Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
N. Carré, A. Troesch

TP n° 5 : Problèmes liés à la représentation des réels

Correction de l'exercice 1 – Classique. Attention aux cas particuliers.

```
import math

def trinome(a,b,c):
    if a == 0:
        if b == 0:
            if c != 0:
                return set()
            else:
                raise ValueError("Équation dégénérée")
        else:
            return {-c / b}
    else:
        delta = b ** 2 - 4 * a * c
        if delta >= 0:
            return {(-b-math.sqrt(delta)) / (2*a), (-b+math.sqrt(delta)) / (2*a)}
        else:
            return set()
```

- Pour $(a,b,c) = (1,2,1)$, on obtient en toute logique : `{-1.0}`
- Pour $(a,b,c) = (0.01, 0.2, 1)$, on obtient : `{-10.000000131708903, -9.999999868291098}`.
  L'ordinateur détecte deux racines distinctes, mais il s'agit en ait de la même.
- Pour $(a,b,c) = (0.011025, 0.21, 1)$, on trouve : `set()`.
  Ainsi, l'ordinateur ne détecte pas la racine double. Le problème vient du fait que l'ordinateur a obtenu une valeur de Δ presque nulle, mais légèrement négative.

On modifie le programme en conséquence de sorte à laisser une petite marge d'erreur possible :

```
def trinome_ameliore(a,b,c):
    if a == 0:
        if b == 0:
            if c != 0:
                return set()
            else:
                raise ValueError("Équation dégénérée")
        else:
            return {-c / b}
    else:
        delta = b ** 2 - 4 * a * c
        if delta > 1e-10:
            return {(-b-math.sqrt(delta)) / (2 * a), (-b+math.sqrt(delta)) / (2* a)}
        elif abs(delta) <= 1e-10:
            return {(-b / (2 * a))}
        else:
            return set()
```

Correction de l'exercice 2 – Méthode d'Archimède pour le calcul de π.

Pour tout $n \geqslant 1$, on note u_n la longueur d'un côté d'un 2^n-gône régulier inscrit dans le cercle unité. Par convention, $u_1 = 2$.

1. **Où se cache π**

   (a) Un secteur du cercle circonscrit à un 2^n-gône régulier, délimité par un côté, définit un angle de $\frac{2\pi}{2^n}$. Ainsi, un demi-côté définit un secteur d'angle $\frac{\pi}{2^n}$. Les notations étant celles de la figure 1, la longueur d'un côté est donc :

$$u_n = 2 \cdot AH = 2 \cdot OA \cdot \sin\left(\frac{\pi}{n}\right) = 2\sin\left(\frac{\pi}{n}\right).$$

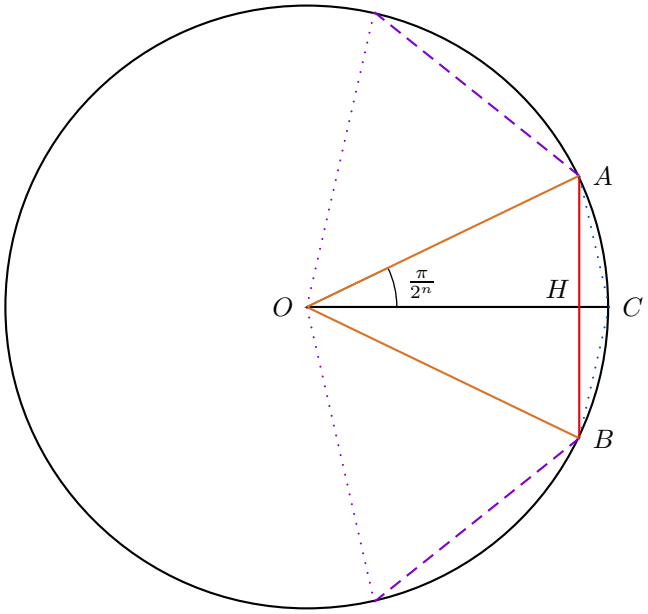

FIGURE 1 – Longueur d'un côté d'un 2^n-gône inscrit dans un cercle de rayon 1

(b) On a alors :

$$2^{n-1}u_n \underset{+\infty}{\sim} 2^{n-1} \cdot 2 \cdot \frac{\pi}{2^n} = \pi.$$

Cet équivalent nous assure que $\lim\limits_{n\to+\infty} 2^{n-1}u_n = \pi$.

La quantité $2^{n-1}u_n$ représente le demi-périmètre du 2^n-gône régulier inscrit dans le cercle unité. Géométriquement, lorsque n tend vers l'infini, le 2^n-gône est de plus en plus proche d'un cercle. Son périmètre tend vers le périmètre du cercle, égal à 2π.

Attention cependant à ces arguments géométriques qui peuvent s'avérer glissants. Imaginez par exemple qu'on approche le cercle unité par le plus grand polygône (non régulier) inclus dans le cercle, et défini par recollement de carrés d'un quadrillage régulier de pas $\frac{1}{2^n}$. Lorsque n tend vers $+\infty$, le tracé de ce polygône s'approche de celui du cercle. Pourtant son périmètre est égal à 8 pour tout n (pourquoi ?)

2. **Calcul naïf**

(a) Soit $n \geqslant 2$. Toujours en s'aidant de la figure 1, d'après le théorème de Pythagore :

$$u_{n+1}^2 = AC^2 = AH^2 + HC^2.$$

Or, toujours d'après Pythagore :

$$HC^2 = (1 - OH)^2 = 1 - 2OH + OH^2 = 2 - 2OH + AH^2.$$

Ainsi,

$$u_{n+1}^2 = 2 - 2OH = 2 - 2\sqrt{OA^2 - \left(\frac{u_n}{2}\right)^2} = 2 - \sqrt{4 - u_n^2}.$$

On obtient bien la relation attendue :

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

Pour $n = 1$, la relation reste vraie, puisque $u_1 = 1$ et $u_2 = \sqrt{2}$ (côté d'un carré inscrit dans un cercle de rayon 1).

On peut bien sûr obtenir également cette relation en utilisant les formules de trigonométrie, à partir de la première question (et plus spécifiquement les formules de duplication de l'angle).

(b) On calcule la suite (u_n) par une récurrence d'ordre 1 :

```
import math

def archimede1(n):
    u = 2
    print('u_0 = 2')
    for i in range(n):
        u = math.sqrt(2 - math.sqrt(4-u**2))
        print('u_{} = {}'.format(i+1,u * (2 ** (i+1))))

n = eval(input('n? '))
archimede1(n)
```

(c) Lorsqu'on demande $n = 40$, on se rend compte que la méthode semble converger jusqu'au rang 14 (on rappelle que $\pi = 3.1415926535...$), puis il y a divergence. On obtient en particulier :

$$u_{26} = 3.162277... \qquad u_{27} = 3.464101... \qquad u_{28} = 4 \qquad \text{puis:} \qquad u_n = 0,$$

pour tout $n > 28$. Évidemment, ce résultat numérique entre en contradiction avec l'analyse théorique. Le problème provient du fait que lorsque n tend vers $+\infty$, u_n tend vers 0, donc le produit $2^{n-1}u_n$ s'effectue entre un infiniment grand et un infiniment petit, ce qui crée une grande perte de précision. Le phénomène est amplifié par le fait que u_n lui-même sera calculé avec une précision diminuant par le fait que pour n grand, u_n est obtenu par différence de deux valeurs quasi-égales (2 et $\sqrt{4-u_n^2}$), provoquant l'annulation d'un grand nombre de chiffres significatifs.

On peut remarquer qu'on peut plus facilement itérer la suite u_n^2, nous épargnant un calcul de racine, espérant ainsi augmenter la précision :

```
def archimede2(n):
    v = 4
    print('u_0 = 2')
    for i in range(n):
        v = 2 - math.sqrt(4-v)
        print('u_{} = {}'.format(i+1,math.sqrt(v) * (2 ** (i+1))))
```

Mais cela ne règle en rien le problème.

3. **Augmentation de la précision**

Nous proposons maintenant une méthode basée sur l'utilisation de grands entiers afin d'augmenter la précision de calcul. On évalue en fait $10^{200} \times u_n$.

(a) Soit pour tout $n \geqslant 2$, v_n et w_n les suites définies par $v_1 = w_1 = 4 \cdot 10^{400}$ et pour tout $n \geqslant 1$:

$$v_{n+1} = 2 \cdot 10^{400} - \left\lceil \sqrt{4 \cdot 10^{800} - 10^{400} v_n} \right\rceil \qquad \text{et} \qquad w_{n+1} = 2 \cdot 10^{400} - \left\lfloor \sqrt{4 \cdot 10^{800} - 10^{400} w_n} \right\rfloor$$

En utilisant le fait que pour tout $x \in \mathbb{R}$,

$$\lfloor x \rfloor \leqslant x \leqslant \lceil x \rceil,$$

une récurrence immédiate amène l'encadrement attendu :

$$\forall n \in \mathbb{N}^*, \quad v_n \leqslant 10^{400} u_n^2 \leqslant w_n.$$

(b) On pourrait alors penser implémenter le calcul de (v_n) de la façon suivante :

```
import math
def calcul_v1(n):
    v = 2*(10**200)
    for i in range(n-1):
        v=2*(10**400)-math.ceil(math.sqrt(4*(10**800)-(10**400)*v))
```

Lorsqu'on demande le calcul de v_{10} (par exemple) par cette méthode, on recontre l'erreur suivante :

```
OverflowError: long int too large to convert to float
```

Cette erreur signifie que les entiers manipulés ne peuvent pas être convertis en réels (car trop longs), donc en particulier, les différentes opérations définies sur les réels (mais pas les entiers), comme par exemple le calcul de la racine carrée, ne peuvent pas être utilisées, puisqu'elles nécessite la conversion préalable des entiers en réels. Il faut donc implémenter un calcul de la racine carrée d'un entier x, retournant soit la partie entière par défaut $\lfloor \sqrt{x} \rfloor$, soit la partie entière par excès $\lceil \sqrt{x} \rceil$, ceci sans quitter le type entier.

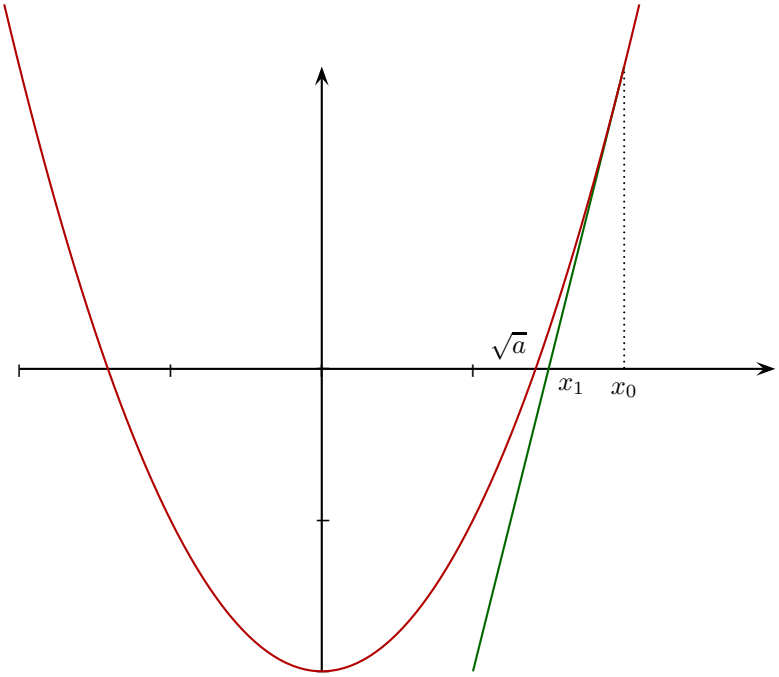

FIGURE 2 – Position de x_1

4. **Calcul de la racine carrée des grands entiers**

(a) Soit $a > 1$, et $f : x \mapsto x^2 - a$. Soit $x_0 > \sqrt{a}$, et x_1 l'abscisse du point d'intersection de la tangente à la courbe de f en x_0 et l'axe des abscisses. La convexité de la courbe de f (du fait de la positivité de sa dérivée seconde) montre que la courbe est au-dessus de la tangente en x_0. En particulier, le point de la tangente d'abscisse $\sqrt{a}$ est d'ordonnée négative. D'après le théorème des valeurs intermédiaires, l'unique point d'intersection de la tangente à la courbe de f en x_0 et l'axe des abscisses a une abscisse x_1 vérifiant $\sqrt{a} < x_1 < x_0$.
Les arguments de convexité n'étant pas au programme de Math Sup, on peut s'assurer par une étude de fonction de la position de la courbe par rapport à la tangente (étude de $g : x \mapsto x^2 - a - (2x_0(x - x_0) + x_0^2 - a)$).

(b) La tangente est d'équation $y = 2x_0(x - x_0) + x_0^2 - a$. On obtient donc :

$$x_1 = \frac{x_0^2 - a}{2x_0} = g(x_0),$$

où $g : x \mapsto \frac{x^2-a}{2x}$.

(c) D'après ce qui précède, si $r_n > \sqrt{a}$ et par définition de la partie entière :

$$r_{n+1} \leqslant g(r_n) < r_n.$$

Ainsi, si pour tout $n \in \mathbb{N}$, $r_n > \sqrt{a}$, (r_n) est une suite strictement décroissante d'entiers, et ne peut donc pas être minorée, d'où une contradiction. Ainsi, il existe n tel que $r_n \leqslant \sqrt{a}$.
Soit N le plus petit entier tel que $r_N \leqslant \sqrt{a}$. On a alors $r_{N-1} > a$ ($N > 0$, si $a > 1$, les cas $a = 0$ et $a = 1$ sont triviaux, et $N = 0$ convient) donc $\sqrt{a} < g(r_{N-1}) < r_{N-1}$. Comme par ailleurs $g_{r_{N-1}} < r_N + 1$, on en déduit que $r_N = \lfloor \sqrt{a} \rfloor$.

(d) On propose alors l'algorithme suivant :

```
def racine(n):
    x=n
    while x*x>n:
        x = (x*x+n)//(2*x)
    return(x)
```

Pour $n = 2 \cdot 10^{400}$, on obtient :

```
14142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415727350138462309122970249248360558507372126441214970999358314132226659275055927557999505011527820605714 7
```

En replaçant la virgule, cela nous donne les 200 premières décimales de $\sqrt{2}$!

(e) On peut améliorer un peu cet algorithme en partant d'une valeur initiale plus proche de $\sqrt{a}$. On peut remarquer que si k est le nombre de chiffres de a, alors $a < 10^k$, donc $\sqrt{a} < 10^{\lceil \frac{k}{2} \rceil}$. On peut donc initialiser le calcul avec cette valeur.

Par ailleurs, dans la fonction précédente, on effectue 2 fois le calcul de x^2. Comme on travaille avec des grands entiers, ce calcul peut être coûteux. On a donc intérêt à effectuer le calcul une seule fois en stockant le résultat dans une variable pour le réutilier ensuite. On obtient alors la version suivante :

```
def racine2(n):
    k = len(str(x))
    x = 10**(math.ceil(k / 2))
    c = x * x
    while c>n:
        x =  (c+n)//(2*x)
        c = x * x
    return(x)
```

(f) L'entier $\lceil \sqrt{x} \rceil$ est égal à $\lfloor \sqrt{x} \rfloor + 1$, sauf si x est entier. Ainsi, on peut définir la valeur entière par excès de la racine de la façon suivante :

```
def racinesup(n):
    r = racine2(n)
    if r * r == n:
        return(r)
    else:
        return(r+1)
```

Le test ne pose pas de problème d'exactitude ici, puisqu'on travaille avec des entiers.

5. (a) Le 2^n-gône dont le cercle unité est le cercle inscrit est obtenu du 2^n incrit dans le cercle par une homothétie de rapport $\frac{OC}{OH}$, c'est-à-dire $\dfrac{1}{\sqrt{1-\frac{u_n^2}{4}}}$. Les deux périmètres diffèrent donc du même facteur, et encadrent le périmètre du cercle, à savoir 2π. On en déduit l'encadrement demandé :

$$2^{n-1}u_n \leqslant \pi \leqslant 2^{n-1}\frac{u_n}{\sqrt{1-\frac{u_n^2}{4}}} = 2^n \frac{u_n}{\sqrt{4-u_n^2}}.$$

(b) Pour la minoration, on utilise v_n, qui minore $10^{400}u_n^2$. Pour la majoration, on utilise w_n, qui majore $10^{400}u_n^2$. On obtient sans problème cet encadrement, dont on sait calculer chacun des termes par les méthodes précédentes, ou par utilisation de la division euclidienne :

$$2^{n-1}\left\lfloor \sqrt{v_n} \right\rfloor \leqslant 10^{200}\pi \leqslant \frac{2^n \left\lceil \sqrt{w_n} \right\rceil}{\sqrt{4\cdot 10^{400}-w_n}} \leqslant \left\lfloor \frac{2^n \left\lceil \sqrt{w_n} \right\rceil}{\sqrt{4\cdot 10^{400}-w_n}} \right\rfloor + 1.$$

L'implémentation du calcul se fait alors ainsi (la fonction `encadrementpi()` répond à la question) :

```
def calcul_v2(n):
    v=4*(10**400)
    for i in range(n-1):
        v=2*(10**400)-racinesup(4*(10**800)-(10**400)*v)
    return v

def calcul_w2(n):
    w=4*(10**400)
    for i in range(n-1):
        w=2*(10**400)-racine(4*(10**800)-(10**400)*w)
    return w
```

```
def archimede3(n):
    return racine(calcul_v2(n))* 2**(n-1)

def convertitsup(x,n):
    return (racinesup(x) * 2**(n-1) * 10**200 * 2)// racine(4 * 10**400-x) + 1

def archimedesup(n):
    return convertitsup(calcul_w2(n),n)

def encadrementpi(n):
    return archimede3(n),archimedesup(n)
```

(c) Pour obtenir le rang qui nous donne le meilleur encadrement, on pourrait itérer le calcul, à l'aide des fonctions précédentes, en utilisant les fonctions précédentes, mais cette itération nous ferait reprendre le calcul du début à chaque étape :

```
def meilleureapprox():
    n = 1
    minorant = archimede3(n)
    majorant = archimedesup(n)
    erreur = majorant - minorant
    meilleur = minorant
    rg = 1
    while minorant != 0:
        if majorant - minorant < erreur:
            erreur = majorant - minorant
            meilleur = minorant
            rg = n
        n += 1
        minorant = archimede3(n)
        majorant = archimedesup(n)
    return minorant, erreur, rg
```

L'implémentation de cette méthode ne s'avère pas satisfaisant, le temps de réponse étant trop long (je n'ai personnellement pas attendu la fin du calcul). On refait donc le calcul de v_n et w_n au sein de l'itération. Cela nous donne la fonction suivante :

```
def meilleureapprox2():
    v=4*(10**400)
    w=4*(10**400)
    n = 1
    minorant = 2 * 10 ** 200
    majorant = 4 * 10 ** 200
    erreur = majorant - minorant
    meilleur = minorant
    rg = 1
    while v != 0:
        if majorant - minorant < erreur:
            erreur = majorant - minorant
            meilleur = minorant
            rg = n
        n += 1
        v=2*(10**400)-racinesup(4*(10**800)-(10**400)*v)
        w=2*(10**400)-racine(4*(10**800)-(10**400)*w)
        minorant = racine(v) * 2 **(n-1)
        majorant = convertitsup(w,n)
    return meilleur, erreur, rg
```

On trouve la meilleure approximation au rang 223

(d) On peut extraire les décimales correctes obtenues, en comparant, pour le rang optimal ci-dessus, les décimales du minorant et les décimales du majorant, jusqu'à la première différence :

```
def decimales_correctes():
```

```
    minorant, erreur, rg = meilleureapprox2()
    majorant = minorant + erreur
    ch_min = str(minorant)
    ch_maj = str(majorant)
    print(minorant, majorant)
    i = 0
    while ch_min[i] == ch_maj[i]:
        i += 1
    return ch_min[:i], rg

approxpi, rg = decimales_correctes()
lg = len(str(approxpi))
print('La meilleure approximation de pi est obtenue au rang {}'.format(rg))
print('on obtient {} chiffres corrects de pi:\n{}'.format(lg,approxpi))
```

On obtient la réponse suivante :

```
La meilleure approximation de pi est obtenue au rang 223
on obtient 133 chiffres corrects de pi:
3141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067982148086513282306647093844609550
```

On peut bien sûr augmenter la précision en travaillant avec des entiers plus longs. Cela dit, cette méthode nécessite des calculs assez lourds, augmentant encore avec la taille des entiers utilisés. Il existe des méthodes de calcul de π beaucoup plus efficaces.

Correction de l'exercice 3 –

1. Je vous laisse le soin de faire le calcul manuel de la racine de 64015.
2. Toutes les boucles sont finies (nombre d'itérations maximal connu à l'avance), donc l'algorithme se termine.
   Pour l'étude de sa correction, on prodède par récurrence par tranches. Soit k le nombre de tranches dans la décomposition de n. On appelle n_i le nombre représenté par les n premières tranches, d_i la racine provisoire trouvée et r_i le reste exprimé. On note t_i la i-ième tranche. On montre que pour tout $i \in [\![0,k]\!]$, on a $d_i^2 \leqslant n_i < (d_i+1)^2$, et $n_i = d_i^2 + r_i$.
   Pour $i=0$, la propriété est triviale, car $n_0 = d_0 = r_0 = 0$.
   Supposons la propriété vraie pour $i \in [\![0, k-1]\!]$. Pour commencer, en multipliant l'encadrement de n_i par 100, on obtient $(10d_i)^2 \leqslant 100n_i$ d'un côté, et puisqu'on est en entiers, $n_i+1 \leqslant (d_i+1)^2$, donc $100n_i+100 \leqslant (10d_i+1O)^2$. On en déduit que :
   $$(10d_i)^2 \leqslant 100n_i \leqslant n_{i+1} < 100n_i + 100 \leqslant (10d_i+10)^2.$$
   Ainsi, l'unique entier d_{i+1} tel que $d_{i+1}^2 \leqslant n_{i+1} < (d_{i+1}+1)^2$ s'écrit bien sous la forme $10d_i + j$, où $j \in [\![0,9]\!]$. Cet entier i vérifie alors :
   $$(10d_i+j)^2 \leqslant n_{i+1} < (10d_i+j+1)^2,$$
   soit, en développant :
   $$100(d_i^2 - n_i) + (20d_i+j)\times j \leqslant t_{i+1} < 100(d_i^2-n_i) + (20d_i+j+1)\times(j+1),$$
   ou encore :
   $$(20d_i+j)\times j \leqslant r_i + t_{i+1} < (20d_i + (j+1))\times(j+1)$$
   L'entier j vérifiant cet encadrement est unique, et c'est bien celui déterminé dans l'algorithme, puisque $r_i + t_{i+1}$ correspond au nombre obtenu après abaissement de la $i+1$-ième tranche derrière le i-ième reste. Ainsi, par définition le nombre obtenu en ajoutant j aux chiffres de d_i est d_{i+1}, donc $d_{i+1} = 10d_i + j$, qui fournit bien l'encadrement requis de n_{i+1} :
   $$d_{i+1}^2 \leqslant n_{i+1} < (d_{i+1}+1)^2.$$
   Par ailleurs, le reste r_{i+1} est défini par l'algorithme par
   $$r_{i+1} = (100r_i+t_i) - (20d_i+j)j = (100r_i+t_i) + 100d_i^2 - (10d_i+j)^2 = 100(r_i+d_i^2) + t_i - d_{i+1}^2 = 100n_i + t_i - d_{i+1}^2 = n_{i+1} - d_{i+1}^2.$$
   On obtient bien la relation $n_{i+1} = d_{i+1}^2 + r_i$
   Ainsi, d'après le principe de récurrence, pour $i=k$, on obtient bien la validité de l'algorithme.
   L'implémentation est faite ci-dessous :

```python
def extractionracine(n):
    d=0
    liste_tranches=[]
    while n != 0:
        (n,r)=divmod(n,100)
        liste_tranches.append(r)
    liste_tranches.reverse()
    but=0
    for tranche in liste_tranches:
        but = 100 * but + tranche
        d = 10 * d
        i = 9
        while (2 * d + i) * i > but:
            i -= 1
        but = but - (2 * d + i) * i
        d = d + i
    return(d)
```

3. On obtient par exemple les 1000 premières décimales de $\sqrt{2}$ en appliquant cet algorithme à 2×10^{1000}. On peut faire un petit travail sur les chaînes de caractères pour représenter le résultat sous forme d'une chaîne de caractères permettant ainsi de replacer la virgule au bon endroit :

```python
resultat = str(extractionracine(2 * (10 ** 1000)))
racinede2 = resultat[0] + '.' + resultat[1:]
print(racinede2)
```

Et pour le plaisir, voilà la réponse :

```
1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885
503875343276415727350138462309122970249248360558507372126441214970999358314132226659
275055927557999505011527820605714701095599716059702745345968620147285174186408891986
095523292304843087143214508397626036279952514079896872533965463318088296406206152583
523950547457502877599617298355752203375318570113543746034084988471603868999706990048
150305440277903164542478230684929369186215805784631115966687130130156185689872372
```