L'algorithme d'Euclide étendu

Jean-François Burnol
24-26 novembre 2017

Ce texte prend la continuation de
http://jf.burnol.free.fr/agregeuclide_fibo.pdf
et voir également
http://jf.burnol.free.fr/agreg171122fractionscontinues.pdf

1 L'algorithme d'Euclide étendu des informaticiens

Par exemple, le voici pris sur
https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide_%C3%A9tendu
Attention de ne pas confondre le symbole ÷ avec un signe plus ! Le ÷ est utilisé sur la page ci-dessus pour le quotient euclidien de ses opérandes, mais on peut facilement le confondre avec un + si on regarde de loin… (très mauvaise notation, donc, mais ici en PDF c'est plus gros et la confusion est moins probable).

Le texte dit que les entrées sont des entiers naturels, mais en fait la variable b doit être non nulle. C'est peut-être un problème de traduction de l'anglais car dans le monde anglo-saxon $\mathbb{N}$ ne contient pas le zéro.

```
L'affectation est notée « := ».
Cet algorithme est écrit avec des affectations simultanées.
Le quotient euclidien est noté « ÷ ».
Entrée : a, b entiers (naturels)
Sortie : r entier (naturel) et  u, v entiers relatifs
         tels que r = pgcd(a, b) et r = a*u+b*v

Initialisation : (r, u, v, r', u', v') := (a, 1, 0, b, 0, 1)
     q  quotient entier <-- ? pour préciser le type ?

tant que (r' ≠ 0) faire
    q := r÷r'
    (r, u, v, r', u', v') := (r', u', v', r - q*r', u - q*u', v - q*v')
    fait
renvoyer (r, u, v)
Les égalités r = a*u+b*v et r' = a*u'+b*v' sont des invariants de boucle
```

Note : les variables r', u', v' désignent à chaque étape les *nouvelles* choses, tandis que r, u, v stockent les *anciennes*.

Voici maintenant la traduction directe de l'algorithme ci-dessus en langage PYTHON. Le symbole ' étant utilisé pour les chaînes de caractères il est illégal dans les noms de variables. Donc j'ai utilisé `rr` à la place de `r'` etc...

```
def bezout(a, b):
    r, u, v, rr, uu, vv = a, 1, 0, b, 0, 1
    while rr:
        q = r // rr
        r, u, v, rr, uu, vv = rr, uu, vv, r - q * rr, u - q * uu, v - q * vv
    return r, u, v
```

On peut aussi faire ainsi, plus économe en variables et un peu en calculs (mais attention `a` et `b` sont variables maintenant, donc le commentaire ci-dessus sur les invariants de boucle se réfère à leurs valeurs initiales) :

```
def bezout(a, b):
    u, v, uu, vv = 1, 0, 0, 1
    while b:
        q, rr = divmod(a, b)
        a, b = b, rr
        u, uu = uu, u - q * uu
        v, vv = vv, v - q * vv
    return a, u, v
```

Cet algorithme met à jour `uu` et `vv` dans la dernière boucle, ce qui est inutile, puisqu'il renvoie `u` et `v`. On pourrait faire :

```
def bezout(a, b):
    u, v, uu, vv = 1, 0, 0, 1
    while True:
        q, rr = divmod(a, b)
        if not rr:  # si rr vaut 0, fin de l'algorithme
            return b, uu, vv
        a, b = b, rr
        u, uu = uu, u - q * uu
        v, vv = vv, v - q * vv
```

mais on perd probablement plus qu'on a gagné (deux tests booléens à chaque itération), sauf si PYTHON reconnaît `while True:` comme ne nécessitant pas vraiment un test. Voir en dernière page une formulation meilleure.

2 Comment découvrir cet algorithme ?

C'est très simple, on commence par mettre au point une méthode récursive. On a $a, b > 0$ et on cherche à trouver $u * a + v * b = d$ avec $d = \mathrm{PGCD}(a, b)$. L'algorithme d'Euclide repose sur l'invariant

$$\mathrm{PGCD}(a, b) = \mathrm{PGCD}(b, r), \qquad r = a - q * b$$

ce qui pourrait d'ailleurs donner l'implémentation récursive suivante :

```
def pgcd(a, b):
    if not b:  # si b est nul
        return a
    return pgcd(b, a % b)  # a % b est le reste dans la division euclidienne
```

Je répète encore une fois que l'algo suppose les variables positives (sinon il pourrait renvoyer un résultat négatif).

On va donc supposer connaître U et V avec $U * b + V * r = d$, que faut-il prendre pour u et v de sorte que $u * a + v * b = d$?

$$d = U * b + V * r = U * b + V * (a - q * b) = V * a + (U - q * V) * b$$

Donc on prend $u = V$, $v = U - q * V$, d'où l'approche suivante récursive :

```
def bezoutrecursif(a, b):
    if not b:  # si b est nul
        return a, 1, 0
    q, r = divmod(a, b)
    d, U, V = bezoutrecursif(b, r)
    return d, V, U - q * V
```

Notre objectif est de transformer cela en une méthode itérative. La clé est de penser « matrices deux-deux ». Notre formule est

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \begin{pmatrix} U \\ V \end{pmatrix}$$

donc, si $q_1, \ldots, q_N$ sont les quotients successifs pendant le déroulement de l'algorithme, on a :

$$\begin{pmatrix} u \\ v \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix}}_{Q_1} \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & -q_2 \end{pmatrix}}_{Q_2} \cdots \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & -q_N \end{pmatrix}}_{Q_N} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \text{colonne de gauche de } Q_1 Q_2 \ldots Q_N$$

Si on pose $M_j = Q_1 \dots Q_j$ ($M_0 = I_2$), et que l'on note X_j la colonne de gauche et Y_j la colonne de droite on a

$$(X_{j+1} \mid Y_{j+1}) = (X_j \mid Y_j)\begin{pmatrix} 0 & 1 \\ 1 & -q_{j+1} \end{pmatrix} \implies X_{j+1} = Y_j, \quad Y_{j+1} = X_j - q_{j+1}Y_j$$

Donc on a la structure récurrente d'ordre deux :

$$M_j = (Y_{j-1} \mid Y_j) \qquad Y_{j+1} = Y_{j-1} - q_{j+1}Y_j$$

Ainsi, on obtient une structure itérative en notant u et v les entrées de l'ancien Y, uu et vv les entrées du Y courant, et en mettant à jour à chaque étape par les affectations simultanées :

```
u, uu := uu, u - q * uu
v, vv := vv, v - q * vv
```

où `q` est le nouveau quotient de la division euclidienne avec les valeurs courantes de `a` et `b`. Plus précisément :

```
Initialisation
    u = 1, v = 0, uu = 0, vv = 1  % colonnes de la matrice identité
Mise à jour
    q, rr := quotient et reste dans la division euclidienne de a par b
    u, uu := uu, u - q * uu
    v, vv := vv, v - q * vv
    a, b  := b, rr  % rr est la même chose que a - q * b
Critère d'arrêt
    b = 0
Valeurs renvoyées en sortie
    a, u, v
Remarque: le dernier q a mis à jour uu et vv, mais n'intervient
    pas pour les valeurs de u et v renvoyées, qui sont les uu et vv
    avant cette ultime mise à jour.
```

On voit qu'on a reconstitué exactement l'algorithme donné initialement.

3 La variante de certains mathématiciens méritants

À la fin de l'algorithme on a donc

$$\begin{pmatrix} u & uu \\ v & vv \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix}}_{Q_1}\underbrace{\begin{pmatrix} 0 & 1 \\ 1 & -q_2 \end{pmatrix}}_{Q_2}\cdots\underbrace{\begin{pmatrix} 0 & 1 \\ 1 & -q_N \end{pmatrix}}_{Q_N}$$

Passons à la transposée

$$\begin{pmatrix} u & v \\ uu & vv \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_N \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix}$$

puis à l'inverse. Le déterminant vaut $(-1)^N$.

$$(-1)^N \begin{pmatrix} vv & -v \\ -uu & u \end{pmatrix} = \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_N & 1 \\ 1 & 0 \end{pmatrix}$$

On voit que cela donne, si l'on veut, naissance à un algorithme alternatif qui ne fait que des additions, et maintenant la structure est

$$(W_n \mid Z_n) = \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_n & 1 \\ 1 & 0 \end{pmatrix}$$

$$(W_{n+1} \mid Z_{n+1}) = (W_n \mid Z_n) \cdot \begin{pmatrix} q_{n+1} & 1 \\ 1 & 0 \end{pmatrix}$$

$$W_{n+1} = q_{n+1} W_n + Z_n, \quad Z_{n+1} = W_n$$

$$(W_n \mid W_{n-1}) = \begin{pmatrix} q_1 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_n & 1 \\ 1 & 0 \end{pmatrix}$$

$$W_{n+1} = q_{n+1} W_n + W_{n-1}$$

Voici une implémentation en Python. Au risque de confusions possibles j'utilise les mêmes notations en `uu`, `vv`, `u`, `v`. Les variables doublées sont à nouveau là pour dénoter les « nouvelles » et les variables simples les « anciennes ». Attention que maintenant dans la matrice à entrées positives `v, vv` sont « en haut » et `u, uu` sont « en bas » et les « nouvelles » sont la colonne de gauche, pas de droite. Bien sûr on voit en comparant que c'est tout comme l'algorithme « des informaticiens » après extraction des signes.

```
def bezout(a, b):
    vv, uu, v, u = 1, 0, 0, 1
    e = 1
    while b:
        q, rr = divmod(a, b)
        a, b = b, rr
        vv, v = q * vv + v, vv
        uu, u = q * uu + u, uu
        e = -e
    return a, e * u, - e * v
```

Si vous voulez tester :

```
>>> import random
>>> for i in range(20):
...     a = random.randrange(0, 1000000)
...     b = random.randrange(0, 1000000)
...     print(a, b)
...     d, u, v = bezout(a, b)
...     print('   ', d, u, v)
...     print('   ', u * a + v * b)
...
```

produit (je mets sur deux colonnes pour ne pas entamer une nouvelle feuille) :

```
87849 45803
    1 -20640 39587
    1
811700 731501
    1 -33976 37701
    1
866113 234531
    1 99211 -366382
    1
596099 579375
    1 163274 -167987
    1
21437 401164
    1 -8103 433
    1
841569 963046
    1 381375 -333269
    1
345805 833478
    1 267907 -111153
    1
706657 406244
    1 -92499 160901
    1
41468 782779
    1 84662 -4485
    1
906866 602346
    2 -36983 55680
    2
```

```
565369 131957
    7 6791 -29096
    7
803745 63802
    1 -3903 49168
    1
38393 621734
    1 221857 -13700
    1
868003 206860
    1 -98853 414796
    1
687073 174165
    1 49777 -196368
    1
674431 374967
    1 22810 -41027
    1
337355 983016
    1 455147 -156199
    1
105097 346170
    1 115603 -35097
    1
333485 215603
    1 87178 -134843
    1
29665 158140
    5 9345 -1753
    5
```

4 On revient sur l'implémentation de l'algorithme standard

Ah, je me disais bien que je ratais quelque chose. Je reviens à l'algorithme pioché sur https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide_%C3%A9tendu.

Finalement je propose ceci

```
def bezout(a, b):
    if b == 0:
        assert a != 0, "Pas de bézouts pour les nuls ;-)"
        return a, 1, 0
    u, v, uu, vv = 1, 0, 0, 1
    q, rr = divmod(a, b)
    while rr:
        a, b = b, rr
        u, uu = uu, u - q * uu
        v, vv = vv, v - q * vv
        q, rr = divmod(a, b)
    return b, uu, vv
```

Vous voyez que ceci évite une mise à jour inutile... le dernier `q` est calculé, mais pas utilisé car il ne sert pas aux coefficients de Bézout. J'en ai profité pour ajouter un test pour la nullité de `b`, car comme on peut utiliser l'algorithme avec `a==0` il est plus élégant de pouvoir le faire aussi avec `b==0`.

Mais il faut un algorithme acceptant les entiers relatifs (par exemple pour trouver des inverses de classes de congruence sans contraintes excessives sur les entrées). De plus, l'exclusion du cas `a==b==0` semble excessive. Donc, (en passant par la variante « positive ») :

```
def bezout(a, b):
    if b == 0:
        if a == 0:
            return 0, 0, 0  # l'idéal 0.Z + 0.Z est 0.Z
        elif a > 0:
            return a, 1, 0
        else:
            return -a, -1, 0
    b, sb = abs(b), -1 if b < 0 else 1  # pas de "sign(x)" natif en Python !
    vv, uu, v, u = 1, 0, 0, 1
    e = 1
    q, rr = divmod(a, b)
```

```
    # ici b > 0,
    #  1. en Python, rr = a % b donne 0 <= rr < b, même avec a < 0
    #  2. le premier q sera négatif si le premier a est < 0
    #  3. le premier q sera nul si 0 <= a < b
    #  4. quoi qu'il arrive, les q d'après, s'il y en a, seront > 0.
    while rr:
        a, b = b, rr
        vv, v = q * vv + v, vv
        uu, u = q * uu + u, uu
        e = -e
        q, rr = divmod(a, b)
    return b, -e * uu, sb * e * vv
```

L'algorithme ci-dessus a un défaut, pas évident à anticiper : pour `b==2*N` positif et pair, `a==-N` négatif, il renvoie `N, 1, 1` et non pas `N, -1, 0`. Certes c'est valable mais on a l'impression qu'il ne fournit pas une solution « minimale ». Donc, je préfère ceci au final :

```
def bezout(a, b):
    if b == 0:
        if a == 0:
            return 0, 0, 0  # l'idéal 0.Z + 0.Z est 0.Z
        return abs(a), -1 if a < 0 else 1, 0
    a, sa = abs(a), -1 if a < 0 else 1
    b, sb = abs(b), -1 if b < 0 else 1
    vv, uu, v, u = 1, 0, 0, 1
    e = 1
    # b > 0, mais 0 <= a < b est possible, ce qui donne un q nul, et si
    # a est nul alors rr aussi et on ne rentre pas dans la boucle sinon
    # le premier a, b = b, rr en fait réalise l'échange de a et de b
    q, rr = divmod(a, b)
    while rr:
        a, b = b, rr
        vv, v = q * vv + v, vv
        uu, u = q * uu + u, uu
        e = -e
        q, rr = divmod(a, b)
    return b, -sa * e * uu, sb * e * vv
```