Annexes I du livre « Comprendre et mieux connaître les nombres »

Philippe Moutou

paru chez Ellipses (février 2018)

1 Algorithmes et programmes

Entiers/Diviseurs

1) Obtention de la liste des nombres premiers (1)
2) Obtention de la liste des nombres premiers (2)
3) Décomposition d'un nombre en facteurs premiers
4) Obtention du PGCD
5) Obtention des nombres premiers avec *n* et inférieurs à *n*
6) Détermination de la suite aliquote d'un nombre *n*
7) Détermination des antécédents aliquotes de *n*

Décimaux/Bases

8) Conversion d'un nombre *n* en base *b*
9) Détermination du quotient de *a* par *b* en base *c*
10) Détermination d'un nombre par dichotomie

Rationnels/Fractions

11) Détermination du rang d'un rationnel
12) Détermination du numérateur et du dénominateur
13) Détermination du numérateur et du dénominateur
14) Fraction continue d'un nombre rationnel

Réels/Approximations

15) Fraction continue d'un nombre quadratique
16) Meilleures approximations rationnelles d'un réel
17) Approximation de ln(a) par la méthode des rectangles
18) Normalité du nombre de Champernowne

Extensions/Complexes

19) Nombres de $\mathbb{N}(\sqrt{2})$
20) Nombres premiers de $\mathbb{N}(\sqrt{2})$
21) Résolution des équations du 3ème degré
22) Tracer une image de l'ensemble de Mandelbrot
23) Produits et quotients de deux quaternions
24) Rotation dans l'espace avec les quaternions

Obtention de la liste des nombres premiers (1)

Appliquons l'algorithme simple qui a été décrit dans le texte : on examine le reste de la division euclidienne de tous les nombres de la forme $6k-1$ et $6k+1$ par tous les nombres premiers inférieurs et pour toutes les valeurs de $k>0$ jusqu'à atteindre n. Pour initialiser la liste des nombres premiers, on y met 2 et 3 car ce sont les seuls nombres premiers qui ne sont pas de cette forme. Pour ne pas recopier deux fois la boucle *for* qui examine les restes dans le programme, nous avons mis cette boucle dans une fonction *test*() qui est appelée deux fois pour chaque valeur de k.

Cet algorithme rudimentaire donne de bons résultats jusqu'à des nombres pas trop grands : sur mon ordinateur personnel, il faut 0,1 *s* pour obtenir les 1229 premiers inférieurs à 10 000 et 6,8 *s* pour les 9592 premiers inférieurs à 100 000.

```
def test(nb):
    for p in Liste_premiers:
        if nb%p==0 :
            return False
    return True

n=int(input("Saisissez un nombre : "))
Liste_premiers=[2,3]
k=1
notEnd=True
while 6*k-1<=n :
    if test(6*k-1)==True :
        Liste_premiers.append(6*k-1)
    if 6*k+1<=n and test(6*k+1)==True :
        Liste_premiers.append(6*k+1)
    k+=1
print("Plus grand premier="+str(Liste_premiers[-1]))
print("Nombre de premiers="+str(len(Liste_premiers)))
print("Liste des premiers: ",Liste_premiers)
```

```
Saisissez un nombre : 100
Plus grand premier=97
Nombre de premiers=25
Liste des premiers:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Obtention de la liste des nombres premiers (2)

L'algorithme d'Ératosthène apporte un gain d'efficacité d'autant plus important que *n* est grand. On commence donc par initialiser une liste avec tous les nombres entiers de 2 à *n*. Ensuite on expurge de cette liste tous les multiples du premier nombre qui est premier puisqu'il s'agit de 2. Pour réaliser cette expurgation, Python possède une instruction très pratique qui recopie une liste selon une certaine condition : au début, la condition est de ne pas être multiple de 2 (*p*%2!=0 signifie le reste de la division de *p* par 2 n'est pas 0) ; par la suite, on enlèvera tous les multiples de 3, puis de 5, etc. Une fois les multiples d'un nombre premier supprimés, on change la valeur du dernier nombre premier trouvé en prenant celui qui suit le précédent dans la liste (Liste_premiers.index(*k*)+1 est l'indice qui suit celui du nombre premier *k* dans la liste). On recommence alors l'expurgation en enlevant tous les multiples de ce nouveau nombre premier, etc. jusqu'à atteindre la plus grande valeur que peut prendre *k* : il s'agit du plus grand nombre premier inférieur à $\sqrt{n}$ (noté *n***0.5 en Python) comme on l'a expliqué dans le texte.

```
n=int(input("Saisissez un nombre : "))
Liste_premiers=list(range(2,n+1))
k=2
nRacine=n**0.5
while k<nRacine :
    Liste_premiers=[p for p in Liste_premiers if p<=k or p%k!=0]
    k=Liste_premiers[Liste_premiers.index(k)+1] # nouveau nombre premier
print("Plus grand premier="+str(Liste_premiers[-1]))
print("Nombre de premiers="+str(len(Liste_premiers)))
print("Liste des premiers: ",Liste premiers)
```

Avec cet algorithme, vieux de deux millénaires, la recherche est accélérée puisqu'il ne faut plus que 0,3 *s* pour trouver les 9592 nombres premiers inférieurs à 100 000 et 6,0 *s* pour trouver les 78 498 nombres premiers inférieurs à 1 000 000 (près de 8 *min* avec l'algorithme précédent).

Décomposition d'un nombre en facteurs premiers

Pour obtenir les facteurs premiers et leur multiplicité, nous allons commencer par établir une liste de nombres premiers. Nous allons donc réutiliser le programme précédent au moyen de l'instruction Liste_facteurs=premiers() qui détermine, dans la fonction premier(), la liste des nombres premiers inférieurs à $\sqrt{n}$ puisque seuls ces nombres peuvent être des facteurs premiers de *n*. Voici la fonction premier() :

```
def premiers():
    L,k,nRacine=list(range(2,n+1)),2,n**0.5
    while k<nRacine :
        L=[p for p in L if p<=k or p%k!=0]
        k=L[L.index(k)+1] # nouveau nombre premier
    return L
```

On enlève de cette liste, les nombres premiers qui ne divisent pas *n*, puis on obtient les multiplicités de chacun des facteurs, successivement, par deux boucles *while* imbriquées qui décomposent progressivement *n* selon chacun de ses facteurs.

Pour ceux qui ne sont pas familiers avec Python, l'instruction *n*//=Liste_facteurs[i] affecte dans *n* l'ancienne valeur de *n* divisée par le nombre Liste_facteurs[i], de même que i+=1 affecte à i l'ancienne valeur de i ajoutée à 1. La division utilise le symbole // car on souhaite garder des entiers (6//2 donne 3 alors 6/2 donne 3.0, un nombre flottant).

La fin du programme met en forme l'affichage de la décomposition (2^3×3^1×5^1 au lieu de [[2, 3], [3, 1], [5, 1]]).

```
n=int(input("Saisissez un nombre : "))
Liste_facteurs,decompo,i,s=premiers(),list(),0,str(n)+"="
Liste_facteurs=[p for p in Liste_facteurs if n%p==0]#expurgation premiers non-facteurs
while n>1 :
    expo=0  #recherche de l'exposant pour chaque facteur premier
    while n%Liste_facteurs[i]==0 :
        expo+=1
        n//=Liste_facteurs[i]
    decompo.append([Liste_facteurs[i],expo])
    i+=1
for rang,facteur in enumerate(decompo) :
        s+=str(decompo[rang][0])+"^"+str(decompo[rang][1])
        if rang<len(decompo)-1 : s+='\u00D7'
print(s)
```

```
Saisissez un nombre : 120
120=2^3×3^1×5^1
```

```
Saisissez un nombre : 75600
75600=2^4×3^3×5^2×7^1
```

```
Saisissez un nombre : 9091
9091=9091^1
```

Obtention du PGCD

Le mode itératif est le mode que l'on utilise au début : il utilise principalement des boucles *for* et/ou des boucles *while*. La fonction *pgcd1*() que nous avons écrit est une traduction de l'algorithme d'Euclide dans ce mode : on effectue des divisions euclidiennes sans se préoccuper du quotient, en utilisant juste le reste (*reste=a%b*). Dans la boucle *while*, on remplace (*a,b*) par (*b,reste*) jusqu'à ce que le reste soit nul. Le PGCD cherché est *b*, le dernier diviseur.

La récursivité simplifie souvent l'écriture d'une fonction, surtout quand celle-ci est, par nature, récursive et c'est le cas de l'algorithme d'Euclide qui utilise la propriété :

PGCD(*a,b*)=PGCD(*b,reste*) tant que *reste*≠0.

La fonction *pgcd2*(), qui donne la traduction en mode récursif de cet algorithme, utilise cette propriété et un test d'arrêt qui examine si le reste est nul. Ainsi *pgcd2*(12,8) calcule 12%8=4 et comme 4≠0 retourne *pgcd2*(8,4) qui calcule 8%4=0 et comme 0=0 retourne 4 qui se substitue à *pgcd2*(8,4) puis à *pgcd2*(12,8) comme valeur de retour.

```
def pgcd1(a,b): # forme itérative
    reste=a%b
    while reste!=0 :
        a,b=b,reste
        reste=a%b
    return b

def pgcd2(a,b): # forme récursive
    reste=a%b
    if reste==0 : return b
    else : return pgcd2(b,reste)

A=int(input('PGCD de A : '))
B=int(input('et B : '))
pgcd=pgcd1(A,B)#remplacer 1 par 2 pour utiliser la forme récursive
print('PGCD({},{})={}'.format(A,B,pgcd))
```

```
PGCD de A : 1000000001
et de B : 10000001
PGCD(1000000001,10000001)=11
```

```
PGCD de A : 654654
et de B : 1001
PGCD(654654,1001)=1001
```

Obtention des nombres premiers avec *n* et inférieurs à *n*

Pour obtenir cette liste L1, on peut déterminer, à l'aide du programme précédent, le PGCD de tous les nombres inférieurs à *n* avec *n* et afficher ces nombres lorsque le PGCD obtenu vaut 1. Mais on peut aussi, plus simplement, chercher la liste des facteurs premiers de *n* et faire la liste de tous les nombres qui n'admettent aucun de ces facteurs comme diviseur. Pour cela, on va utiliser la fonction *premiers*() définie 2 pages plus haut.

Pour compléter cette étude, on peut faire la liste L2 des diviseurs de *n* et, enfin, la liste L3 des autres nombres inférieurs (ceux qui ne sont ni des diviseurs de *n* ni des nombres premiers avec *n*). Voir dans le texte le type de figure obtenu lorsqu'on joint les sommets d'un polygone convexe à *n* côtés, en prenant les sommets de *k* en *k*, selon que *k* appartienne à L1, L2 ou L3. Pour obtenir L2, on va, dans un 1er temps, retrancher des nombres inférieurs à *n* ceux qui sont dans L1. Ensuite, on crée l'ensemble L3 en prenant dans L2 ceux qui divise *n*. Enfin, on retire de L2 les nombres de L3. Remarquez comme cela est simple à effectuer avec Python.

```
n=int(input("Saisissez un nombre : "))
Liste_facteurs=premiers()
Liste_facteurs=[p for p in Liste_facteurs if n%p==0]
L1=list(range(1,n+1))
for k in Liste_facteurs : L1=[p for p in L1 if p%k!=0]
L2=[p for p in list(range(2,n+1)) if p not in L1]
L3=[p for p in L2 if n%p!=0]
L2=[p for p in L2 if p not in L3]
print("L1="+str(L1))
print("L2="+str(L2))
print("L3="+str(L3))
```

```
Saisissez un nombre : 21
L1=[1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20]
L2=[3, 7, 21]
L3=[6, 9, 12, 14, 15, 18]
Saisissez un nombre : 17
L1=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
L2=[17]
L3=[]
```

Détermination de la suite aliquote d'un nombre *n*

Dans une fonction *decomposition*(), on détermine les nombres premiers nécessaires pour décomposer en facteurs premiers les termes de la suite. Pour obtenir la somme des diviseurs sans les calculer explicitement, on remarque qu'elle s'obtient en effectuant le produit des sommes de chaque facteur aux différentes multiplicités possibles. Prenons $1176=2^3\times3^1\times7^2$, un de ses diviseurs est $98=2^1\times3^0\times7^2$, un autre est $12=2^2\times3^0\times7^0$. La somme des diviseurs de 1176 est le produit $(2^0+2^1+2^2+2^3)\times(3^0+3^1)\times(7^0+7^1+7^2)=3420$ (développez cette expression pour retrouver tous les diviseurs) dont on enlève 1176 pour trouver 2244, le suivant aliquote de 1176.
Toutes ces opérations sont confiées à la fonction *suivant*(). Dans le programme principal, on crée une boucle qui s'arrête lorsqu'on tombe sur 1, sur un cycle (nombre parfait, ou cycle de nombres amicaux) ou sur un nombre trop grand (seuil).

```
def suivant(nb) :#détermine la somme des diviseurs stricts de nb
    global Lprem
    decompo=decomposition(nb)
    prod=1
    for facteur in decompo :
        expo,som=0,0
        while expo<=facteur[1] :
            som+=facteur[0]**expo
            expo+=1
        prod*=som
    return prod-nb
```

```
n=int(input("De quel nombre voulez-vous partir : "))
suite,seuil=[],1000000000
seuil=1000000000
while n<seuil and n not in suite and n!=1 :
    suite.append(n)
    n=suivant(n)
suite.append(n)
if n==1 : print(suite)
elif n>=seuil :
    print("suite indéterminée de longueur {}.".format(len(suite)),suite)
else :
    rang=0
    while suite[rang]!=suite[-1] : rang+=1
    print("cycle de longueur {} : ".format(len(suite)-rang-1),suite)
```

Détermination des antécédents aliquotes de *n*

Nous allons confier la tâche de déterminer l'antécédent aliquote de *n* à la fonction *antecedent*(). On y balaie tout l'éventail des candidats possibles entre 1 et $(n-1)^2$ en comparant le successeur aliquote du candidat à *n*. La fonction renvoie le nombre d'antécédents trouvés.
Dans le programme principal, on relance la recherche si il n'y a qu'un antécédent, afin de remonter la suite aliquote. Dans les cas où il y a plusieurs antécédents possibles ou bien aucun, on arrête la recherche en affichant les résultats.

```
def antecedant(nb) :#détermine le ou les antécédents aliquotes de nb
    global suite
    k,q,limite=1,0,(nb-1)**2
    while k<=limite :
        if suivant(k)==nb :
            suite.append(k)
            print("la chaîne continue à partir de {} avec l'antécédent {}".format(nb,k))
            q+=1
        k+=1
    return q
```

```
debut=int(input("De quel nombre voulez-vous partir : "))
suite=[debut]
notEnd=True
while notEnd:
    debut=suite[-1]
    nombre=antecedant(debut)
    if nombre==0 :
        notEnd=False
        print("la chaîne se termine sur {} qui est intouchable.".format(debut))
    elif nombre>1 :
        k,notEnd=1,False
        parents=' et '+str(suite[-k])
        while k<nombre-1 :
            k+=1
            parents=', '+str(suite[-k])+parents
        parents=str(suite[-nombre])+parents
        print("la chaîne bifurque à partir de {} dont les antécédents sont {}"
            .format(debut,parents))
    else :
        print("la chaîne continue à partir de {} avec l'antécédent unique {}"
            .format(debut,suite[-1]))
```

```
De quel nombre voulez-vous partir : 86
la chaîne continue à partir de 86 avec l'antécédent 166
la chaîne continue à partir de 86 avec l'antécédent unique 166
la chaîne continue à partir de 166 avec l'antécédent 212
la chaîne continue à partir de 166 avec l'antécédent 326
la chaîne bifurque à partir de 166 dont les antécédents sont 212 et 326
```

Conversion d'un nombre *n* en base *b*

Limitons nous aux bases de 2 à 36 pour utiliser, dans le cas où *b*=36, les dix chiffres de la base 10 et les 26 lettres de l'alphabet minuscule (on aurait pu ajouter les 26 lettres majuscules pour aller jusqu'à la base 60).

Pour effectuer cette conversion, nous utilisons la fonction *conversionBase*() qui va, tout simplement, opérer une suite de divisions euclidiennes qui ont toutes la base *b* comme diviseur et qui prennent, comme dividendes, le nombre *n* puis les quotients successifs, jusqu'à ce que ce dividende soit trop petit pour être divisé par la base. Le dernier quotient est alors 0 et les chiffres de l'écriture de *n* en base *b* sont les différents restes, dans l'ordre inverse où ils ont été obtenus. Il suffit donc de concaténer les restes dans l'ordre inverse en utilisant le tableau des chiffres (par exemple chiffre[10] est « a »). C'est ce que réalise l'instruction ecriture=chiffre[reste[j]]+ecriture qui, notez-le bien, accole le chiffre suivant *devant* le précédent.

Exemple : $2016=168\times12+0$ (le 1er reste est 0) ; $168=14\times12+0$ (le 2ème reste est 0) ; $14=1\times12+2$ (le 3ème reste est 2) ; $1=0\times12+1$ (le 4ème reste est 1). Le nombre s'écrit alors 1200 en base 12.

Vérification : $2016=1\times12^3+2\times12^2+0\times12^1+0\times12^0=1728+288$.

```
def conversionBase():
    reste,quotient,num,ecriture=list(),nb,nb,""
    while quotient>0:
        quotient=num//base
        reste.append(num%base)
        num=quotient
    for j in range(len(reste)): ecriture=chiffre[reste[j]]+ecriture
    return ecriture
```

```
chiffre=["0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f","g","h",\
         "i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]
nb=int(input("Entrer le nombre (en base 10) : "))
base=int(input("Entrer la base (en base 10) : "))
print("Le voici écrit dans cette base : {}".format(conversionBase()))
```

```
Entrer le nombre (en base 10) : 255
Entrer la base (en base 10) : 16
Le voici écrit dans cette base : ff
```

```
Entrer le nombre (en base 10) : 2016
Entrer la base (en base 10) : 12
Le voici écrit dans cette base : 1200
```

Détermination du quotient de *a* par *b* en base *c*

Avec les mêmes limitations que pour le programme précédent dont nous reprenons la fonction *conversionBase*(), nous allons effectuer la suite des divisions successives de la partie décimale par la base et enregistrer les chiffres du quotient ainsi que les restes. À chaque étape nous examinons la suite des restes déjà obtenus pour déterminer si le nouveau reste est original (les divisions continuent, sauf si ce reste est nul) ou bien s'il s'est déjà présenté (le développement décimal est périodique).

Dans le cas d'un développement périodique, on extrait de la séquence des quotients la partie qui se répète (l'instruction ecriture[-longueur:] réalise cela, la notation de l'indice -longueur: négatif et suivi du symbole « : » indiquant que l'on veut extraire de la chaîne de caractères ecriture une sous-chaîne de longueur longueur en partant de la fin).

```
chiffre=["0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f","g","h",\
         "i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]
a=int(input("Entrer le dividende (base 10) : "))
b=int(input("Entrer le diviseur (base 10) : "))
c=int(input("Entrer la base de sortie (<36) : "))
if a//b>0 : ecriture=conversionBase(a//b,c)  #la partie entière du quotient en base c
else : ecriture="0"
if a%b==0: print("Votre nombre est entier, le voici dans cette base : "+ecriture)
else :
    reste,longueur,suiteRestes,fin=a%b,0,list(),0
    ecriture+=",";
    suiteRestes.append(reste)
    while fin==0:
        reste*=c
        chiffreQuotient=reste//b
        suiteRestes.append(reste%b)
        ecriture+=chiffre[chiffreQuotient]
        if suiteRestes[-1]==0 : fin=1
        else:
            for i in range(len(suiteRestes)-1) :
                if suiteRestes[-1]==suiteRestes[i] :
                    fin=2
                    longueur=len(suiteRestes)-i-1
                    sequence=str(ecriture[-longueur:])
                    ecriture+=sequence+"..."
                else : reste=suiteRestes[-1]
    print("Voici le développement illimité du quotient : "+ecriture)
    if fin==2 : print("Votre nombre a une écriture décimale illimitée périodique\
 de période "+str(longueur)+", la séquence qui se répète est "+sequence)
    else : print("Votre nombre a une écriture finie en base "+str(c)+" : "+ecriture)
```

Détermination d'un nombre par dichotomie

L'algorithme de la dichotomie a été expliqué dans le texte (chapitre 2 – approximations décimales). Nous avons défini un sous-programme, la fonction *image*(), pour déterminer l'image d'un nombre *x* par une fonction particulière. Sur l'exemple donné, la fonction est une fonction polynôme du 5ème degré : $x \mapsto x^5+3x^2+2x-1$.

La question est de déterminer un antécédent de 0, c'est-à-dire de résoudre l'équation $x^5+3x^2+2x-1=0$. Il est toujours possible de se ramener à ce problème : si on veut résoudre l'équation $x^5+3x^2+2x-1=-1$, il suffit de transformer l'égalité pour faire apparaître un 0 à droite : $x^5+3x^2+2x=0$ et changer l'expression de la fonction en conséquence. On peut se servir de la courbe pour localiser la (ou les) solution(s). Sur la courbe de la fonction précédente on voit que l'équation $x^5+3x^2+2x-1=-1$ a trois solutions : 0, -1 et un nombre de]-0,9;-0,5[qui est environ -0.8105357137661486. Si l'intervalle entré ne convient pas (absence de solution ou non monotonie), l'algorithme donne une des bornes, à la précision près. Cela arrive si on entre *a*=-1 et *b*=0 au lieu de *a*=-0,9 et *b*=-0,5…

```
def image(x) :
    return x**5+3*x**2+2*x-1
```

```
a=float(input('Quelle est a, la borne inférieure? '))
b=float(input('Quelle est b, la borne supérieure? '))
c=0
d=image(a)
e=image(b)
p=int(input("Quelle est la valeur absolue de l'exposant de la précision souhaitée? "))
while b-a>10**-p :
    c=(a+b)/2
    f=image(c)
    if d*f<0 :
        b,e=c,f
    else :
        a,d=c,f
print('voici votre solution : {}'.format(c))
```

```
Quelle est a, la borne inférieure? 0
Quelle est b, la borne supérieure? 1
Quelle est la valeur absolue de l'exposant de la précision souhaitée? 15
voici votre solution : 0.332319309167608
```

Détermination du rang d'un rationnel

Le principe de dénombrement des rationnels (Cantor) a été expliqué dans le texte. On fait croître la somme S du numérateur et du dénominateur des fractions irréductibles, en ordonnant pour chaque valeur de S, les fractions par leur numérateur.

L'algorithme est très simple : il contient deux boucles *while*. La 1ère teste la somme S qui ne doit pas dépasser la valeur (connue dès le départ) du nombre rationnel entré. La 2de teste la valeur du numérateur qui ne doit pas dépasser S-1. Il faut déterminer si une fraction est réductible ou non, car seules celles qui ne le sont pas sont comptabilisées pour le rang. Cela est fait par l'appel d'une fonction *pgcd*() qui fonctionne sur le principe de l'algorithme d'Euclide. On incrémente enfin les compteurs comme il convient dans ces deux boucles (au passage on comptabilise aussi les fractions réductibles).

```
n=int(input('Quelle est le numérateur ? '))
d=int(input('Quelle est le dénominateur ? '))
som,tot,rang=1,0,0
while som<=n+d:
    num,denom=0,som
    while num<som:
        if pgcd(num,denom)==1:
            rang+=1
        tot+=1
        num+=1
        denom-=1
        if num==n and denom==d:
            print('Rang de votre fraction={}'.format(rang))
            print('Total des fractions inférieures={}, \
ratio rang/tot ={}%'.format(tot,round(rang/tot*100,2)))
            break
    som+=1
```

```
Quelle est le numérateur ? 1
Quelle est le dénominateur ? 6
Rang de votre fraction=12
Total des fractions inférieures=22, ratio rang/tot =54.55%
```

Détermination du numérateur et du dénominateur

Lorsqu'un nombre rationnel est connu par son écriture décimale illimitée : une partie fixe (*pnd*) et la séquence périodique (*sp*), on veut retrouver le numérateur (*num*) et le dénominateur (*denom*) de l'écriture fractionnaire irréductible.
On commence par trouver la puissance de dix qui permet de ramener la partie fixe au format '*xxyy.*' pour avoir un nombre dont la séquence périodique commence après la virgule. Les instructions Python utilisées appartiennent aux méthodes de la classe *string* (caractère) : *len*(), *.index*() et *replace*()). Ensuite, on fait comme décrit dans le texte, si il y a *N* chiffres qui se répètent, on prend comme *num* et comme *denom* : $10^N \times (pnd + sp) - pnd$ et $10^N - 1$. Les formats en Python étant stricts, il y a des conversions à faire, en respectant les formats : '12.' n'est pas convertible en *float*, mais '12.0' l'est. À la fin, on doit diviser par la puissance de dix trouvée au début et chercher le PGCD(*num*;*denom*) pour simplifier.

```
def composition(pnd,sp):
    # la partie non-périodique (pnd) est au format 'xx.yy'
    # la suite périodique (sp) est au format 'zz'
    P=len(pnd)-(pnd.index('.')+1)
    pnd=pnd.replace('.','')+'.' #pnd est au format 'xxyy.'
    N=len(sp)
    num=int(10**N*float(pnd+sp)-float(pnd+'0'))
    denom=int((10**N-1)*10**P)
    div=pgcd(num,denom)
    return int(num//div),int(denom//div)

n=input('Quelle est la partie non-périodique (ex: 0. ou 12.3) ? ')
p=input('Quelle est la partie périodique (ex: 0, 3 ou 12) ? ')
num,denom=composition(n,p)
print('Numérateur={}, dénominateur={}'.format(num,denom))
```

```
Quelle est la partie non-périodique (ex: 0. ou 12.3) ? 0.12
Quelle est la partie périodique (ex: 0, 3 ou 12) ? 345
Numérateur=4111, dénominateur=33300

Quelle est la partie non-périodique (ex: 0. ou 12.3) ? 4.5
Quelle est la partie périodique (ex: 0, 3 ou 12) ? 0
Numérateur=9, dénominateur=2
```

Recherche de la 1ère séquence périodique de longueur n

Nous réutilisons la structure du programme de la division de *a* par *b* en base *c* qui nécessite la fonction *conversionBase*(). Si on lance en l'état, ce programme donne 31 (le 1er nombre dont l'inverse a une séquence périodique de 30 chiffres en base 12). On peut modifier le test en fin de boucle : au lieu de if longueur==n on écrit if longueur==n or longueur==5 or longueur==10 or longueur==13. Si, de plus, on modifie la longueur n dans les initialisations : n,a,b,c=17,1,1,12, cela va conduire à trouver le 1er nombre dont l'inverse a une séquence périodique de 5, 10, 13 ou 17 chiffres en base 12. Le résultat sera 19 141 dont l'inverse a une séquence de longueur 10 en base 12. On peut relancer la recherche en changeant la valeur initiale de b pour optimiser (b=19 141) et en enlevant le or longueur==10 du test final.

```
chiffre=["0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f","g","h",\
         "i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]
n,a,b,c=30,1,1,12 # >>> à modifier directement si nécessaire
while True:
    if b%1000==0:print("nombre en cours : "+str(b))
    if a//b>0 : ecriture=conversionBase(a//b,c)#partie entière en base c
    else : ecriture="0"
    if a%b==0: longueur=0 # le nombre est entier
    else :
        reste,longueur,suiteRestes,fin=a%b,0,list(),0
        ecriture+=",";
        suiteRestes.append(reste)
        while fin==0:
            reste*=c
            chiffreQuotient=reste//b
            suiteRestes.append(reste%b)
            ecriture+=chiffre[chiffreQuotient]
            if len(suiteRestes)>n+1 : break # pour optimiser la recherche
            if suiteRestes[-1]==0 :fin=1
            else:
                for i in range(len(suiteRestes)-1) :
                    if suiteRestes[-1]==suiteRestes[i] :
                        fin=2
                        longueur=len(suiteRestes)-i-1
                        sequence=str(ecriture[-longueur:])
                        ecriture+=sequence+"..."
                    else : reste=suiteRestes[-1]
    if longueur==n: break # >>> à compléter directement si nécessaire
    b+=1
print("Longueur séquence : "+str(longueur))
print("Voici le nombre cherché en base dix : "+str(b))
print("Le voici écrit dans votre base {} : {}".format(c,conversionBase(b,c)))
print("Voici le développement illimité du quotient : "+ecriture)
print("La séquence qui se répète est "+sequence)
```

Fraction continue d'un nombre rationnel

On entre le numérateur et le dénominateur *a* et *b* d'une fraction et la décomposition est confiée à la fonction *fracContinue*() qui renvoie une liste constituée des coefficients de la fraction continue *a/b*. Le 1er coefficient est la partie entière, les autres sont les quotients successifs de l'algorithme d'Euclide. L'affichage d'une liste en Python utilisant la convention de notation des fractions continues adoptée dans le texte, il n'y a rien à faire sur ce point.

```
def fracContinue(a,b):
    quotient=a//b
    reste=a%b
    lst=[quotient]
    if reste==0 : return lst # cas a/b entier
    while reste!=0 :
        a,b=b,reste
        quotient,reste=a//b,a%b
        lst.append(quotient)
    return lst

a=int(input("Numérateur = "))
b=int(input("Dénominateur = "))
print("Fraction continue finie :{}.".format(fracContinue(a,b)))
```

```
Numérateur = 50
Dénominateur = 34
Fraction continue finie :[1, 2, 8].

Numérateur = 55
Dénominateur = 34
Fraction continue finie :[1, 1, 1, 1, 1, 1, 1, 2].
```

Pour déterminer, à numérateur N égal, le dénominateur D<N de la fraction continue la plus longue :

```
N=int(input('Quel est le numérateur ? '))
D,Dmax,Lmax,LstMax=1,0,0,()
while D<N:
    liste=fracContinue(N,D)
    if len(liste)>Lmax:Dmax,Lmax,LstMax=D,len(liste),liste
    D+=1
print('Si numérateur={}, dénominateur de la +longue FC={}'.format(N,Dmax))
print('Cette FC est {}'.format(LstMax))
```

```
Quel est le numérateur ? 55
La fraction de numérateur 55 ayant la plus longue FC a pour dénominateur 34
Cette FC est [1, 1, 1, 1, 1, 1, 1, 2]
```

Fraction continue d'un nombre quadratique

On entre trois entiers *a*, *b* et *c*. Si *b*>0 et *c*≠0, le nombre $\frac{a+\sqrt{b}}{c}$ est un nombre quadratique. En supposant que l'écriture ne soit pas simplifiable, lorsque *b* n'est pas un carré, ce nombre est irrationnel et sa fraction continue est périodique. Le programme détecte la périodicité en comparant des valeurs approchées des parties décimales résiduelles (les nombres *y*[i]). Cela donne des résultats satisfaisants mais, dans certains cas, cela peut entraîner des erreurs. De meilleurs algorithmes peuvent être écrits qui n'ont pas ce défaut (travailler avec des triplets d'entiers au lieu de valeurs approchées décimales). Par exemple $\frac{2+\sqrt{19}}{7}$ =[0, 1, $\overline{9, 1, 11, 3, 3, 3, 1, 3, 3, 3, 11, 1, 9, 3, 1, 29, 1, 3}$] alors que notre algorithme se trompe à partir du 16ème coefficient (il donne [0, 1, 9, 1, 11, 3, 3, 3, 1, 3, 3, 3, 11, 1, 9, 2, 1, ...]).

```
from math import sqrt
def fracContinue(a,b):# méthode pour une fraction continue rationnelle a/b
    quotient=a//b
    reste=a%b
    lst=[quotient]
    if reste==0 : return lst # cas a/b entier
    while reste!=0 :
        a,b=b,reste
        quotient,reste=a//b,a%b
        lst.append(quotient)
    return lst
# Programme principal
print("Entrer un nombre quadratique (a+sqrt(b))/c:")
a=int(input("a = "))
b=int(input("b = "))
c=int(input("c = "))
if int(sqrt(b))==sqrt(b):
    print("Fraction continue finie :{}.".format(fracContinue(int(a+sqrt(b)),c)))
else:
    x=(a+sqrt(b))/c     # le nombre quadratique(valeur approchée décimale)
    p=False             # périodique l'écriture? par défaut non
    q=[int(x)]          # la suite des quotients partiels de la fraction continue
    y=[x-q[0]]          # la suite des parties décimales résiduelles
    i=0                 # nombre d'étapes
    while p==False :
        q.append(int(1/y[i]))
        y.append(1/y[i]-q[i+1])
        for k in range(i+1):
            if q[i+1]==q[k] and int(1000000*y[i+1])==int(1000000*y[k]):
                p,r=True,i+1-k
                del(q[i+1])
                break
        i+=1
    print("Fraction continue périodique :{}\nLongueur de la période : {}".format(q,r))
```

Meilleures approximations rationnelles d'un réel

Cet algorithme de recherche est assez rudimentaire mais son objectif est simple et pratique : étant donné un nombre réel *nb* connu par ses premières décimales et une fraction $\frac{a}{b}$ qui s'en approche, quelle est la première fraction qui s'écrit avec des nombres plus petits et qui s'en approche mieux ? L'image montre que si $\frac{9864101}{3628800}$ est une fraction qui s'approche assez bien de *e*, la première fraction qui s'en approche mieux est $\frac{20504}{7543}$. L'affichage de *q* (le dénominateur testé) toutes les 100 valeurs sert à indiquer l'avancement de la recherche.

Si on veut modifier le nombre réel, il faut juste modifier les trois nombres des deux premières lignes (*a*, *b* et *nb*). En entrant, par exemple 314,100,3.141592653589793 (soit $\frac{314}{100}$ comme une valeur approchée de π), on obtient sans aucun délai la valeur $\frac{22}{7}$.

```
a,b=9864101,3628800 # mettre ici le numérateur et dénominateur de la fraction
nb=2.718281828459045235360287471352662497 # mettre ici le nombre réel
p,q,approx=0,1,round(nb,1)
while True:
    if abs(p/q-nb)<abs(a/b-nb):break
    if p/q>nb and p/q>a/b:
        q+=1
        if q%100==0:print('q={}'.format(q))
        p=1
        while p/q<approx: p+=1 # pour diminuer le temps de recherche
    else:p+=1
print("Voici la première meilleure fraction :{}/{} ".format(p,q))
print("L'écart avec le nombre est {}.".format(abs(p/q-nb)))
print("Alors qu'avec {}/{} il est de {}".format(a,b,abs(a/b-nb)))
```

```
q=100       ....
q=200       q=7000
q=300       q=7100
q=400       q=7200
q=500       q=7300
q=600       q=7400
q=700       q=7500
q=800       Voici la première meilleure fraction :20504/7543
q=900       L'écart avec le nombre est 2.2263479060313784e-08.
q=1000      Alors qu'avec 9864101/3628800 il est de 2.7312660577649694e-08
```

Approximation de ln(a) par la méthode des rectangles

On entre deux entiers a>1 et p (pas trop grand) pour avoir une valeur approchée de ln(a) à la précision 10^{-p}. Pour cela, on utilise les sommes S_k et T_k qui encadrent ce nombre. Les subdivisions de l'intervalle [1 ; a] sont au nombre $n=\frac{(a-1)^2\times 10^{-p}}{a}$ d'après ce qu'on a vu dans le texte.

Cette méthode n'est pas très efficace et prend beaucoup de temps quand a est grand (par exemple 100) ainsi que p (par exemple 6). Par contre, à quelques modifications près, on peut obtenir le logarithme de n'importe quel nombre décimal :

- remplacer a=int(input(... par a=float(input(...
- convertir n en entier : n=int(((a-1)**2*10**p)/a)

```
a=int(input("De quel nombre a voulez-vous approcher ln(a)? : "))
p=int(input("Quelle est la précision voulue (p de 10^(-p)) : "))
n=((a-1)**2*10**p)//a
S,T=0,0
for k in range(n):
    S+=(a-1)/(n+k*(a-1)+1)
    T+=(a-1)/(n+k*(a-1))
print("Sn= {},Tn= {},Tn-Sn= {} : ".format(S,T,T-S))
```

```
De quel nombre a voulez-vous approcher ln(a)? : 2
Quelle est la précision voulue (p de 10^(-p)) : 6
Sn= 0.6931466805601864,Tn= 0.6931476805601863,Tn-Sn= 9.999999999177334e-07 :

De quel nombre a voulez-vous approcher ln(a)? : 100
Quelle est la précision voulue (p de 10^(-p)) : 6
Sn= 4.605170675887216,Tn= 4.605170685988217,Tn-Sn= 1.010100092457833e-08 :
```

Un autre algorithme plus efficace pour calculer ln(2) selon la formule des puissances.

```
n=int(input("Combien de termes voulez-vous? : "))
S=0
for k in range(1,n+1) :
    S+=1/((2*k-1)*(3)**(2*k-1))
print("ln(2)=2Sn= {}".format(2*S))
```

```
Combien de termes voulez-vous? : 4
ln(2)=2Sn= 0.6931347573322881
```

Normalité du nombre de Champernowne

Nous ne voulons tester qu'une partie infime de la normalité de C_{10}, le nombre 0,1234567891011... dont les décimales sont les nombres entiers en base dix. Nous calculons les fréquences des quarante suites de décimales « x », « 0x », « 00x » et « 000x » pour tous les chiffres x. Le paramètre « rang » permet de régler la profondeur de l'investigation ; dans l'état, on examine les 10 000 premières décimales. Pour donner une idée de l'affichage sur la console, nous avons pris rang=100.

La seule complication de ce comptage vient de la fonction *count*() de Python qui ne prend en compte que les séquences ne se chevauchant pas. On peut utiliser cette fonction pour les séquences « x » mais pas pour les autres car sinon dans la suite « 0001 » par exemple, on ne compterait qu'une fois la séquence « 00 » alors qu'elle est présente deux fois. On a donc écrit une boucle *while* pour dénombrer correctement les diverses séquences.

```
S1=[["0",0],["1",0],["2",0],["3",0],["4",0],["5",0],["6",0],["7",0],["8",0],["9",0]]
S2=[["00",0],["01",0],["02",0],["03",0],["04",0],["05",0],["06",0],["07",0],["08",0],["09",0]]
S3=[["000",0],["001",0],["002",0],["003",0],["004",0],["005",0],["006",0],["007",0],["008",0],["009",0]]
S4=[["0000",0],["0001",0],["0002",0],["0003",0],["0004",0],["0005",0],["0006",0],["0007",0],["0008",0],["0009",0]]
dec,k,rang="",1,10000 # le rang est à modifier : c'est le nombre de décimales examinées
while len(dec)<rang:
    dec+=str(k)
    k+=1
while len(dec)>rang:dec=dec[:-1]
for i in range(10):
    S1[i][1]=dec.count(S1[i][0])
    j,k=0,0
    while j>=0:
        j=dec.find(S2[i][0],k)
        if j>0:
            k=j+1
            S2[i][1]+=1
    j,k=0,0
    while j>=0:
        j=dec.find(S3[i][0],k)
        if j>=0:
            k=j+1
            S3[i][1]+=1
    j,k=0,0
    while j>=0:
        j=dec.find(S4[i][0],k)
        if j>=0:
            k=j+1
            S4[i][1]+=1
print("dec : {}".format(dec))
print("S1  : {}".format(S1))
print("S2  : {}".format(S2))
print("S3  : {}".format(S3))
print("S4  : {}".format(S4))
```

```
dec : 12345678910111213141516171819202122232425262728
2930313233343536373839404142434445464748495051525354 5
S1  : [['0', 5], ['1', 16], ['2', 16], ['3', 16], ['4', 16],
       ['5', 11], ['6', 5], ['7', 5], ['8', 5], ['9', 5]]

S2  : [['00', 0], ['01', 1], ['02', 1], ['03', 1], ['04', 1],
       ['05', 1], ['06', 0], ['07', 0], ['08', 0],['09', 0]]

S3  : [['000', 0], ['001', 0], ['002', 0], ['003', 0] ,['004', 0],
       ['005', 0], ['006', 0], ['007', 0], ['008', 0], ['009', 0]]

S4  : [['0000', 0], ['0001', 0], ['0002', 0], ['0003' , 0],
       ['0004', 0], ['0005', 0], ['0006', 0], ['0007', 0],
       ['0008', 0], ['0009', 0]]
```

Nombres de $\mathbb{N}(\sqrt{2})$

Les nombres $a+b\sqrt{2}$ où a et b sont des entiers positifs constituent une classe appelée NV2 pour laquelle sont définis quelques méthodes utiles comme la comparaison, le produit, le quotient... La comparaison par exemple ne peut se baser sur la valeur décimale, nécessairement approximative. Pour comparer deux nombres donnés par les couples $(a;b)$, $(a';b')$, on utilisera donc la méthode « *compareTo*(*x*) » qui renvoie 0, 1 ou -1 selon que le nombre auquel elle s'applique est égal, supérieur ou inférieur au nombre *x*. La quantité « delta » est utile pour cette comparaison lorsque $a-a'$ et $b-b'$ ne sont pas du même signe.

Le petit programme d'accompagnement montre les usages ordinaires de cette classe. Nous réutiliserons par ailleurs celle-ci à la page suivante.

```
from math import floor,sqrt
class NV2 :# Classe pour les nombres premiers de N(sqrt 2)
    def __init__(self,A,B):
        self.a,self.b,self.aq,self.bq=A,B,0,0
    def getA(self) : return self.a
    def getB(self) : return self.b
    def getIn(self) : return self.a+self.b
    def compareTo(self,f):
        if self.a==f.a :
            if self.b==f.b : return 0
            if self.b-f.b<0 : return -1
            return 1
        if self.b==f.b or (self.a-f.a)*(self.b-f.b)>0:
            if self.a-f.a<0 : return -1
            return 1
        delta=(self.a-f.a)**2-2*(self.b-f.b)**2
        if self.a-f.a>0 and delta<0 or self.a-f.a<0 and delta>0 : return -1
        return 1
    def getProduit(self,X) :
        return NV2(X.a*self.a+2*X.b*self.b,X.a*self.b+X.b*self.a)
    def divise(self,X) :
        self.aq=float(X.a*self.a-2*X.b*self.b)/(self.a**2-2*self.b**2)
        self.bq=float(X.b*self.a-X.a*self.b)/(self.a**2-2*self.b**2)
        if floor(self.aq)==self.aq and floor(self.bq)==self.bq : return True
        return False
    def afficheQuotient(self) :
        print(" {} + {}.sqrt(2) approx {}".format(self.aq,self.bq,self.aq+sqrt(2)*self.bq))
    def affiche(self) :
        print(" {} + {}.sqrt(2) approx {}".format(self.a,self.b,self.a+sqrt(2)*self.b))
```

```
a1=int(input("Entrer la partie rationnelle (a) du premier nombre : "))
b1=int(input("Entrer la partie irrationnelle (b) du premier nombre : "))
a2=int(input("Entrer la partie rationnelle (a) du second nombre : "))
b2=int(input("Entrer la partie irrationnelle (b) du second nombre : "))
x1,x2=NV2(a1,b1),NV2(a2,b2)
c,s=x1.compareTo(x2),"égal"
if c==1 : s="supérieur"
elif c==-1 : s="inférieur"
print("Le premier nombre {} est {} au second {}".format(x1.a+sqrt(2)*x1.b,s,x2.a+sqrt(2)*x2.b))
print("Le produit des deux nombres vaut :")
x1.getProduit(x2).affiche()
if x1.divise(x2)==True :
    print("Le premier divise le second, le quotient est :")
    x1.afficheQuotient()
if x2.divise(x1)==True :
    print("Le second divise le premier, le quotient est :")
    x2.afficheQuotient()
```

```
Entrer la partie rationnelle (a) du premier nombre : -6
Entrer la partie irrationnelle (b) du premier nombre : 5
Entrer la partie rationnelle (a) du second nombre : 0
Entrer la partie irrationnelle (b) du second nombre : 1
Le premier nombre 1.0710678118654755 est inférieur au
                                  au second 1.4142135623730951
Le produit des deux nombres vaut :
 10 + -6.sqrt(2) approx 1.5147186257614287
Le second divise le premier, le quotient est :
 5.0 + -3.0.sqrt(2) approx 0.7573593128807143
```

Nombres premiers de $\mathbb{N}(\sqrt{2})$

Les nombres $a+b\sqrt{2}$ de cet ensemble étant ordonnés par les valeurs de *s=a+b* croissantes, cet algorithme passe en revue les premiers nombres (jusqu'à *s=smax*) et affiche les nombres premiers trouvés par une méthode inspirée du crible d'Ératosthène. En modifiant le paramètre *smax* on obtient plus ou moins de nombres premiers, sur notre illustration, *smax=4*. La fonction *mettreAjourLaListeDesMultiples*() fait ce que dit son nom, elle complète la liste nombresMultiples des nombres composés.

```
def mettreAjourLaListeDesMultiples() :
    global nombresMultiples
    nProduit=NV2(no.a,no.b)
    reserve=list(nombresMultiples)
    finished=False
    while finished==False:
        finished=True
        for nm in nombresMultiples :
            ajout=nProduit.getProduit(nm)
            if ajout.getIn()<=smax and reserve.count(ajout)==0 :
                reserve.append(ajout)
                finished=False
        nProduit=nProduit.getProduit(no)
    nombresMultiples=reserve

smax,nnp,nnt=20,0,0 #smax est à modifier pour aller plus ou moins loin
nombresMultiples,nombresPremiers=[NV2(1,0)],[]
for s in range(1,smax+1):
    for b in range(s+1):
        a=s-b
        nnt+=1
        no=NV2(a,b)
        premier=True
        for nm in nombresMultiples :
            if nm.compareTo(no)==0 :
                premier=False
                break
        if premier==True :
            nombresPremiers.append(no)
            nnp+=1
            print("Nombre premier no {} : {}+{}.sqrt(2), total : {},\
proportion : {}%".format(nnp,a,b,nnt,nnp/nnt*100))
            mettreAjourLaListeDesMultiples()
```

```
Nombre premier no 1 : 0+1.sqrt(2), total : 2,proportion : 50.0%
Nombre premier no 2 : 1+1.sqrt(2), total : 4,proportion : 50.0%
Nombre premier no 3 : 3+0.sqrt(2), total : 6,proportion : 50.0%
Nombre premier no 4 : 1+2.sqrt(2), total : 8,proportion : 50.0%
Nombre premier no 5 : 3+1.sqrt(2), total : 11,proportion : 45.454545454545455%
Nombre premier no 6 : 1+3.sqrt(2), total : 13,proportion : 46.15384615384615%
```

Résolution des équations du 3ème degré

La méthode de Tartaglia-Cardan décrite dans le texte est appliquée au cas général où l'on a $a x^3+b x^2+c x+d=0$ avec $a\neq 0$. Les trois types de solutions sont distingués à l'aide du signe du discriminant (Delta) de l'équation transformée. Les solutions sont données approximativement

```
from math import *
def racineCubique(nbr):
    return copysign((abs(nbr)**(1./3.)),nbr)
def jFois(cpx):
    return complex((-cpx.real-cpx.imag*sqrt(3))/2,(cpx.real*sqrt(3)-cpx.imag)/2)
def j2Fois(cpx):
    return complex((-cpx.real+cpx.imag*sqrt(3))/2,(-cpx.real*sqrt(3)-cpx.imag)/2)
def affichage(cpx):
    if cpx.imag>0:return("{}+{} i".format(cpx.real,cpx.imag))
    else:return("{}-{} i".format(cpx.real,-cpx.imag))

a,b=int(input("a (coeff. de x^3)= ")),int(input("b (coeff. de x^2)= "))
c,d=int(input("c (coeff. de x^1)= ")),int(input("d (coeff. de x^0)= "))
p_num,p_denom,q_num,q_denom=3*a*c-b*b,3*a*a,27*a*a*d-9*a*b*c+2*b*b*b,27*a*a*a
delta_num,delta_denom=q_num**2+4*p_num**3,(3*a)**6
print("p={}/{}, q={}/{}".format(p_num,p_denom,q_num,q_denom))
if delta_num>0:
    print("Delta={}/{}>0".format(delta_num,delta_denom))
    print("Il y a une solution réelle et deux solutions complexes conjugués.")
    u=(-q_num/q_denom+sqrt(delta_num/delta_denom))/2/a
    v=(-q_num/q_denom-sqrt(delta_num/delta_denom))/2/a
    x=racineCubique(u)+racineCubique(v)-b/3/a
    print("racine réelle approx : {}".format(x))
    x=jFois(complex(racineCubique(u)))+j2Fois(complex(racineCubique(v)))-b/3/a
    print("racine complexe 1 : {}".format(affichage(x)))
    x=j2Fois(complex(racineCubique(u)))+jFois(complex(racineCubique(v)))-b/3/a
    print("racine complexe 2 : {}".format(affichage(x)))
elif delta_num==0:
    print("Delta=0".format(delta_num,delta_denom))
    if p_num==0 and q_num==0:print("Il n'y a qu'une solution réelle triple:{}".format(-b/3/a))
    else:
        print("Il y a deux solutions réelles: une simple, l'autre double.")
        print("racine simple approx : {}".format(3*p_denom*q_num/(p_num*q_denom)-b/3/a))
        print("racine double approx : {}".format(-3*p_denom*q_num/(2*p_num*q_denom)-b/3/a))
else:
    print("Delta={}/{}<0".format(delta_num,delta_denom))
    print("Il y a trois solutions réelles.")
    X=acos(-q_num/q_denom/2*sqrt(27/(-(p_num/p_denom)**3)))/3
    for k in range(3):
        x=2*sqrt(-p_num/p_denom/3)*cos(X+2*k*pi/3)-b/3/a
        print("racine réelle {} approx : {}".format(k+1,x))
```

```
a (coeff. de x^3)= 1
b (coeff. de x^2)= 2
c (coeff. de x^1)= 3
d (coeff. de x^0)= 5
p=5/3, q=97/27
Delta=9909/729>0
Il y a une solution réelle et deux solutions complexes conjugués.
racine réelle approx : -1.8437342778980685
racine complexe 1 : -0.07813286105096562+1.644926377599972 i
racine complexe 2 : -0.07813286105096562-1.644926377599972 i
```

```
a (coeff. de x^3)= 1
b (coeff. de x^2)= 5
c (coeff. de x^1)= 1
d (coeff. de x^0)= -1
p=-22/3, q=178/27
Delta=-10908/729<0
Il y a trois solutions réelles.
racine réelle 1 approx : 0.34889421750071636
racine réelle 2 approx : -4.744826077681923
racine réelle 3 approx : -0.6040681398187946
```

Tracer une image de l'ensemble de Mandelbrot

Pour chaque pixel de coordonnées (*a*;*b*) de la fenêtre d'affichage, on pose *c*=*a*+*bi* (un nombre complexe) et on examine le destin de la suite $z_n \mapsto z_{n+1}=z_n^2+c$ avec $z_0=0$. Si, pour *n*<*t*, on a $|z_n|>2$, alors le point n'appartient pas à l'ensemble et on le colorie avec une nuance de gris. Les paramètres modifiables sont ceux de la 1ère ligne : *t* (la profondeur de l'examen), *minX* et *minY* (les valeurs de début pour *a* et *b*), *cote* (l'amplitude pour *a* et *b*), *pix* (la dimension de la fenêtre). Avec les paramètres choisis, l'ensemble *M* est tracé presque en entier. Modifier les paramètres pour zoomer manuellement. La plus grande partie du programme correspond au traçage d'un cadre gradué pour se repérer et au choix d'une couleur (ici, une nuance de gris).

```
from tkinter import *
#profondeur,min(a),min(b),amplitude,taille
t,minX,minY,cote,pix=200,-1.5,-1,2,600
fen=Tk()
fen.title('Ensemble de Mandelbrot')
cadre=Canvas(fen,bg='white',height=pix,width=pix,borderwidth=0)
cadre.pack()
pas=cote/pix
for a in range(pix):
    for b in range(pix):
        c=complex(minX+a*pas,minY+b*pas)
        z,s=complex(0,0),0
        while abs(z)<2 and s<t:
            z=z**2+c
            s+=1
        if s==t: cadre.create_rectangle(a,b,a+1,b+1,width=0,fill='black')
        else:
            coul,s='#',(int)(255/t)*s
            if s>15 : coul+=hex(s)[2:]+hex(s)[2:]+hex(s)[2:]
            else : coul+='0'+hex(s)[2:]+'0'+hex(s)[2:]+'0'+hex(s)[2:]
            cadre.create_rectangle(a,b,a+1,b+1,width=0,fill=coul)
carre=cadre.create_rectangle(5,5,pix-5,pix-5,width=1,outline='white')
cadre.create_line((pix-4)/3,5,(pix-4)/3,15,width=1,fill='white')
cadre.create_line(2*(pix-4)/3,5,2*(pix-4)/3,15,width=1,fill='white')
cadre.create_line(5,(pix-4)/3,15,(pix-4)/3,width=1,fill='white')
cadre.create_line(5,2*(pix-4)/3,15,2*(pix-4)/3,width=1,fill='white')
cadre.create_text((pix-4)/3,25,text=str(round(minX+cote/3,4)),fill='white')
cadre.create_text(2*(pix-4)/3,25,text=str(round(minX+2*cote/3,4)),fill='white')
cadre.create_text(35,(pix-4)/3,text=str(round(minY+cote/3,4)),fill='white')
cadre.create_text(35,2*(pix-4)/3,text=str(round(minY+2*cote/3,4)),fill='white')
fen.mainloop()
```

Produits et quotients de deux quaternions

La multiplication des quaternions n'étant ni très intuitive ni commutative, il peut être amusant de le vérifier en exécutant ce programme qui calcule, pour les deux quaternions $q=a+bi+cj+dk$ et $q'=a'+b'i+c'j+d'k$ qui sont entrés, en plus des deux produits qq' et $q'q$: la somme $q+q'$, les deux différences $q-q'$ et $q'-q$ (elles sont opposées), les quatre quotients qq'^{-1} et $q'^{-1}q$ (de q par q') et $q'q^{-1}$ et $q^{-1}q'$ (de q' par q).

```
from math import sqrt
def difference(a,b,c,d,e,f,g,h):
    return "{}+{}i+{}j+{}k".format(a-e,b-f,c-g,d-h)
def produit(a,b,c,d,e,f,g,h):
    return "{}+{}i+{}j+{}k".format(round(a*e-b*f-c*g-d*h,3),round(a*f+b*e+c*h-d*g,3),\
                                   round(a*g-b*h+c*e+d*f,3),round(a*h+b*g-c*f+d*e,3))
def quotientDroite(a,b,c,d,e,f,g,h):
    n=sqrt(e**2+f**2+g**2+h**2)
    return produit(a,b,c,d,e/n,-f/n,-g/n,-h/n)
def quotientGauche(a,b,c,d,e,f,g,h):
    n=sqrt(e**2+f**2+g**2+h**2)
    return produit(e/n,-f/n,-g/n,-h/n,a,b,c,d)

a1,b1=float(input("a (scalaire)= ")),float(input("b (vecteur i)= "))
c1,d1=float(input("c (vecteur j)= ")),float(input("d (vecteur k)= "))
a2,b2=float(input("a' (scalaire)= ")),float(input("b' (vecteur i)= "))
c2,d2=float(input("c' (vecteur j)= ")),float(input("d' (vecteur k)= "))
print("Somme : q+q'=q'+q={}+{}i+{}j+{}k".format(a1+a2,b1+b2,c1+c2,d1+d2))
print("Différence1 : q-q'={}".format(difference(a1,b1,c1,d1,a2,b2,c2,d2)))
print("Différence2 : q'-q={}".format(difference(a2,b2,c2,d2,a1,b1,c1,d1)))
print("Produit1 : qq'={}".format(produit(a1,b1,c1,d1,a2,b2,c2,d2)))
print("Produit2 : q'q={}".format(produit(a2,b2,c2,d2,a1,b1,c1,d1)))
print("Quotient1 à droite : qq'-1={}".format(quotientDroite(a1,b1,c1,d1,a2,b2,c2,d2)))
print("Quotient1 à gauche : q'-1q={}".format(quotientGauche(a1,b1,c1,d1,a2,b2,c2,d2)))
print("Quotient2 à droite : q'q-1={}".format(quotientDroite(a2,b2,c2,d2,a1,b1,c1,d1)))
print("Quotient2 à gauche : q-1q'={}".format(quotientGauche(a2,b2,c2,d2,a1,b1,c1,d1)))
```

```
a (scalaire)= 0
b (vecteur i)= 3
c (vecteur j)= 0
d (vecteur k)= -1
a' (scalaire)= 2
b' (vecteur i)= 0
c' (vecteur j)= 1
d' (vecteur k)= 1
Somme : q+q'=q'+q=2.0+3.0i+1.0j+0.0k
Différence1 : q-q'=-2.0+3.0i+-1.0j+-2.0k
Différence2 : q'-q=2.0+-3.0i+1.0j+2.0k
Produit1 : qq'=1.0+7.0i+-3.0j+1.0k
Produit2 : q'q=1.0+5.0i+3.0j+-5.0k
Quotient1 à droite : qq'-1=-0.408+2.041i+1.225j+-2.041k
Quotient1 à gauche : q'-1q=-0.408+2.858i+-1.225j+0.408k
Quotient2 à droite : q'q-1=-0.316+-1.581i+-0.949j+1.581k
Quotient2 à gauche : q-1q'=-0.316+-2.214i+0.949j+-0.316k
```

Rotation dans l'espace avec les quaternions

Nous envisageons ici la rotation d'un vecteur $\vec{w}$ de coordonnées (*x*,*y*,*z*) par la rotation définie par le quaternion *q*=*a*+*bi*+*cj*+*dk*. La conjugaison de ce vecteur par *q* (qui aura été normalisé) a été définie dans le texte par la formule $q\vec{w}q^{-1}$, ce qui a conduit aux expressions des composantes (*x'*,*y'*,*z'*) du vecteur image données dans le texte et utilisées ici. Pour déterminer l'angle de la rotation, on utilise l'égalité des parties scalaires du vecteur normalisé $\cos\frac{\alpha}{2}=\frac{a}{\sqrt{a^2+b^2+c^2+d^2}}$. Cet angle connu, on peut déterminer les composantes du vecteur unitaire autour duquel se fait la rotation.

Le premier exemple donné a été détaillé dans le texte (angle :120°, vecteur :*i*+*j*+*k*), pour l'autre exemple, seule la partie scalaire est changée.

```
from math import sqrt,acos,sin,pi
a1,b1=float(input("a (scalaire)= ")),float(input("b (vecteur i)= "))
c1,d1=float(input("c (vecteur j)= ")),float(input("d (vecteur k)= "))
norme=sqrt(a1**2+b1**2+c1**2+d1**2)
a2,b2,c2,d2=a1/norme,b1/norme,c1/norme,d1/norme
if norme==1:print("Le quaternion est unitaire")
else:print("Voici votre quaternion normalisé :{}+{}i+{}j+{}k"\
          .format(round(a2,3),round(b2,3),round(c2,3),round(d2,3)))
print("L'angle de la rotation est {}rad, \
soit environ {}°".format(round(2*acos(a2),3),round(360*acos(a2)/pi,3)))
s=sin(2*acos(a2))
print("Le vecteur unitaire :{}i+{}j+{}k".format(round(b2/s,3),round(c2/s,3),round(d2/s,3)))
ab,ac,ad,bb,bc,bd,cc,cd,dd=a2*b2,a2*c2,a2*d2,b2*b2,b2*c2,b2*d2,c2*c2,d2*c2,d2*d2
print("x'={}x+{}y+{}z".format(round(1-2*(cc+dd),3),round(2*(bc-ad),3),round(2*(ac+bd),3)))
print("y'={}x+{}y+{}z".format(round(2*(ad+bc),3),round(1-2*(bb+dd),3),round(2*(cd-ab),3)))
print("z'={}x+{}y+{}z".format(round(2*(bd-ac),3),round(2*(ab+cd),3),round(1-2*(bb+cc),3)))
```

```
a (scalaire)= 0.5
b (vecteur i)= 0.5
c (vecteur j)= 0.5
d (vecteur k)= 0.5
Le quaternion est unitaire
L'angle de la rotation est 2.094rad, soit environ 120.0°
Le vecteur unitaire :0.577i+0.577j+0.577k
x'=0.0x+0.0y+1.0z
y'=1.0x+0.0y+0.0z
z'=0.0x+1.0y+0.0z
```

```
a (scalaire)= 1.5
b (vecteur i)= 0.5
c (vecteur j)= 0.5
d (vecteur k)= 0.5
Voici votre quaternion normalisé :0.866+0.289i+0.289j+0.289k
L'angle de la rotation est 1.047rad, soit environ 60.0°
Le vecteur unitaire :0.333i+0.333j+0.333k
x'=0.667x+-0.333y+0.667z
y'=0.667x+0.667y+-0.333z
z'=-0.333x+0.667y+0.667z
```