3
Algorithme d'Euclide et résolution de l'équation de Bézout

I
Algorithme de Blankinship

1. L'algorithme d'Euclide permet de calculer par divisions euclidiennes successives le pgcd d d'une famille finie $(v_i)_{0\leqslant i<n}$ d'entiers naturels.
Il existe alors une famille $(a_i)_{0\leqslant i<n}$ d'entiers relatifs qui vérifient l'équation de Bézout :

$$\sum_{0\leqslant i<n} a_i v_i = d.$$

2. L'algorithme de Blankinship, variation de l'algorithme d'Euclide, permet de calculer une solution particulière de l'équation de Bézout.
2.1 Initialement, on considère la matrice

$$M_0 = \in \mathfrak{M}_{n,n+1}(\mathbb{Z}).$$

2.2 Pour passer de la matrice M_k à la matrice M_{k+1},
– on choisit un élément non nul $x_{i_0,n}$ de la dernière colonne de M_k ;
– pour chaque $i \neq i_0$, on effectue la division euclidienne de $x_{i,n}$ par $x_{i_0,n}$:

$$x_{i,n} = q_i x_{i_0,n} + r_i$$

puis l'opération de pivot

$$L_i \leftarrow L_i - q_i L_{i_0}$$

sur la matrice M_k.
2.3 Le processus s'arrête lorsque la dernière colonne de la matrice M_k ne contient plus qu'un seul coefficient $x_{i_0,n}$ non nul.
– Cet unique coefficient non nul est le pgcd de la famille $(v_i)_{0\leqslant i<n}$.
– Les n premiers coefficients $(x_{i_0,j})_{0\leqslant j<n}$ situés sur la même ligne donnent une solution de l'équation de Bézout :

$$\sum_{0\leqslant j<n} x_{i_0,j} v_j = x_{i_0,n}.$$

Q 1. *On reprend les notations de* [**2**].
Q 1.a *Quels sont les coefficients situés sur la dernière colonne de M_{k+1} ?*
Q 1.b *Pourquoi a-t-on intérêt à choisir pour $x_{i_0,n}$ le plus petit élément non nul de la dernière colonne de M_k ?*

Q 2. Test d'arrêt

Soit `V`*, un tableau unidimensionnel d'entiers naturels.*
Comment définir une fonction booléenne `continuer(V)` *qui retourne* `True` *si, et seulement si, le tableau* `V` *contient au moins deux entiers non nuls ?*
On pourra utiliser les méthodes de numpy, en particulier `np.sum`*.*

Q 3. Localisation du pivot

Soit `V`*, un tableau unidimensionnel d'entiers naturels. On sait que les éléments de ce tableau ne sont pas tous nuls.*
Compléter le code suivant pour que la fonction `indice_min(V)` *retourne l'indice du plus petit élément non nul du tableau* `V` *(ou, en cas d'*ex æquo*, l'indice de l'un des plus petits éléments non nuls du tableau).*

```
def indice_min(V):
  longueur = V.size
  i_min, min_non_nul = 0, V[0]
```

Q 4. *Proposer une fonction* `Blankinship(V)` *dont l'unique argument est une colonne de n entiers naturels (c'est-à-dire un tableau de format $(n,1)$) et qui retourne le couple (a,d) où d est le pgcd des éléments $(v_i)_{0\leqslant i<n}$ et $a=(a_i)_{0\leqslant i<n}$, un tableau d'entiers relatifs de format $(n,)$ tel que*

$$\sum_{0\leqslant i<n} a_i v_i = d.$$

On définira la matrice $M_0 \in \mathfrak{M}_{n,n+1}(\mathbb{Z})$ avec le code suivant.

```
nb_entiers = len(V)
I = np.identity(nb_entiers)
M = np.hstack((I,V))
```

On rappelle que la dernière colonne du tableau `M` *s'obtient avec* `M[:,-1]`*.*

II

Application : lemme chinois

3. Étant données une famille $(m_i)_{0\leqslant i<n}$ de **modules** (des entiers naturels) et une famille $(r_i)_{0\leqslant i<n}$ de **restes**, c'est-à-dire des entiers tels que

$$\forall\, 0 \leqslant i < n, \quad 0 \leqslant r_i < m_i,$$

on cherche un entier relatif x, aussi petit que possible, tel que

$$(\star) \qquad \forall\, 0 \leqslant i < n, \quad x = r_i \pmod{m_i}.$$

4. Lorsque les modules m_i sont deux à deux premiers entre eux, le lemme chinois assure l'existence d'un tel entier relatif x et la démonstration du lemme donne une méthode explicite pour calculer cet entier.

Q 5. *Expliquer pourquoi il n'existe aucun entier relatif x tel que*

$$x = 7 \pmod{15} \quad \textit{et} \quad x = 12 \pmod{21}.$$

5. Pour tout $0 \leqslant i < n$, on définit l'entier naturel μ_i par

$$\mu_i = \prod_{\substack{0 \leqslant k < n \\ k \neq i}} m_k.$$

5.1 Comme les modules m_i sont deux à deux premiers entre eux, les μ_i sont premiers dans leur ensemble et il existe une famille $(a_i)_{0 \leqslant i < n}$ d'entiers relatifs tels que

$$\sum_{0 \leqslant i < n} a_i \mu_i = 1.$$

5.2 Pour $0 \leqslant i < n$, on pose alors

$$e_i = a_i \mu_i \in \mathbb{Z}.$$

Ces entiers vérifient :

$$e_i = 1 \pmod{m_i} \quad \text{et} \quad \forall\, j \neq i, \quad e_i = 0 \pmod{m_j}.$$

5.3 Par conséquent, l'entier

$$x_0 = \sum_{0 \leqslant i < n} r_i e_i$$

est une solution particulière du système $(\star)$.

5.4 Enfin, un entier x est solution du système $(\star)$ si, et seulement si,

$$\exists\, k \in \mathbb{Z}, \qquad x = x_0 + k \prod_{0 \leqslant i < n} m_i.$$

Q 6. *On dispose de deux listes* `modules` *et* `restes` *qui contiennent respectivements les modules m_i et les restes r_i.*

Q 6.a *On veut construire un tableau* `Mu` *qui contient les entiers μ_i pour en déduire un tableau* `A` *qui contienne les entiers a_i à l'aide de l'instruction suivante.*

```
A = Blankinship(Mu)[0]
```

Quel doit être le format du tableau `Mu` *?*
Construire le tableau `Mu` *en respectant cette contrainte de format. On pourra utiliser la fonction* `np.product`*.*
Quel est alors le format du tableau `A` *?*

Q 6.b *Comment construire le tableau* `base_can` *qui contient les entiers e_i ?*

Q 6.c *En déduire une solution particulière de $(\star)$, puis la plus petite solution de $(\star)$ en valeur absolue.*

Réponses aux questions

I Algorithme de Blankinship

R 1.a La dernière colonne de M_{k+1} contient le diviseur $x_{i_0,n}$ sur la ligne i_0 et les restes r_i sur chacune des autres lignes.

R 1.b Par définition de la division euclidienne, le reste r_i vérifie :

$$0 \leqslant r_i < x_{i_0,n}.$$

Plus $x_{i_0,n}$ est petit, plus les coefficients de la dernière colonne de M_{k+1} seront petits : on se donne ainsi le moyen d'arriver au résultat en faisant le plus petit nombre d'opérations.

R 2. Test d'arrêt

Première méthode : comme le tableau `V` contient des *entiers positifs*, il contient au moins deux entiers *strictement* positifs si, et seulement si, la somme des éléments de `V` est strictement supérieure au plus grand élément de `V`.

```
def continuer(V):
  return np.sum(V)>np.max(V)
```

Deuxième méthode : on compare chaque élément de `V` à 0 et on compte le nombre de succès. Comme le nombre n de succès est un *entier*, $n \geqslant 2$ équivaut à $n > 1$.

```
def continuer(V):
  cptr = 0
  for x in V:     # On parcourt la liste.
    if (x>0):     # À chaque élément non nul,
      cptr += 1   # on incrémente le compteur.
  return (n>1)
```

Deuxième méthode, version rapide : En python, les booléens `True` et `False` sont identifiés à 1 et 0, donc la somme d'un tableau booléen est égale au nombre d'éléments de ce tableau qui ont la valeur `True`.

```
def continuer(V):
  positifs = (V>0)
  return np.sum(positifs)>1
```

R 3. Localisation du pivot

On va parcourir le tableau pour comparer ses éléments à 0.
Si `V[0]` est nul, on cherche d'abord une première valeur non nulle dans le tableau pour servir de référence. (Comme le tableau n'est pas identiquement nul, une telle valeur existe.)
Ensuite, à chaque fois qu'on rencontre une valeur *non nulle* et *strictement inférieure* à la valeur de `min_non_nul`, on enregistre la valeur de l'indice ainsi que la nouvelle valeur de `min_non_nul`.

```
def indice_min(V):
  longueur = V.size
  i_min, min_non_nul = 0, V[0]
  # Recherche d'une valeur de référence non nulle
  i = 0
  while (min_non_nul==0):
    if (V[i]>0):
      i_min, min_non_nul = i, V[i]
    i += 1    # Boucle while : penser à incrémenter le compteur !
  # Localisation de la plus petite valeur non nulle dans le reste du tableau
  for j in range(i, longueur):
    if (V[j]>0) and (V[j]<min_non_nul):
      i_min, min_non_nul = j, V[j]
  return i_min
```

Variante (qui ne respecte pas les contraintes de l'énoncé) : On prend comme première valeur de référence le maximum du tableau (qui est strictement positif, puisque le tableau n'est pas identiquement nul). On passe ensuite en revue tous les éléments du tableau en notant à chaque fois qu'on trouve une valeur inférieure à la valeur de référence.

```
def indice_min(V):
  min_non_nul = np.max(V)
  for i in range(V.size):
    if (V[i]!=0) and (V[i]<=m):
      i_min, min_non_nul = i, V[i]
  return i_min
```

REMARQUE.— Le test porte bien sur `(V[i]<=m)` et non pas sur `(V[i]<m)`. En effet, lorsque le tableau n'a plus qu'un seul élément non nul, cet élément est le maximum et on ne trouvera *aucun* indice i tel que $V_i < m$, de telle sorte que la variable `i_min` n'a aucune valeur à la sortie de la boucle `for` !

R 4. On initialise le tableau `M`. On effectue ensuite les opérations de pivot sur les lignes en suivant la méthode exposée plus haut. Le processus s'arrête quand le booléen `continuer(M[:,-1])` devient faux, c'est-à-dire lorsque la dernière colonne de `M` ne contient plus qu'un seul coefficient non nul. On localise ce coefficient à l'aide de la fonction `indice_min` et on retourne la ligne convenable en deux morceaux : les premiers coefficients donnent une solution de l'équation de Bézout ; le dernier coefficient donne le pgcd.

```
def Blankinship(V):
  nb_entiers = len(V)
  I = np.identity(nb_entiers)
  M = np.hstack((I,V))
  while (continuer(M[:,-1])):
    i_min = indice_min(M[:,-1])  # position du plus petit élément non nul
    for i in range(nb_entiers):
      if (i!=i_min):
        q = M[i,-1]//M[i_min,-1]  # quotient de la division euclidienne
        M[i,:] -= q*M[i_min,:]    # opération de pivot sur la ligne i
  i0 = indice_min(M[:,-1])
  a, d = M[i0,:-1], M[i0,-1]
  return a, d
```

Le format de `a` est du type $(n,)$.

II Application : lemme chinois

R 5. Si $x = 7 \pmod{15}$, alors $x = 1 \pmod 3$ tandis que si $x = 12 \pmod{21}$, alors $x = 0 \pmod 3$. Ces deux équations sont contradictoires.
Le lemme chinois montre en particulier qu'une telle contradiction ne peut apparaître lorsque les modules m_i sont deux à deux premiers entre eux. (Ici, les modules 15 et 21 ne sont pas premiers entre eux.)

R 6.a D'après **[Q4.]**, le format du tableau `Mu` doit être de la forme $(n, 1)$. Il s'agit donc d'un tableau *bi*dimensionnel.
Pour calculer les μ_i, le plus simple est de remarquer que

$$\forall\, 0 \leqslant i < n, \quad m_i \mu_i = \prod_{0 \leqslant k < n} m_k$$

c'est-à-dire que μ_i est le quotient de la division du produit des m_k par m_i. On pose donc :

```
Module = np.product(modules)
```

ce qui nous servira encore pour appliquer la formule générale du **[5.4]**.

Il y a (au moins) deux manières de construire la colonne `Mu` :
– On construit une liste de listes que l'on convertit en tableau du bon format.

```
Mu = np.array([[Module//m] for m in modules])
```

– On construit une liste simple, qu'on convertit explicitement en tableau du format convenable.

```
Mu = [Module//m for m in modules]
nb_mod = len(modules)
Mu = np.array(Mu).reshape((nb_mod,1))
```

D'après **[Q4.]**, le tableau `A` est *uni*dimensionnel et son format est de la forme $(n,)$.

R 6.b Il faut multiplier chaque élément du tableau `A` avec l'élément correspondant du tableau `Mu` : pour cela, il faut d'abord donner le *même format* à ces deux tableaux.

```
Mu = Mu.reshape(nb_mod,)
base_can = A*Mu
```

Le tableau `base_can` est alors unidimensionnel, de format $(n,)$.
L'exécution du code suivant permet de vérifier les calculs.

```
for e in base_can:
  print([e%m for m in modules])
```

R 6.c D'après **[5.3]**, on obtient une solution particulière par

```
sol_part = np.sum(base_can*residus)
```

ou par :

```
sol_part = np.dot(base_can, residus)   # Produit scalaire canonique
```

Notons M, le produit des modules m_i. La formule générale du **[5.4]** peut aussi se lire sous la forme :

$$x_0 = -kM + x$$

qui doit évoquer une division euclidienne. On en déduit que le reste R de la division euclidienne de x_0 par M est la plus petite solution *positive* de $(\star)$.
Pour en déduire la plus petite solution en valeur absolue, il suffit de comparer le reste R avec $R - M$, qui est la plus grande solution négative de $(\star)$. Comme $R \geqslant 0$ et que $R - M < 0$, alors

$$\begin{aligned} R < |R - M| &\iff R < M - R \\ &\iff R < {}^{M}/_{2}. \end{aligned}$$

```
sol_mini_pos = sol_part % Module     # Reste de la division euclidienne
if (sol_mini_pos < Module//2):
    sol_mini = sol_mini_pos
else:
    sol_mini = sol_mini_pos-Module
```