Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique pour tous
A. Troesch, J.-P. Becirspahic

2013/2014

Corrigé du TP n° 3

Exercice 1.

La première idée qui vient à l'esprit consiste à utiliser les formules $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$, ce qui conduit à écrire :

```
from math import sqrt

def solve(a, b, c):
    delta = b**2-4*a*c
    if delta > 0:
        x = (-b-sqrt(delta))/2/a
        y = (-b+sqrt(delta))/2/a
        print('deux racines simples {} et {}'.format(x, y))
    elif delta == 0:
        x = -b/2/a
        print('une racine simple',x)
    else:
        print('pas de solution')
```

Essayons avec les valeurs suggérées :

```
>>> solve(0.01, 0.2, 1)
deux racines simples -10.000000131708903 et -9.999999868291098

>>> solve(0.011025, 0.21, 1)
pas de solution
```

Le problème est que dans les deux cas, le discriminant (théorique) est nul ; cependant, dans le premier cas le calcul numérique donne `delta = 6.938893903907228e-18` et dans le second, `delta = -6.938893903907228e-18`.
Pour palier à ce problème, on peut observer que lorsque $\Delta \ll b^2$, les deux racines de l'équation sont quasiment confondues. On pourrait donc remplacer la condition $\Delta = 0$ par : $|\Delta| \leqslant \varepsilon b^2$, la valeur de ε étant choisie petite. Ceci conduirait à la modification suivante :

```
def solve(a, b, c, epsilon=1e-15):
    delta = b**2-4*a*c
    if delta > epsilon*b**2:
        x = (-b-sqrt(delta))/2/a
        y = (-b+sqrt(delta))/2/a
        print('deux racines simples {} et {}'.format(x, y))
    elif delta < -epsilon*b**2:
        print('pas de solution')
    else:
        x = -b/2/a
        print('une racine simple',x)
```

Essayons avec cette nouvelle fonction :

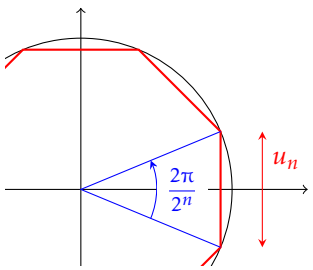
```
>>> solve(0.01, 0.2, 1)
une racine simple -10.0

>>> solve(0.011025, 0.21, 1)
une racine simple -9.523809523809524
```

Exercice 2.

La méthode décrite ici pour calculer une valeur approchée de π consiste à approcher le périmètre du cercle unité (égal à 2π) par le périmètre d'un polygone régulier à 2^n côtés inscrit dans le cercle unité.

Si u_n désigne la longueur d'un côté, la figure ci-contre permet de constater que $\sin\left(\frac{\pi}{2^n}\right) = \frac{u_n}{2}$. Ainsi, $2^{n-1}u_n = 2^n \sin\left(\frac{\pi}{2^n}\right)$ et $\lim 2^{n-1}u_n = \pi$.



On calcule $4 - u_n^2 = 4\cos^2\left(\frac{\pi}{2^n}\right) = 4\left(1 - 2\sin^2\left(\frac{\pi}{2^{n+1}}\right)\right)^2 = 4\left(1 - \frac{u_{n+1}^2}{2}\right)^2 = \left(2 - u_{n+1}^2\right)^2$ donc

$2 - u_{n+1}^2 = \sqrt{4 - u_n^2}$ et $u_{n+1} = \sqrt{2 - \sqrt{4 - u_n^2}}$.

L'algorithme naïf consiste à écrire simplement :

```
from math import sqrt

def archimede1(n):
    u = 2.
    print(u)
    for k in range(1, n):
        u = sqrt(2-sqrt(4-u**2))
        print(2**k*u)
```

Au départ, tout semble bien se passer, mais les choses se gâtent lorsque n augmente :

```
>>> archimede1(40)
2.0
2.8284271247461903
3.0614674589207187
3.121445152258053
..................
3.1415138011441455
3.1415729403678827
..................
3.1415926334632482
3.141592654807589
..................
3.1415929109396727
3.141594125195191
..................
3.142451272494134
3.1622776601683795
3.1622776601683795
3.4641016151377544
4.0
0.0
0.0
...
0.0
```

L'explication est facile à trouver : la suite u converge vers 0, donc calculer le produit $2^{n-1} \times u_n$ revient à multiplier un infiniment grand par un infiniment petit, avec rapidement une perte drastique de précision (et un produit qui devient nul dès lors que u_n est numériquement évalué à 0).

Utiliser le développement limité à l'ordre 2 de la fonction $x \mapsto \sqrt{1-x}$ conduit à remplacer la relation de récurrence précédente par la relation $u_{n+1} \approx \frac{u_n}{2} + \frac{u_n^3}{64}$. Cependant, cette relation n'est acceptable que pour de grandes valeurs de n ; l'idée consiste donc à utiliser la relation initiale en deçà d'un certain rang n_0 et cette approximation au delà, dans l'espoir que la perte de précision du calcul numérique de u_n soit atténuée.
Ceci conduit à l'algorithme suivant :

```
def archimede2(n, n0=10):
    u = 2.
    print(u)
    for k in range(1, n0):
        u = sqrt(2-sqrt(4-u**2))
        print(2**k*u)
    for k in range(n0, n):
        u = u/2+u**3/64
        print(2**k*u)
```

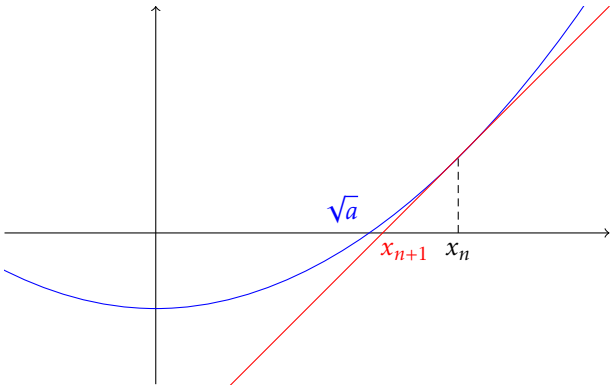
Connaissant la valeur de π, on observe expérimentalement que la valeur $n_0 = 10$ semble donner les meilleurs résultats et conduit à l'approximation $\pi \approx 3.1415926535763585$.

Néanmoins, cette solution n'est guère satisfaisante : la suite ainsi définie ne converge plus vers π mais vers une valeur très proche, et on ne maîtrise pas le nombre de chiffres significatifs que l'on est susceptible d'obtenir.

Pour obtenir un nombre bien plus conséquent de décimales, il va falloir abandonner les nombres flottants, la perte de précision inhérente à tout calcul effectué avec eux étant rédhibitoire, et travailler avec les nombres entiers. C'est pourquoi on cherche à calculer $\widetilde{u}_n = \lfloor 10^{200} u_n \rfloor$, de sorte que $10^{-200}\widetilde{u}_n$ soit une approximation décimale à 200 chiffres significatifs de u_n. Cependant, pour obtenir $\widetilde{u}_n$ il faut être en mesure de calculer sans perte de précision $\lfloor \sqrt{n} \rfloor$ et $\lceil \sqrt{n} \rceil$ pour de grandes valeurs de n, ce qui exclut la version naïve `int(sqrt(n))`, qui utilise le type `float` et donc conduit à des erreurs.

Calcul de la racine carrée d'un entier

La méthode de Newton-Raphson, dite aussi méthode des tangentes, consiste à remplacer l'équation (non linéaire) $f(x)=0$ par l'équation linéaire approchée $f(c)+(x-c)f'(c)=0$. On est ainsi conduit à étudier la suite $(x_n)_{n\in\mathbb{N}}$ définie par la relation de récurrence $x_{n+1}=x_n-\dfrac{f(x_n)}{f'(x_n)}$. Dans le cas de la fonction $f:x\mapsto x^2-a$ on obtient $x_{n+1}=g(x_n)$ avec $g(x)=\dfrac{1}{2}\left(x+\dfrac{a}{x}\right)$.



La convexité de la fonction f (le fait que son graphe soit situé au dessus de chacune de ses tangentes) prouve que si $x_0>\sqrt{a}$, la suite $(x_n)_{n\in\mathbb{N}}$ décroit et converge vers $\sqrt{a}$.

Lorsque a est entier, on définit une suite d'entiers naturels en posant $r_0=a$ et $\forall n\in\mathbb{N}$, $r_{n+1}=\lfloor g(r_n)\rfloor$.
Nous avons $r_{n+1}-r_n=\left\lfloor\dfrac{1}{2}\left(\dfrac{a}{r_n}-r_n\right)\right\rfloor$ donc tant qu'on aura $r_n>\sqrt{a}$ nous aurons $\dfrac{1}{2}\left(\dfrac{a}{r_n}-r_n\right)<0$ et ainsi $r_{n+1}-r_n\leqslant -1$.
Ceci assure l'existence d'un rang N vérifiant $r_N\leqslant\sqrt{a}<r_{N-1}$.
L'inégalité $r_N\leqslant\sqrt{a}$ prouve que $r_N\leqslant\lfloor\sqrt{a}\rfloor$; l'inégalité $r_{N-1}>\sqrt{a}$ prouve que $g(r_{N-1})>\sqrt{a}$ et donc que $r_N=\lfloor g(r_{N-1})\rfloor\geqslant\lfloor\sqrt{a}\rfloor$.
Ainsi, nous avons $r_N=\lfloor\sqrt{a}\rfloor$, et cette quantité peut être calculée à l'aide de la fonction :

```
def isqrt(a):
    r = a
    while r**2 > a:
        r = (r**2+a)//(2*r)
    return r
```

Il est possible d'améliorer un peu les performances de cette fonction en essayant de partir d'une valeur initiale plus proche de $\sqrt{a}$. En notant k le nombre de chiffres de l'écriture décimale de a nous avons $a<10^k$ donc $\sqrt{a}<10^{\lceil k/2\rceil}$, ce qui permet de choisir $r_0=10^{\lceil k/2\rceil}$.
Par ailleurs, on peut observer que le carré de r est calculé deux fois. Or le temps de calcul croît avec le nombre de chiffres de r ; puisqu'on prévoit d'appliquer cette fonction à de très grandes valeurs on a intérêt à n'effectuer ce calcul qu'une fois, ce qui conduit à la version suivante :

```
def isqrt(a):
    k = len(str(a))
    r = 10**((k+1)//2)
    r2 = r*r
    while r2 > a:
        r = (r2 + a)//(2*r)
        r2 = r*r
    return r
```

Pour obtenir $\lceil\sqrt{a}\rceil$ il suffit d'observer que $\lceil x\rceil=\begin{cases}\lfloor x\rfloor+1 & \text{lorsque } x\notin\mathbb{Z}\\ \lfloor x\rfloor & \text{sinon}\end{cases}$, soit :

```
def iisqrt(a):
    r = isqrt(a)
    if r*r == a:
        return r
    else:
        return r+1
```

Retour vers la méthode d'Archimède

Il ne semble pas qu'on soit en mesure de prouver que la formule de l'énoncé donne effectivement la valeur de $\widetilde{u_n}$, aussi allons nous procéder par encadrement en considérant les deux fonctions :

$$f(x) = \left\lfloor \sqrt{2.10^{400} - \left\lceil \sqrt{4.10^{800} - 10^{400}x^2} \right\rceil} \right\rfloor$$
$$g(x) = \left\lceil \sqrt{2.10^{400} - \left\lfloor \sqrt{4.10^{800} - 10^{400}x^2} \right\rfloor} \right\rceil$$

et en définissant les suites $(v_n)_{n\in\mathbb{N}^*}$ et $(w_n)_{n\in\mathbb{N}^*}$ par : $v_1 = w_1 = 2.10^{200}$ et $v_{n+1} = f(v_n)$, $w_{n+1} = g(w_n)$.

La suite $x_n = 10^{200}u_n$ vérifie la relation de récurrence $x_{n+1} = \sqrt{2.10^{400} - \sqrt{4.10^{800} - 10^{400}x_n^2}}$ donc $f(x_n) \leqslant x_{n+1} \leqslant g(x_n)$.
Les fonctions f et g sont croissantes donc cet encadrement permet de prouver par récurrence que pour tout $n \geqslant 1$, $v_n \leqslant x_n \leqslant w_n$, ce qui donne une approximation entière par défaut et par excès de $10^{200}u_n$.
Ceci nous amène aux définitions suivantes :

```
def defaut(n):
    v = 2*10**200
    for i in range(n-1):
        v = isqrt(2*10**400-iisqrt(4*10**800-10**400*v**2))
    return 2**(n-1)*v

def exces(n):
    w = 2*10**200
    for i in range(n-1):
        w = iisqrt(2*10**400-isqrt(4*10**800-10**400*w**2))
    return 2**(n-1)*w
```

Essayons avec $n = 50$ puis avec $n = 100$:

```
>>> defaut(50)
31415926535897932384626433832754262776155769885412437966882133232316358545431861932134758882
70727948482992901255130437524757486317437097562753314934808378461552770636084240363422059 2
3256568429120847872

>>> defaut(100)
31415926535897932384626433832795028841971693993751058209749413764322582213624769153530391060
47376132104178903852025403310130926577281292092209750275378656491402246052246701550724229660
9243118858893524992
```

On calcule en outre $w(50) - v(50) = 1125899906842624 < 10^{16}$ et $w_{100} - v_{100} = 126765060022822940149670320537 6 < 10^{31}$ ce qui assure 185 décimales exactes pour le calcul de u_{50} et 170 décimales pour u_{100}.
Compte tenu de la valeur connue de π on observe que u_{50} donne 30 décimales exactes et u_{100} 60 décimales exactes de π.

Remarque. Il est possible d'améliorer les performances de ces fonctions en itérant non pas la suite $(u_n)_{n\in\mathbb{N}^*}$ mais la suite $(u_n^2)_{n\in\mathbb{N}^*}$, ce qui permet de n'effectuer qu'un calcul de racine carrée à chaque étape plutôt que deux. Ceci donne la version :

```
def defaut2(n):
    vv = 4*10**400
    for i in range(n-1):
        vv = 2*10**400-iisqrt(4*10**800-10**400*vv)
    return 2**(n-1)*isqrt(vv)
```

Exercice 3.

Commençons par illustrer la méthode en calculant $\left\lfloor \sqrt{64015} \right\rfloor$:

6.4 0.1 5	2 5 3
− 4	② × 2 = 4
2.4 0	4 ⑤ × 5 = 2 2 5
− 2 2 5	5 0 ③ × 3 = 1 5 0 9
1 5.1 5	
− 1 5 0 9	
6	

Une fois toutes les tranches traitées, on a obtenu $\left\lfloor \sqrt{64015} \right\rfloor = 253$.

Débutons la rédaction de l'algorithme correspondant par une fonction qui découpe en tranches un entier en retournant la liste des tranches :

```
def tranches(n):
    x = n
    lst = []
    while(x>0):
        lst = [x % 100] + lst
        x //= 100
    return lst
```

Poursuivons avec une fonction qui recherche le plus grand entier i vérifiant $(20r+i)\times i \leqslant x$:

```
def cherche(r, x):
    i = 9
    while (20*r+i)*i > x:
        i -= 1
    return i
```

Écrivons enfin la fonction principale :

```
def sqrt(n):
    lst = tranches(n)
    x = r = 0
    while len(lst) > 0:
        x = 100*x+lst.pop(0)
        i = cherche(r, x)
        x -= (20*r+i)*i
        r = r*10+i
    return r
```

Cette dernière fonction réalise l'itération de deux suites r et x. Dans le cas où $n = 64015$, elles prennent successivement les valeurs : $(r_1, x_1) = (2, 2)$, $(r_2, x_2) = (25, 15)$, $(r_3, x_3) = (253, 6)$.
La terminaison de l'algorithme est assurée par le fait que la méthode `pop(0)` supprime le premier élément de la liste à qui elle est appliquée.
Notons $t_1, t_2, \ldots, t_p$ les différentes tranches de n, et prouvons les invariants : $r_k = \left\lfloor \sqrt{[t_1 t_2 \cdots t_k]} \right\rfloor$ et $r_k^2 + x_k = [t_1 t_2 \cdots t_k]$, les crochets désignant ici la concaténation des tranches.
C'est clair pour $k = 0$ car $r_0 = x_0$ et $[] = 0$.
Supposons le résultat acquis au rang k, et prouvons-le au rang $k+1$: nous avons $r_{k+1} = 10r_k + i$, i étant le plus grand entier vérifiant $(20r_k + i) \times i \leqslant 100x_k + t_{k+1}$. Ainsi,

$$r_{k+1}^2 = 100r_k^2 + (20r_k + i) \times i \leqslant 100r_k^2 + 100x_k + t_{k+1} = 100[t_1 t_2 \cdots t_k] + t_{k+1} = [t_1 t_2 \cdots t_{k+1}]$$

Le caractère maximal de i prouve que $r_{k+1} = \left\lfloor \sqrt{[t_1 t_2 \cdots t_{k+1}]} \right\rfloor$. Enfin,

$$x_{k+1} = 100x_k + t_{k+1} - (20r_k + i) \times i = 100x_k + t_{k+1} - r_{k+1}^2 + 100r_k^2 = 100[t_1 t_2 \cdots t_k] + t_{k+1} - r_{k+1}^2 = [t_1 t_2 \cdots t_{k+1}] - r_{k+1}^2$$

ce qui achève la démonstration.
On en déduit que cette fonction retourne la valeur de $r_p = \left\lfloor \sqrt{[t_1 t_2 \cdots t_p]} \right\rfloor = \lfloor \sqrt{n} \rfloor$.
Pour obtenir les 1000 premières décimales de $\sqrt{2}$, il suffit d'appliquer cette fonction à l'entier 2×10^{2000} :

```
>>> sqrt(2*10**2000)
14142135623730950488016887242096980785696718753769480731766797379907324784621070 38...
```