S54MA2M7 : Informatique 2
Le pivot de Gauss et al.

Luigi Santocanale
LIS, Aix-Marseille Université

5 mars 2019

Plan

Moralité

La carte du jour :

- deux algorithmes d'algèbre linéaire :
  pivot de Gauss, Gram-Schmidt

Aussi, réflexions autour

- du calcul numérique
- de l'informatique dans l'apprentissage/enseignements des maths

Beaucoup de matériel est tiré de :

- le(s) cours de Matthieu Moy :

  `https://matthieu-moy.fr/cours/formation-python/`

Autre source :

- `https://www.math.ust.hk/~mamu/courses/231/Slides/CH06_2A.pdf`

Plan

Solutions des systèmes d'équations

- Système d'equations :
  couple (A, b) avec $A \in M(n, m)$ et $b \in R^n$.

- Solution : $x \in R^m$ tel que

$$A \cdot x = b.$$

  C'est-à-dire :

$$\sum_{j=1,\ldots,m} a_{i,j} x_j = b_i, \quad \text{pour tout } i = 1, \ldots, n,$$

  avec $A = (a_{i,j} \mid i = 0, \ldots, n-1, j = 0, \ldots, m-1)$.
- On pense à (A, b) comme une matrice $(A, b) \in M(n, m+1)$.
- On se focalisera sur le cas $A \in M(n, n)$ $(m = n)$

Solution des systèmes avec *numpy*

```python
import numpy as np

def test(n):
    A = np.random.random([n, n])
    b = np.random.random([n])
    x = np.linalg.solve(A, b)
    b_computed= np.dot(A, x)
    err_max = abs(b - b_computed).max()
    return err_max

for i in range(1, 1000):
    print("Size {}, error : {}".format(i, test(i)))
```

Erreurs avec les flottants arrondis

Python 3.6

```
1  x1 = 1./3.
2  x2 = 1./4.
3  x3 = x1 - x2
4  x4 = 12*x3
5  x5 = x4 -1
6  print(x5)
```

Edit this code

point; use the Back and Forward buttons to jump there.

Program terminated Forward > Last >>

Print output (drag lower right corner to resize)

```
-2.220446049250313e-16
```

Frames Objects

Global frame

x1	0.3333
x2	0.25
x3	0.0833
x4	1
x5	0

```
ipdb> p x1
0.3333333333333333
ipdb> p x2
0.25
ipdb> p x3
0.08333333333333331
ipdb> p x4
0.9999999999999998
ipdb> p x5
-2.220446049250313e-16
```

Systèmes triangulaires

On suppose $n = m$. A est une matrice triangulaire supérieure. De la forme

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n &= b_1 \\ a_{2,2}x_2 + \ldots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{n,n}x_n &= b_n \end{aligned}$$

Solution calculée récursivement (substitution en arrière) :

$$x_n = \frac{b_n}{a_{n,n}}, \qquad x_i = \frac{1}{a_{i,i}}(b_i - \sum_{i<j} a_{i,j}x_j).$$

Posons

$$c_n = b_n, \qquad c_i = b_i - \sum_{i<j} a_{i,j}x_j.$$

Si $c_i \neq 0$, on a besoin $a_{i,i} \neq 0$.

Pivot de Gauss

- Méthode pour transformer un système (A, b) en un système (A', b') tel que :
  - $A \cdot x = b$ ssi $A' \cdot x = b'$,
  - A' est triangulaire supérieure.
- Avec l'algorithme de solution des systèmes triangulaires, donne une méthode pour résoudre (A, b).
- On supposera $A \in M(n, n)$,
  mais on peut généraliser à $A \in M(n, m)$.

Opérations élémentaires

Principe :

> Si $C \in M(n,n)$ est inversible, alors
>
> $$A \cdot x = b \quad \text{ssi} \quad C \cdot A \cdot x = C \cdot b\,.$$

- $(A,b)_i$: ligne i du système (A,b)
- Opérations élémentaires sur les lignes :

  Échange : $(A,b)_i, (A,b)_j \leftarrow (A,b)_j, (A,b)_i$

  Transvection : $(A,b)_i \leftarrow (A,b)_i + \lambda (A,b)_j$

  Ces opérations s'expriment comme multiplication par une matrice inversible :

  $$(A,b) \leftarrow C \cdot (A,b)\,.$$

Exemple, avec $n = 3$

$$\left[\begin{array}{ccc|c} 0.93 & 0.74 & 0.54 & 0.86 \\ 0.15 & 0.80 & 0.94 & 0.66 \\ 0.21 & 0.15 & 0.06 & 0.14 \end{array}\right] \rightsquigarrow \left[\begin{array}{ccc|c} 0.93 & 0.74 & 0.54 & 0.86 \\ 0.00 & 0.68 & 0.86 & 0.52 \\ 0.21 & 0.15 & 0.06 & 0.14 \end{array}\right]$$

$$\rightsquigarrow \left[\begin{array}{ccc|c} 0.93 & 0.74 & 0.54 & 0.86 \\ 0.00 & 0.68 & 0.86 & 0.52 \\ 0.00 & -0.02 & -0.06 & -0.05 \end{array}\right]$$

$$\rightsquigarrow \left[\begin{array}{ccc|c} 0.93 & 0.74 & 0.54 & 0.86 \\ 0.00 & 0.68 & 0.86 & 0.52 \\ 0.00 & 0.00 & -0.03 & -0.04 \end{array}\right]$$

Mise en forme triangulaire

Définition

Soient $A \in M(n,n)$ et $1 \leq \ell \leq n$.
A est ℓ-triangulaire si $a_{i,j} = 0$,

$$\text{pour tout } i,j \text{ tels que } j \leq \ell \text{ et } i > j.$$

Exemple : 1-triangualaire, pas 2-triangulaire.

$$\begin{bmatrix} 0.93 & 0.74 & 0.54 \\ 0.00 & 0.68 & 0.86 \\ 0.00 & -0.02 & -0.06 \end{bmatrix}$$

Remarque

- Si $A \in M(n,n)$ est $n-1$-triangulaire, alors elle est n-triangulaire.
- Tout A est 0-triangualaire.

Algorithme

On itère la procédure suivante avec $\ell = 1, \ldots, n-1$.

Entrée : $(A, b) \in M(n, n+1)$ tel que A est $\ell - 1$-triangulaire.
Sortie : $(A', b') \in M(n, n+1)$ tel que

- A' est ℓ-triangulaire,
- (A', b') a le même ensemble de solutions.

(faire une copie de A)

si $a_{i,\ell} = 0$, pour tout $i > \ell$:

    retourner A

sinon :

..... soit $ip \geq \ell$ tel que $a_{ip,\ell} \neq 0$ # choix du pivot

..... $(A, b)_\ell, (A, b)_{ip} \leftarrow (A, b)_{ip}, (A, b)_\ell$

..... pour tout $i > \ell$:

..... $(A, b)_i \leftarrow (A, b)_i - \lambda (A, b)_\ell$ avec $\lambda = \frac{a_{i,\ell}}{a_{\ell,\ell}}$

retourner A

Précision

Pour soucis de précision avec les arrondis
on chosit, dans la procedure, un pivot de valeur absolu maximum.

Entrée : $(A,b) \in M(n,n+1)$ tel que A est $\ell-1$-triangulaire.
Sortie : $(A',b') \in M(n,n+1)$ tel que

si $a_{i,\ell} = 0$, pour tout $i > \ell$:
    retourner A
sinon :
..... soit $ip \geq \ell$ tel que $|a_{ip,\ell}| \geq |a_{i,\ell}|$, pour $\ell \leq i \leq n$
..... $(A,b)_\ell, (A,b)_{ip} \leftarrow (A,b)_{ip}, (A,b)_\ell$
..... pour tout $i > \ell$:
..... $(A,b)_i \leftarrow (A,b)_i - \lambda (A,b)_\ell$ avec $\lambda = \frac{a_{i,\ell}}{a_{\ell,\ell}}$
retourner A

Complexité

- On itère la procédure $n-1$ fois.
- On suppose les operations sur les lignes coûtent $O(1)$.
- On itère la boucle de la procédure

$$\sum_{i=\ell+1,\ldots,n} 1 = \sum_{i=1,\ldots,n-\ell} 1 = \frac{(n-\ell)(n-\ell+1)}{2}$$

  fois.
- Coût de la mise en forme triangulaire :

$$\sum_{\ell=1,\ldots,n-1} \frac{(n-\ell)(n-\ell+1)}{2} = \sum_{i=1,\ldots,n-1} \frac{i(i+1)}{2} \leq (n-1)\frac{n(n+1)}{2} = O(n^3)$$

- Coût de resoudre un système triangulaire : $O(n^2)$
- Coût total : $O(n^3)+O(n^2) = O(n^3)$.

Plan

Les opérations sur les lignes

```python
import numpy as np

def swap_lines(A, i, j):
    """Echange les lignes i et j
    danas la matrice A"""

    tmp = A[i, :].copy()
    A[i, :] = A[j, :]
    A[j, :] = tmp
    return A

def transvection_lines(A, i, j, x):
    """A_j <- A_j + xA_j """

    A[j, :] = A[j, :] + x * A[i, :]
    return A
```

- Que se passe t'il si on remplace

  `tmp = A[i, :].copy()` par `tmp = A[i, :]` ?

Choix du pivot

```
def pivot_index(A, l):
    """Etant donnee une matrice A et un indice l,
    retourne l'indice i >= l t.q. abs(A_{i,l}) est
 ↪  maximum"""

    n = A.shape[0] # nombre de lignes
    i = l
    for k in range(l + 1, n):
        if abs(A[k, l]) > abs(A[i, l]):
            i = k
    return i
```

Mise en forme triangulaire

```
def gauss(A, b):
    A1 = A.copy() # pour ne pas detruire A
    b1 = b.copy()
    n = len(b) # on pourrait aussi la taille de a
    for l in range(n - 1):
        ipiv = pivot_index(A, l)
        if ipiv != l: # echanges
            swap_lines(A1, l, ipiv)
            tmp = b1[ipiv]
            b1[ipiv] = b1[l]
            b1[l] = tmp
        for k in range(l + 1, n): # pivotage
            factor = -A1[k, l] / A1[l, l]
            transvection_lines(A1, l, k, factor)
            b1[k] = b1[k] + b1[l] * factor
    return A1, b1
```

Solutions

```
def solve_triangular(A, b):
    n = len(b)
    x = np.zeros([n]) # vecteur de n zeros
    for i in range(n - 1, -1, -1):
        # x_i = 1/a_{i,i} ( b_i - sum_{j=i+1}^{n} a_{i,j} x_j )
        somme = np.dot(A[i, :], x)
        x[i] = (b[i] - somme) / A[i, i]
    return x

def solve(A, b):
    A1, b1 = gauss(A, b)
    return solve_triangular(A1, b1)
```

Test des valeurs numériques

```
def test():
    n = 5
    # les matrices inversibles sont denses
    A = np.random.random([n, n])
    x = np.random.random([n])
    b = np.dot(A, x)
    x_solved = solve(A, b)
    return abs(x - x_solved).max()

if __name__ == '__main__':
    print(test())
```

Plan

Calcul numérique

- Calcul numérique $=$ calcul en utilisant des nombres, en général en virgule flottante.
- Calcul numérique $\neq$ calcul symbolique.
- Nombre flottant :

$$\text{signe + mantisse + exposant.}$$

  Mantisse : chiffres significatifs.
- Valeur d'un flottant :

$$(-1)^{signe} * 1.mantisse * 2^{exposant}$$

- Précision avec les flottants Python (double précision = 64 bits) :
  - 1 bit de signe
  - 52 bits de mantisse
    $\Rightarrow$ environ 15 à 16 décimales significatives
  - 11 bits d'exposant
    $\Rightarrow$ représentation des nombres de 10^{-308} à 10^{308}

Notion de précision, exemple sur un calcul de π

Par Grégory-Leibniz :

$$\arctan(1) = \frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(1)^n}{2n+1}$$

Objectf : calculer π (avec des nombres flottants, à l'aide cette formule).

Plusieurs problèmes de précision :

- La formule mathématique est infinie.
  Le calcul informatique s'arrêtera après un nombre fini d'itérations.
- Le résultat attendu n'est pas un rationnel.
  Il n'est même **pas un flottant représentable**.
  Le meilleur calcul de π ne pourra donner qu'un arrondi à $\approx 10^{-15}$ près.
- Chaque opération $(/, +, \ldots)$ peut introduire une erreur d'arrondi (erreur relative de 10^{-15}).
  Les erreurs peuvent se cumuler.

Calcul de π en Python

```python
def gregory_leibniz(N):
    result = 0
    denom = 1
    signe = 1
    for k in range(N):
        result = result + signe / float(denom)
        # prepare iteration suivante:
        denom = denom + 2
        signe = -signe
    return result * 4

print(gregory_leibniz(1))
print(gregory_leibniz(10))
print(gregory_leibniz(100))
print(gregory_leibniz(1000))
```

```
4.0
3.0418396189294032
3.1315929035585537
3.140592653839794
```

Affichage de la précision en Python

```
In [1]: # Changer la précision de l'affichage (et
 ↪  seulement de l'affichage)

In [2]: %precision 2
Out[2]: '%.2f'

In [3]: gregory_leibniz(10 ** 4)
Out[3]: 3.14

In [4]: %precision 25
Out[4]: '%.25f'

In [5]: gregory_leibniz(10 ** 4)
Out[5]: 3.1414926535900344894969294

In [6]: import math

In [7]: math.pi
Out[7]: 3.1415926535897931159979635

In [23]: # Retour à la valeur par défaut

In [23]: %precision
Out[23]: '%r'
```

Affichage de la précision (suite)

Critique au transparent précédent ?

Les (non-)propriétés des nombres flottants

- Erreurs d'arrondis :

```
In [1]: 1.0 / 3 - 1.0 / 4 - 1.0 / 12
Out[1]: -1.3877787807814457e-17
```

- Non-commutativité :

```
In [2]: 1 + 1e-16 - 1
Out[2]: 0.0

In [3]: 1 - 1 + 1e-16
Out[3]: 1e-16
```

- Répresentation décimale inexacte :

```
In [4]: 1.2 - 1.0 - 0.2
Out[4]: -5.551115123125783e-17
```

1.2 n'est pas représentable en machine.
L'ordinateur utilise « le flottant représentable le plus proche de 1.2 » à la place ...

Conséquences

- Pas de résultats exacts
- La précision du calcul dépend de beaucoup d'éléments.
- En particulier l'ordre d'évaluation : en général
  la précision est meilleure en sommant du plus petit au plus grand.
- Le test

  ```
  x == 0.0
  ```

  avec `x` un flottant est presque toujours une erreur.
- Si on s'y prend bien, on perd $\approx 10^{-16}$ en précision relative à chaque calcul.
  Acceptable, par rapport à la nature du problème.
- Si on s'y prend mal, le résultat peut être complètement faux !

Calcul à une précision donnée

Par l'exemple : recherche dichotomique d'une zéro d'une fonction.

Soit f une fonction continue qui change de signe entre a et b avec $f(a) < 0$ et $f(b) > 0$.

```
def zero_dichotomie(f, a, b, epsilon):
    while b - a > epsilon:
        pivot = (a + b) / 2.0
        value = f(pivot)
        if value <= 0:
            a = pivot
        else:
            b = pivot
    return a
```

```
In [3]: zero_dichotomie(lambda x : x ** 2 - 2, 0.0, 2.0,
 ↪  1e-15)
Out[3]: 1.414213562373095

In [4]: import math

In [5]: math.sqrt(2)
Out[5]: 1.4142135623730951
```

Pivotage avec Gauss

- Avec l'élimination de Gauss et la substitution en arrière, on a échangé des lignes si $a_{\ell,\ell} = 0/$
- L'échange des lignes est de la forme

$$(A, b)_\ell \leftrightarrow (A, b)_i \,,$$

  où $i \geq \ell$ est tel que $a_{i,\ell} \neq 0$.
- Afin de réduire les erreurs d'arrondis, on a choisi (pivotage partiel) $i \geq \ell$ tel que $|a_{i,\ell}|$ est maximum.
- Le pivotage peut être plus complexe (pivotage complet), avec des colonnes.

Les problèmes

- Si $a_{\ell,\ell}$ est petit par rapport à $a_{i,\ell}$ alors

$$\lambda = \frac{a_{i,\ell}}{a_{\ell,\ell}}$$

  peut être bien plus grand que 1.
- Erreurs d'arrondis dans le calculs de

$$(A,b)_i \leftarrow (A,b)_i - \lambda (A,b)_\ell \,.$$

- Aussi, lors de la résolution du système triangulaire, le calcul

$$x_i = \frac{1}{a_{i,i}}(b_i - \sum_{i<j} a_{i,j} x_j)$$

  avec un petit valeur des $a_{i,i}$ peuvent accroître les erreurs.

Exemple (I)

- Considérons
$$\left[\begin{array}{cc|c} 0.003000 & 59.14 & 59.17 \\ 5.291 & -6.130 & 46.78 \end{array}\right]$$
- Solution exacte $x_1 = 10.00$ et $x_2 = 1.000$.
- Arithmétique en base 10, avec une mantisse de 4.
- Le premier élément pivot $a_{1,1} = 0.003000$ est petit et
$$\lambda = \frac{5.291}{0.003000} = 1763.66 \text{ est arrondi à } 1764.$$
- Le calcul $(A,b)_2 \leftarrow (A,b)_2 - \lambda(A,b)_1$ donne
$$\left[\begin{array}{cc|c} 0.003000 & 59.14 & 59.17 \\ 0 & -104300 & -104400 \end{array}\right]$$

Exemple (II)

- La substition en arrière donne $x_2 = 1.001$, OK.
- Cependant, à cause du petit pivot $a_{1,1} = 0.003000$,

$$x_1 = \frac{59.17 - (59.14)(1.001)}{0.003000} = -10.00$$

contient le peitit erreur 0.001 multiplié par

$$\frac{59.14}{0.003000} \approx 20000$$

Cela détruit la solution exacte $x = 10$.

Plan

Rappels

Soient $u, v \in R^n$.
Produit interne :

$$\langle u, v \rangle := u \cdot v = \sum_{i=1}^{n} u_i v_i$$

Projection sur u :

$$proj_u(v) := \frac{\langle u, v \rangle}{\langle u, u \rangle} u$$

Norme :

$$||v|| := \sqrt{\langle v, v \rangle}$$

Remarque :

$$\langle u, proj_u(v) \rangle = \langle u, \frac{\langle u, v \rangle}{\langle u, u \rangle} u \rangle = \frac{\langle u, v \rangle}{\langle u, u \rangle} \langle u, u \rangle = \langle u, v \rangle$$

L'algorithme de Gram–Schmidt

Base orthogonale et orthonormale ...

Entrée : vecteurs indépendants $v_1, v_2, \ldots, v_n$
Sortie : vecteurs indépendants $e_1, e_2, \ldots, e_n$ tels que

- $Span(v_1, v_2, \ldots, v_n) = Span(e_1, e_2, \ldots, e_n)$
- $e_1, e_2, \ldots, e_n$ est une base ortho-normale : $e_i \cdot e_j = \delta_{i,j}$

Pour $i = 1, \ldots, n$:

$u_i := v_i - \sum_{\ell<i} proj_{u_\ell}(v_i)$

$e_i := \frac{u_i}{||u_i||}$

return $e_1, e_2, \ldots, e_n$

Vérif

$$\langle e_i, e_i \rangle = \langle \frac{u_i}{||u_i||}, \frac{u_i}{||u_i||} \rangle = \frac{\langle u_i, u_i \rangle}{||u_i||^2} = 1 \,.$$

Pour $i < j$:

$$\begin{aligned}
\langle u_i, u_j \rangle &= \langle u_i, v_j - \sum_{\ell<j} proj_{u_\ell}(v_j) \rangle \\
&= \langle u_i, v_j \rangle - \sum_{\ell<j} \langle u_i, proj_{u_\ell}(v_j) \rangle \\
&= \langle u_i, v_j \rangle - \sum_{\ell<j, \ell \neq i} \langle u_i, u_\ell \rangle \lambda_\ell - \langle u_i, proj_{u_i}(v_j) \rangle \\
&= \langle u_i, v_j \rangle - \langle u_i, v_j \rangle = 0 \,.
\end{aligned}$$

Instabilité numérique et Gram-Schmidt modifié

A l'implémentation, les vecteurs u_i peuvent ne pas être orthogonaux, à cause des erreurs d'arrondis.
Au lieu de calculer

$$u_i := v_i - proj_{u_1}(v_i) - \ldots - proj_{u_{i-1}}(v_i)$$

on calcule comme suit :

$$\begin{aligned}
u_i^{(1)} &:= v_i - proj_{u_1}(v_i) \\
u_i^{(2)} &:= u_i^{(1)} - proj_{u_1}(u_i^{(1)}) \\
&\vdots \\
u_i^{(i-1)} &:= u_i^{(i-2)} - proj_{u_{i-1}}(u_i^{(i-2)}) \\
u_i &:= \frac{u_i^{(i-1)}}{||u_i^{(i-1)}||}
\end{aligned}$$

$u_i^{(\ell)}$ est forcé à être orthogonal à $u_i^{(\ell-1)}$ (et aux erreurs introduits avec son calcul).

A vous de le programmer