Examen Programmation ENSAE première année 2006
Examen écrit (1 heure)

Lors de la correction, je n'ai pas enlevé de points pour les erreurs de syntaxe et accordé les points de la question à partir du moment où l'idée principale de l'algorithme était présente même si le programme ne retournait pas exactement le résultat. Devant un ordinateur, après quelques essais, vous auriez, à partir de votre première réponse, rapidement obtenu un programme correct.

0.0.1 Programme à deviner *

Enoncé

Que fait le programme suivant ? Que contiendra `res` à la fin du programme ? (1 point)

```
l = [ 0,1,2,3,4,6,5,8,9,10]
res = True
for i in range (1,len (l)) :
    if l[i-1] > l[i] :
        res = False
```

Correction

Le programme vérifie que la liste `l` est triée, `res` vaut `True` si elle est triée, `False` sinon. Et dans ce cas précis, elle vaut `False` car la liste n'est pas triée (4,6,5). **fin exo 0.0.1** □

0.0.2 Somme des chiffres d'un nombre *

Enoncé

Ecrire une fonction qui calcule la somme des chiffres d'un entier positif. (2 points)

Correction

Tout d'abord, obtenir la liste des chiffres d'un nombre entier `n` n'est pas immédiat bien qu'elle puisse s'écrire en une ligne :

```
def somme (n) :
    return sum ( [ int (c) for c in str (n) ] )
```

Cette écriture résumée cache une conversion en chaîne de caractères. Une boucle est inévitable à moins de procéder par récurrence. Commençons par une version plus étendue et qui passe par les chaînes de caractères. Pour cette version, il ne faut pas oublier de faire la conversion inverse, c'est-à-dire celle d'un caractère en nombre.

```
def somme (n) :
    l = str (n)           # il ne faut pas confondre l=str (n) avec l = "n"
    s = 0
    for c in l :          # ou   for i in range (0, len (c)) :
        s += int (c)      # ou       s += int (c [i])
    return s
```

Une version numérique maintenant, celle qui utilise l'opération % ou modulo pour obtenir le dernier chiffre :

```
def somme (n) :
    s = 0
    while n > 0 :
        s += n % 10
        n /= 10    # ici, c'est une division entière, si vous n'êtes pas sûr :
                   #       n = int (n/10)
    return n
```

Enfin, une autre solution utilisant la récurrence et sans oublier la condition d'arrêt :

```
def somme (n) :
    if n <= 0 : return 0
    else : return (n % 10) + somme ( n / 10 )
```

Parmi les autres solutions, certaines, exotiques, ont utilisé la fonction `log` en base 10 ou encore la fonction `exp`. En voici une :

```
import math
def somme (n) :
    k = int (math.log (n) / math.log (10) + 1)
    s = 0
    for i in range (1,k+1) :
        d = 10 ** i   # ou encore d = int (exp ( k * log (10) ) )
        c = n / d
        e = n - c * d
        f = e / (d / 10)
        s += f
    return s
```

L'idée principale permet de construire une fonction retournant le résultat souhaité mais il faut avouer qu'il est plus facile de faire une erreur dans cette dernière fonction que dans les trois précédentes.

fin exo 0.0.2 □

0.0.3 Calculer le résultat d'un programme **

Enoncé

Que vaut `n` à la fin de l'exécution du programme suivant ? Expliquez en quelques mots le raisonnement suivi. (2 points)

```
n = 0
for i in range (0,10) :
    if (n + i) % 3 == 0 :
        n += 1
```

Correction

L'ensemble `range`$(0, 10)$ est égale à la liste $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$. Dans ce programme, `n` est incrémenté lorsque `i` + `n` est un multiple de 3. La première incrémentation a lieu lors de la première itération puisque 0 est un multiple de 3. On déroule le programme dans le tableau suivant :

i	0	1	2	3	4	5	6	7	8	9
n avant le test	0	1	1	2	2	3	3	4	4	5
i+n	0	2	3	5	6	8	9	11	12	14
n après le test	1	1	2	2	3	3	4	4	5	5

`range`$(0, 10)$ contient les nombres de 0 à 10 **exclu**. La réponse cherchée est donc 5. **fin exo 0.0.3** □

0.0.4 Suite récurrente (Fibonacci) *

Enoncé

Ecrire un programme qui calcule l'élément u_{13} de la suite définie par :

$$\begin{array}{rcl} u_1 & = & 1 \\ u_2 & = & 2 \\ \forall n \geqslant 2, \; u_n & = & u_{n-1} + u_{n-2} \end{array}$$

u_{13} est le nombre de manières possibles pour une grenouille de monter en haut d'une échelle de 13 barreaux si elle peut faire des bonds de un ou deux barreaux seulement. Si cette affirmation, qui n'est pas à démontrer, vous semble douteuse, avant d'y revenir, faites d'abord le reste de l'énoncé. (2 points)

Correction

Tout d'abord, la grenouille : elle peut faire des bonds de un ou deux barreaux à chaque fois. Donc, lorsqu'elle est sur le barreau n, elle a pu venir du barreau $n-1$ et faire un saut de 1 barreau ou venir du barreau $n-2$ et faire un bond de 2 barreaux. Par conséquent, le nombre de manières qu'a la grenouille d'atterrir sur le barreau n, c'est la somme des manières possibles de venir jusqu'au barreau $n-1$ et du barreau $n-2$.

Maintenant, comment le programmer ? Tout d'abord les solutions récursives, les plus simples à programmer :

```
def grenouille (n) :
    if   n == 2 : return 2
    elif n == 1 : return 1
    else :        return grenouille (n-1) + grenouille (n-2)
print grenouille (13)
```

Cette solution n'est pas très rapide car si `grenouille(13)` va être appelée une fois, `grenouille(12)` une fois aussi, mais `grenouille(11)` va être appelée deux fois... Cette solution implique un grand nombre de calculs inutiles.

Après la solution récursive descendante (`n` décroît), la solution récursive montante (`n` croît) :

```
def grenouille (fin, n = 2, u1 = 1, u2 = 2) :
    if   fin == 1   : return u1
    elif fin == 2   : return u2
    elif n   == fin : return u2
    u = u1 + u2
    return grenouille (fin, n+1, u2, u)
print grenouille (13)
```

La méthode la plus simple est non récursive mais il ne faut pas se tromper dans les indices :

```
def grenouille (n) :
    if n == 1 : return 1
    u1 = 1
    u2 = 2
    for i in range (3,n+1) :
        u  = u1 + u2   # il est impossible de
        u1 = u2        # résumer ces trois lignes
        u2 = u         # en deux
    return u2
print grenouille (13)
```

Quelques variantes utilisant les listes :

```
def grenouille (n) :
    if n == 1 : return 1
    u = [1,2]
    for i in range (2,n) :
        u.append (u [i-1] + u [i-2])
    return u [n-1]
print grenouille (12)
```

Ou encore :

```
def grenouille (n) :
    if n == 1 : return 1
    u = range (0, n)  # il ne faut pas oublier de créer le tableau
                      # avec autant de cases que nécessaire
                      # ici 13
    u [0] = 1
    u [1] = 2
    for i in range (2,n) :
        u [i] = u [i-1] + u [i-2]
    return u [n-1]
print grenouille (12)
```

fin exo 0.0.4 □

0.0.5 Comprendre une erreur d'exécution *

Enoncé

On écrit le programme suivant :

```
a = "abcdefghijklmnopqrstuvwxyz"
print len (a)
d = {}
for i in range (0,len (a)) :
    d [ a [ i ] ] = i

print d ["M"]
```

Une erreur est déclenchée lors de l'exécution :

```
examen.py:14:  KeyError: 'M'
```

Que faudrait-il écrire pour que le programme marche ? Qu'affichera-t-il alors ? (1 point)

Correction

L'erreur signifie que la clé "M" n'est pas présente dans le dictionnaire d. Elle n'est présente qu'en minuscule. La modification proposée est la suivante :

```
a = "abcdefghijklmnopqrstuvwxyz"
print len (a)
d = {}
for i in range (0,len (a)) :
    d [ a [ i ] ] = i

print d ["m"]         ######   ligne modifiée
```

Mais il était également possible de procéder comme suit :

```
a = "abcdefghijklmnopqrstuvwxyz"
a = a.upper ()        #######   ligne ajoutée
print len (a)
d = {}
for i in range (0,len (a)) :
    d [ a [ i ] ] = i

print d ["M"]
```

Dans les deux cas, le programme affiche la position de `M` dans l'alphabet qui est 12 car les indices commencent à 0.

fin exo 0.0.5 □

0.0.6 Copie de variables **

Enoncé

```
def somme (tab) :
    l = tab[0]
    for i in range (1, len (tab)) :
        l += tab [i]
    return l
ens = [[0,1],[2,3]]
print somme ( ens )   # affiche [0,1,2,3]
print ens             # affiche [ [0,1,2,3], [2,3] ]
```

La fonction `somme` est censée faire la concaténation de toutes les listes contenues dans `ens`. Le résultat retourné est effectivement celui désiré mais la fonction modifie également la liste `ens`, pourquoi ? (2 points)

Correction

Le problème vient du fait qu'une affectation en *Python* (seconde ligne de la fonction `somme`) ne fait pas une copie mais crée un second identificateur pour désigner la même chose. Ici, `l` et `tab`[0] désignent la même liste, modifier l'une modifie l'autre. Ceci explique le résultat. Pour corriger, il fallait faire une copie explicite de `tab`[0] :

```
import copy                      ###### ligne ajoutée
def somme (tab) :
    l = copy.copy (tab[0])       ###### ligne modifiée
    for i in range (1, len (tab)) :
        l += tab [i]
    return l
ens = [[0,1],[2,3]]
print somme ( ens )   # affiche [0,1,2,3]
print ens             # affiche [ [0,1,2,3], [2,3] ]
```

Il était possible, dans ce cas, de se passer de copie en écrivant :

```
def somme (tab) :
    l = []                           ###### ligne modifiée
    for i in range (0, len (tab)) :  ###### ligne modifiée
        l += tab [i]
    return l
ens = [[0,1],[2,3]]
print somme ( ens )   # affiche [0,1,2,3]
print ens             # affiche [ [0,1,2,3], [2,3] ]
```

fin exo 0.0.6 □

0.0.7 Comprendre une erreur d'exécution **

Enoncé

On écrit le programme suivant :

```
li  = range (0,10)
sup = [0,9]
for i in sup :
    del li [i]
print li
```

Mais il produit une erreur :

```
examen.py:44:  IndexError: list assignment index out of range
```

Que se passe-t-il ? Comment corriger le programme pour qu'il marche correctement ? Qu'affichera-t-il alors ? (2 points)

Correction

L'erreur signifie qu'on cherche à accéder à un élément de la liste mais l'indice est soit négatif, soit trop grand. Pour comprendre pourquoi cette erreur se produit, il faut suivre les modifications de la liste `li` dans la boucle `for`. Au départ, `li` vaut `range(0,10)`, soit :

```
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Lors du premier passage de la boucle, on supprime l'élément d'indice 0, la liste `li` est donc égale à :

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Et elle ne contient plus que 9 éléments au lieu de 10, donc le dernier élément est celui d'indice 8. Or lors du second passage dans la boucle `for`, on cherche à supprimer l'élément d'indice 9, ce qui est impossible. L'idée la plus simple pour corriger l'erreur est de supprimer les éléments en commençant par ordre d'indices décroissant :

```
li  = range (0,10)
sup = [9,0]      ####### ligne modifiée
for i in sup :
    del li [i]
print li
```

Mais on pouvait tout à fait écrire aussi `sup` $= [0,8]$ même si ce n'est la réponse que je conseillerais. L'objectif du programme n'était pas de supprimer toute la liste `li`.

Une dernière précision, pour supprimer l'élément d'indice `i` de la liste `li`, on peut écrire soit `del li[i]` soit `del li[i:i+1]`.

fin exo 0.0.7 □

0.0.8 Comprendre une erreur de logique **

Enoncé

Le programme suivant fonctionne mais le résultat n'est pas celui escompté.

```
l = ["un", "deux", "trois", "quatre", "cinq"]
for i in range (0,len (l)) :
    mi = i
    for j in range (i, len (l)) :
        if l[mi] < l [j] : mi = j
    e      = l [i]
    l [mi] = l [i]
    l [i]  = e
print l
```

Le résultat affiché est :

```
['un', 'deux', 'deux', 'deux', 'cinq']
```

Qu'est censé faire ce programme ? Quelle est l'erreur ? (2 points)

Correction

Ce programme est censé effectuer un tri par ordre alphabétique **décroissant**. Le problème intervient lors de la permutation de l'élément `l[i]` avec l'élément `l[mi]`. Il faut donc écrire :

```
l = ["un", "deux", "trois", "quatre", "cinq"]
for i in range (0,len (l)) :
    mi = i
    for j in range (i, len (l)) :
        if l[mi] < l [j] : mi = j
    e      = l [mi]       ######## ligne modifiée
    l [mi] = l [i]
    l [i]  = e
print l
```

fin exo 0.0.8 □

0.0.9 Coût d'un algorithme **

Enoncé

Le coût d'un algorithme ou d'un programme est le nombre d'opérations (additions, multiplications, tests, ...) qu'il effectue. Il s'exprime comme un multiple d'une fonction de la dimension des données que le programme manipule. Par exemple : $O(n)$, $O(n^2)$, $O(n \ln n)$, ...

Quel est le coût de la fonction `variance` en fonction de la longueur de la liste `tab` ? N'y a-t-il pas moyen de faire plus rapide ? (2 points)

```
def moyenne (tab) :
    s = 0.0
    for x in tab :
        s += x
    return s / len (tab)

def variance (tab) :
    s = 0.0
    for x in tab :
        t  = x - moyenne (tab)
        s += t * t
    return s / len (tab)

l = [ 0,1,2, 2,3,1,3,0]
print moyenne (l)
print variance (l)
```

Correction

Tout d'abord, le coût d'un algorithme est très souvent exprimé comme un multiple de la dimension des données qu'il traite. Ici, la dimension est la taille du tableau `tab`. Par exemple, si on note `n` = `len`(`tab`), alors le coût de la fonction `moyenne` s'écrit $O(n)$ car cette fonction fait la somme des n éléments du tableau.

La fonction `variance` contient quant à elle un petit piège. Si elle contient elle aussi une boucle, chacun des n passages dans cette boucle fait appel à la fonction `moyenne`. Le coût de la fonction `variance` est donc $O(n^2)$.

Il est possible d'accélérer le programme car la fonction `moyenne` retourne le même résultat à chaque passage dans la boucle. Il suffit de mémoriser son résultat dans une variable avant d'entrer dans la boucle comme suit :

```
def variance (tab) :
    s = 0.0
    m = moyenne (tab)
```

```
    for x in tab :
        t  = x - m
        s += t * t
    return s / len (tab)
```

Le coût de la fonction `variance` est alors $O(n)$.

Remarque 0.1 : coût d'un algorithme
Le coût d'un algorithme peut être évalué de manière plus précise et nécessiter un résultat comme $n^2 + 3n + 2$ mais cette exigence est rarement utile pour des langages comme *Python*. L'expression `for x in tab :` cache nécessairement un test qu'il faudrait prendre en compte si plus de précision était exigée. Il faudrait également se tourner vers un autre langage de programmation, plus précis dans sa syntaxe. Par exemple, lorsqu'on conçoit un programme avec le langage C ou C++, à partir du même code informatique, on peut construire deux programmes exécutables. Le premier (ou version *debug*), lent, sert à la mise au point : il inclut des tests supplémentaires permettant de vérifier à chaque étape qu'il n'y a pas eu d'erreur (une division par zéro par exemple). Lorsqu'on est sûr que le programme marche, on construit la seconde version (ou *release*), plus rapide, dont ont été ôtés tous ces tests de conception devenus inutiles.

Python aboutit à un programme lent qui inclut une quantité de tests invisibles pour celui qui programme mais qui détecte les erreurs plus vite et favorise une conception rapide. Il n'est pas adapté au traitement d'information en grand nombre et fait une multitude d'opérations cachées.

fin exo 0.0.9 □

0.0.10 Héritage **

Enoncé

On a besoin dans un programme de créer une classe `carre` et une classe `rectangle`. Mais on ne sait pas quelle classe doit hériter de l'autre. Dans le premier programme, `rectangle` hérite de `carre`.

```
class carre :
    def __init__ (self, a) :
        self.a = a
    def surface (self) :
        return self.a ** 2

class rectangle (carre) :
    def __init__ (self, a,b) :
        carre.__init__(self,a)
        self.b = b
    def surface (self) :
        return self.a * self.b
```

Dans le second programme, c'est la classe `carre` qui hérite de la classe `rectangle`.

```
class rectangle :
    def __init__ (self, a,b) :
        self.a = a
        self.b = b
    def surface (self) :
        return self.a * self.b

class carre (rectangle) :
    def __init__ (self, a) :
        rectangle.__init__ (self, a,a)
    def surface (self) :
        return self.a ** 2
```

1) Dans le second programme, est-il nécessaire de redéfinir la méthode `surface` dans la classe `carre` ? Justifiez. (1 point)

2) Quel est le sens d'héritage qui vous paraît le plus censé, `class rectangle(carre)` ou `class carre(rectangle)` ? Justifiez. (1 point)

3) On désire ajouter la classe `losange`. Est-il plus simple que `rectangle` hérite de la classe `carre` ou l'inverse pour introduire la classe `losange` ? Quel ou quels attributs supplémentaires faut-il introduire dans la classe `losange` ? (1 point)

Correction

1) Le principe de l'héritage est qu'une classe `carre` héritant de la classe `rectangle` hérite de ses attributs et méthodes. L'aire d'un carré est égale à celle d'un rectangle dont les côtés sont égaux, par conséquent, la méthode `surface` de la classe retourne la même valeur que celle de la classe `rectangle`. Il n'est donc pas nécessaire de la redéfinir.

2) D'après la réponse de la première question, il paraît plus logique de considérer que `carre` hérite de `rectangle`.

3) Un losange est défini par un côté et un angle ou un côté et la longueur d'une de ses diagonales, soit dans les deux cas, deux paramètres. Dans la première question, il paraissait plus logique que la classe la plus spécifique hérite de la classe la plus générale afin de bénéficier de ses méthodes. Pour introduire le losange, il paraît plus logique de partir du plus spécifique pour aller au plus général afin que chaque classe ne contienne que les informations qui lui sont nécessaires.

```
class carre :
    def __init__ (self, a) :
        self.a = a
    def surface (self) :
        return self.a ** 2

class rectangle (carre) :
    def __init__ (self, a,b) :
        carre.__init__(self,a)
        self.b = b
    def surface (self) :
        return self.a * self.b

class losange (carre) :
    def __init__ (self, a,theta) :
        carre.__init__(self,a)
        self.theta = theta
    def surface (self) :
        return self.a * math.cos (self.theta) * self.a * math.sin (self.theta) * 2
```

Le sens de l'héritage dépend de vos besoins. Si l'héritage porte principalement sur les méthodes, il est préférable de partir du plus général pour aller au plus spécifique. La première classe sert d'interface pour toutes ses filles. Si l'héritage porte principalement sur les attributs, il est préférable de partir du plus spécifique au plus général. Dans le cas général, il n'y a pas d'héritage plus sensé qu'un autre mais pour un problème donné, il y a souvent un héritage plus sensé qu'un autre.

fin exo 0.0.10 □

0.0.11 Précision des calculs ***

Enoncé

1) On exécute le programme suivant :

```
x = 1.0
for i in range (0,15) :
    x = x / 10
    print i, "\t", 1.0 - x, "\t", x, "\t", x **(0.5)
```

Il affiche à l'écran le résultat suivant :

```
0    0.90000000000000002220     0.1              0.316227766017
1    0.98999999999999999112     0.01             0.1
2    0.99899999999999999911     0.001            0.0316227766017
3    0.99990000000000001101     0.0001           0.01
4    0.99999000000000004551     1e-05            0.00316227766017
5    0.99999899999999997124     1e-06            0.001
6    0.99999990000000005264     1e-07            0.000316227766017
7    0.99999998999999994975     1e-08            0.0001
8    0.99999999900000002828     1e-09            3.16227766017e-05
9    0.99999999989999999173     1e-10            1e-05
10   0.99999999998999999917     1e-11            3.16227766017e-06
11   0.99999999999900002212     1e-12            1e-06
12   0.99999999999989996891     1e-13            3.16227766017e-07
13   0.99999999999999000799     1e-14            1e-07
14   0.99999999999999900080     1e-15            3.16227766017e-08
15   0.99999999999999988898     1e-16            1e-08
16   1.00000000000000000000     1e-17            3.16227766017e-09
17   1.00000000000000000000     1e-18            1e-09
18   1.00000000000000000000     1e-19            3.16227766017e-10
19   1.00000000000000000000     1e-20            1e-10
```

Que peut-on en déduire ? (1 point)

2) On écrit une classe `matrice_carree_2` qui représente une matrice carrée de dimension 2.

```
class matrice_carree_2 :
    def __init__ (self, a,b,c,d) :
        self.a, self.b, self.c, self.d = a,b,c,d

    def determinant (self) :
        return self.a * self.d - self.b * self.c

m1 = matrice_carree_2 (1.0,1e-6,1e-6,1.0)
m2 = matrice_carree_2 (1.0,1e-9,1e-9,1.0)
print m1.determinant ()
print m2.determinant ()
```

Qu'affichent les deux dernières lignes ? (1 point)

3) On considère la matrice $M = \begin{pmatrix} 1 & 10^{-9} \\ 10^{-9} & 1 \end{pmatrix}$.

On pose $D = \det(M) = 1 - 10^{-18}$ et $T = tr(M) = 2$. Δ est le déterminant de M et T sa trace. On sait que les valeurs propres de M notées λ_1, λ_2 vérifient :

$$\begin{array}{rcl} D &=& \lambda_1 \lambda_2 \\ T &=& \lambda_1 + \lambda_2 \end{array}$$

On vérifie que $(x - \lambda_1)(x - \lambda_2) = x^2 - x(\lambda_1 + \lambda_2) + \lambda_1\lambda_2$. Les valeurs propres de M sont donc solutions de l'équation : $x^2 - Tx + D = 0$.

Le discriminant de ce polynôme est $\Delta = T^2 - 4D$. On peut donc exprimer les valeurs propres de la matrice M par :

$$\begin{array}{rcl} \lambda_1 &=& \frac{T - \sqrt{\Delta}}{2} \\ \lambda_2 &=& \frac{T + \sqrt{\Delta}}{2} \end{array}$$

On ajoute donc la méthode suivante à la classe `matrice_carree_2` :

```
        def valeurs_propres (self) :
            det   = self.determinant ()
            trace = self.a + self.d
            delta = trace ** 2 - 4 * det
            l1    = 0.5 * (trace - (delta ** (0.5)) )
            l2    = 0.5 * (trace + (delta ** (0.5)) )
            return l1,l2
```

D'après la précédente question, que retourne cette méthode pour la matrice M ? (à justifier) (1 point)

4) On décompose la matrice $M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 10^{-9} \\ 10^{-9} & 0 \end{pmatrix} = I + M'$.

On peut démontrer que si λ est une valeur propre de M', alors $1+\lambda$ est une valeur propre de M. Que donne le calcul des valeurs propres de M' si on utilise la méthode `valeurs_propres` pour ces deux matrices ? (1 point)

5) On considère maintenant la matrice $M'' = \begin{pmatrix} 1 & 10^{-9} \\ -10^{-9} & 1 \end{pmatrix}$. En décomposant la matrice M'' de la même manière qu'à la question 4, quelles sont les valeurs propres retournées par le programme pour la matrice M'' ? Quelles sont ses vraies valeurs propres ? (1 point)

Correction

L'exercice a pour but de montrer que l'ordinateur ne fait que des calculs approchés et que la précision du résultat dépend de la méthode numérique employée.

1) Le programme montre que l'ordinateur affiche `1` lorsqu'il calcule $1 - 10^{-17}$. Cela signifie que la précision des calculs en *Python* est au mieux de 10^{-16}.

2) Il s'agit ici de donner le résultat que calcule l'ordinateur et non le résultat théorique qui sera toujours exact. On cherche à calculer ici le déterminant des matrices M_1 et M_2 définies par :

$$M_1 = \begin{pmatrix} 1 & 10^{-6} \\ 10^{-6} & 1 \end{pmatrix} \text{ et } M_2 = \begin{pmatrix} 1 & 10^{-9} \\ 10^{-9} & 1 \end{pmatrix}$$

Or, le programme proposé calcule les déterminants comme suit :

$$\det M_1 = 1 - 10^{-12} \text{ et } \det M_2 = 1 - 10^{-18}$$

D'après les affichages du programme de la question 1, le programme de la question 2 donnera comme réponse :

```
0.99999999999900002212
1.00000000000000000000
```

La seconde valeur est donc fausse.

3) Le déterminant est utilisé pour calculer les valeurs propres d'une matrice. Cette question s'intéresse à la répercussion de l'approximation faite pour le déterminant sur les valeurs propres. D'après l'énoncé, les deux valeurs propres sont calculées comme étant :

```
l1    = 0.5 * (trace - ((trace ** 2 - 4 * det) ** (0.5)) )
l2    = 0.5 * (trace + ((trace ** 2 - 4 * det) ** (0.5)) )
```

Pour la matrice M_1, on obtient donc en remplaçant le déterminant par 0.99999999999900002212, on obtient les réponses données par le petit programme de la question 1 :

```
l1 = 1,000001
l2 = 0.99999899999999997124  # égale à 1 - 1e-6
```

Pour la matrice M_2, le déterminant vaut 1. En remplaçant `trace` par 2 et `det` par 1, on obtient :

```
l1 = 1
l2 = 1
```

4) On change la méthode de calcul pour la matrice M_2, on écrit que $M_2 = I + \begin{pmatrix} 0 & 10^{-9} \\ 10^{-9} & 0 \end{pmatrix} = I + M_2'$.

Cette fois-ci le déterminant calculé par *Python* est bien $\texttt{1e} - \texttt{18}$. La trace de la matrice M_2' est nulle, on applique les formules suivantes à la matrice M_2' pour trouver :

```
l1    = 0.5 * (trace - ((trace ** 2 - 4 * det) ** (0.5)) ) = - det ** 0.5 = -1e-9
l2    = 0.5 * (trace + ((trace ** 2 - 4 * det) ** (0.5)) ) = det ** 0.5 = 1e-9
```

D'après l'énoncé, les valeurs propres de la matrice M_2 sont les sommes de celles de la matrice I et de la matrice M_2'. Par conséquent, ce second calcul mène au résultat suivant :

```
l1    = 1-1e-9  = 0.99999999900000002828
l2    = 1+ 1e-9 = 1.000000001
```

5) La matrice M'' n'est en fait pas diagonalisable, c'est-à-dire que $tr\,(M'')^2 - 4 * \det M'' = 4 - 4(1+10^{-18}) < 0$. Or le calcul proposé par la question 3 aboutit au même résultat faux que pour la matrice M_2, les deux valeurs propres trouvées seront égales à 1. Si on applique la décomposition de la question 4 :

$$M'' = I + \begin{pmatrix} 0 & -10^{-9} \\ 10^{-9} & 0 \end{pmatrix} = I + N''$$

Le programme calcule sans erreur le discriminant négatif de la matrice N'' qui n'est pas diagonalisable. Il est donc impossible d'obtenir des valeurs propres réelles pour la matrice M'' avec cette seconde méthode.

Cette question montre qu'une erreur d'approximation peut rendre une matrice diagonalisable alors qu'elle ne l'est pas. Il est possible d'accroître la précision des calculs mais il faut faire appel à des modules externes [1].

fin exo 0.0.11 □

1. comme le module GMPy, http://code.google.com/p/gmpy/

Index

Table des matières