08573  |
+------------+------------+------------+------------+------------+------------+
|  +0.00000  |  +1.00000  |  +2.64710  |  -1.43245  |  +2.43804  |  -0.93267  |
+------------+------------+------------+------------+------------+------------+
|  +0.00000  |  +0.00000  |  +1.00000  |  -0.80571  |  +1.58324  |  -0.39301  |
+------------+------------+------------+------------+------------+------------+
|  +0.00000  |  +0.00000  |  +0.00000  |  +1.00000  |  +4.52743  |  +1.23846  |
+------------+------------+------------+------------+------------+------------+
|  +0.00000  |  +0.00000  |  +0.00000  |  +0.00000  |  +1.00000  |  +0.33929  |
+------------+------------+------------+------------+------------+------------+
|  +0.00000  |  +0.00000  |  -0.00000  |  +0.00000  |  +0.00000  |  +1.00000  |
+------------+------------+------------+------------+------------+------------+
```

La matrice A_1 possède bien une forme triangulaire, ce qui est rassurant. Maintenant, la matrice B_1 devrait contenir la solution (unique) du système $AX = B$. Calculons donc $AB_1 - B$.

[82]:
```
A1, X, D = pivot_gauss_partiel(A, B)
print(check_prod(A, X, B))
```

```
1.3089336417583914e-15
```

Tout a l'air de fonctionner. Passons maintenant à quelques tests sur la complexité de l'algorithme, comparée à celle du pivot total. Résolvons un système de 300 équations et comparons les temps pour le pivot partiel et le pivot total.

[83]:
```
n = 300
A = randmat(n, n)
B = randmat(n, 1)
t1 = time.time()
A1, B1, D = pivot_gauss_partiel(A, B)
t2 = time.time()
print('Pivot partiel: %.2fs' % (t2 - t1))
t3 = time.time()
A1, B1, D = pivot_gauss(A, B)
t4 = time.time()
print('Pivot total: %.2fs' % (t4 - t3))
print('Quotient: %.2f' % ((t4 - t3) / (t2 - t1)))
```

```
Pivot partiel: 2.73s
Pivot total: 8.08s
Quotient: 2.96
```

Des calculs un peu plus poussés montrent que le quotient des complexités théoriques est

$$3 - \frac{33}{4n} + \frac{149}{16n^2} o\left(\frac{1}{n^2}\right)$$

Il suffit pour cela de reprendre leur valeur et de faire un développement asymptotique. Qu'obtenons-nous pour $n = 200$?

[84]:
```
def quot(n):
    return 3 - 33 / (4 * n) + 149 / (16 * n ** 2)
```

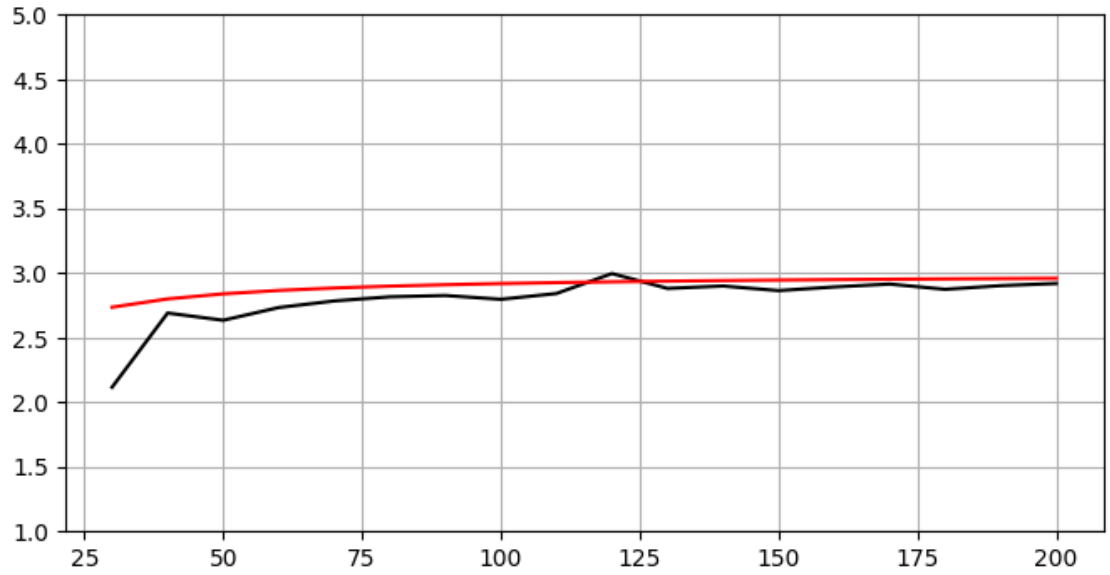
[85]:
```
print(quot(300))
```

```
2.9726034722222225
```

La pratique et la théorie s'accordent vraiment bien. Cela justifie *a posteriori* le choix de compter les opérations sur des flottants pour évaluer la complexité.

Allons, un dernier test ! Résolvons des systèmes de n équations à n inconnues pour $n = 30, 40, 50, \dots, 200$. Traçons le quotient des temps pivot total / pivot partiel en fonction de n. Traçons également en rouge notre estimation théorique du quotient.

[86]:
```
s1 = []
s2 = []
rng = range(30, 201, 10)
for n in rng:
    A = randmat(n, n)
    B = randmat(n, 1)
    t1 = time.time()
    A1, B1, D = pivot_gauss_partiel(A, B)
    t2 = time.time()
    t3 = time.time()
    A1, B1, D = pivot_gauss(A, B)
    t4 = time.time()
    s1.append((t4 - t3) / (t2 - t1))
    s2.append(quot(n))
plt.ylim(1, 5)
plt.plot(rng, s1, 'k')
plt.plot(rng, s2, 'r')
plt.grid()
```



Notre estimation théorique est tout à fait satisfaisante … Si jamais il reste un lecteur pour lire ces lignes je lui dis bravo :-) …