Approx_Racines

December 16, 2018

1 Approximation des racines d'une équation

Marc Lorenzi - 28 avril 2018

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline
        from math import *

In [2]: plt.rcParams['figure.figsize'] = (8, 8)
```

1.1 1. Résolution par dichotomie

Soit $f : [a, b] \to \mathbb{R}$ une fonction continue telle que que $f(a)$ et $f(b)$ sont de signes opposés. Le théorème des valeurs intermédiaires affirme qu'il existe $\xi \in [a, b]$ tel que $f(\xi) = 0$.

Posons $a_0 = a$ et $b_0 = b$. Soit $c = \frac{a+b}{2}$. Si $f(a)$ et $f(c)$ sont de signes contraires alors il existe une racine de f sur $[a, c]$. Posons alors $a_1 = a_0$ et $b_1 = c$. Sinon, il existe une racine de f sur $[c, b]$ et on pose $a_1 = c$ et $b_1 = b_0$. La fonction f possède donc une racine ξ_1 sur l'intervalle $[a_1, b_1]$ et $f(a_1)$ et $f(b_1)$ sont de signes contraires.

On peut évidemment recommencer à loisir cette opération ... On définit ainsi une suite $[a_n, b_n]$ de segments emboîtés tels que f possède une racine ξ_n sur l'intervalle $[a_n, b_n]$ et $f(a_n)$ et $f(b_n)$ sont de signes contraires.

De plus, $b_{n+1} - a_{n+1} = \frac{1}{2}(b_n - a_n)$ donc $b_n - a_n = \frac{1}{2^n}(b_0 - a_0)$. La longueur des segments tend vers 0 lorsque n tend vers l'infini. Par le théorème des segments emboîtés, les suites (a_n) et (b_n) sont adjacentes. Notons ξ leur limite commune. Pour tout n, $a_n \leq \xi_n \leq b_n$ donc, par le théorème d'encadrement des limites, ξ_n tend vers ξ. Or, $f(\xi_n) = 0$ et f est continue en ξ. Donc $f(\xi) = 0$: ξ est une racine de f.

La fonction `racine` prend en paramètres une fonction f, deux réels a et b et un réel $\varepsilon > 0$. Elle suppose que f est continue sur $[a, b]$ et que $f(a)$ et $f(b)$ sont de signes contraires. Tant que $|b - a| > \varepsilon$, elle coupe l'intervalle $[a, b]$ en son milieu c et elle recommence sur le demi-intervalle adéquat.

La fonction renvoie une approximation d'une racine de f ainsi que le nombre d'itérations effectuées pour la trouver.

```
In [3]: def racine(f, a, b, epsilon):
            c = (a + b) / 2
            cpt = 0
            while abs(b - a) > epsilon:
```

```
        cpt = cpt + 1
        c = (a + b) / 2
        if f(a) * f(c) <= 0: b = c
        else: a = c
    return (c, cpt)
```

Testons sur $f : x \mapsto x^2 - 2$ entre 1 et 2. Ce sera notre exemple fétiche jusqu'à la fin du notebook mais rien ne vous empêche d'essayer plein d'autres fonctions.

```
In [4]: racine(lambda x: x ** 2 - 2, 1, 2, 1e-15)
```

```
Out[4]: (1.414213562373095, 50)
```

Pourquoi 50 itérations ? Eh bien, à la k-ième itération, la longueur de l'intervalle $[a_n, b_n]$ est $\frac{b-a}{2^k}$. Le nombre d'itérations est donc le plus petit entier k tel que $\frac{b-a}{2^k} \leq \varepsilon$, ou encore $k \geq \lg \frac{b-a}{\varepsilon}$, lg étant le logarithme en base 2.

```
In [5]: log(1 / 10 ** (-15)) / log(2)
```

```
Out[5]: 49.82892142331043
```

Effectivement, 50 est la bonne valeur.
Quelques remarques pour terminer :

1. La fonction renvoie **une** racine de f, en fin une approximation. Quelle racine ? On ne sait pas ...
2. La méthode par dichotomie possède une propriété infiniment précieuse : elle renvoie un **encadrement** d'une racine (à condition de renvoyer (a, b) et pas c, évidemment). Ainsi, on maîtrise l'erreur commise.

Exercice : Soit $f : [a, b] \to \mathbb{R}$ continue ayant une racine. Soit $E = \{x \in [a, b], f(x) = 0\}$. Montrer que E a un plus petit élément.

1.2 2. Points fixes d'une fonction

1.2.1 2.1 Introduction

Soit $f : I \to I$ où I est un intervalle de $\mathbb{R}$. Un **point fixe** de f est un réel $x \in I$ tel que $f(x) = x$. Donnons nous $x_0 \in I$ et considérons la suite $(x_n)_{n\geq 0}$ définie par récurrence par $x_{n+1} = f(x_n)$. Sous certaines conditions, lorsque n tend vers l'infini x_n tend vers un point fixe de f. Nous expliciterons quelques conditions suffisantes dans ce qui va suivre.

1.2.2 2.2 Recherche d'un point fixe

La fonction `point_fixe` prend en paramètres une fonction f, un réel x_0 et un réel $\varepsilon > 0$. Tant que $|f(x) - x| > \varepsilon$, elle remplace x par $f(x)$. Toutefois, la fonction abandonne après 100 itérations, même si la condition d'arrêt n'est pas remplie.

Ce test d'arrêt est plus que discutable, voire absurde. Mais faute de mieux on s'en contentera. Il est bon toutefois de noter que, sous certaines conditions, ce test est cohérent.

```
In [6]: def point_fixe(f, x0, epsilon):
            x = x0
            cpt = 0
            while abs(f(x) - x) > epsilon:
                x = f(x)
                cpt = cpt + 1
                if cpt >= 100: break
            return (x, cpt)

In [7]: point_fixe(lambda x: cos(x), 0, 1e-15)

Out[7]: (0.7390851332151611, 87)
```

Un point fixe de la fonction cosinus est atteint "à 10^{-15} près" en 87 itérations.

1.2.3 2.3 Visualiser les itérations

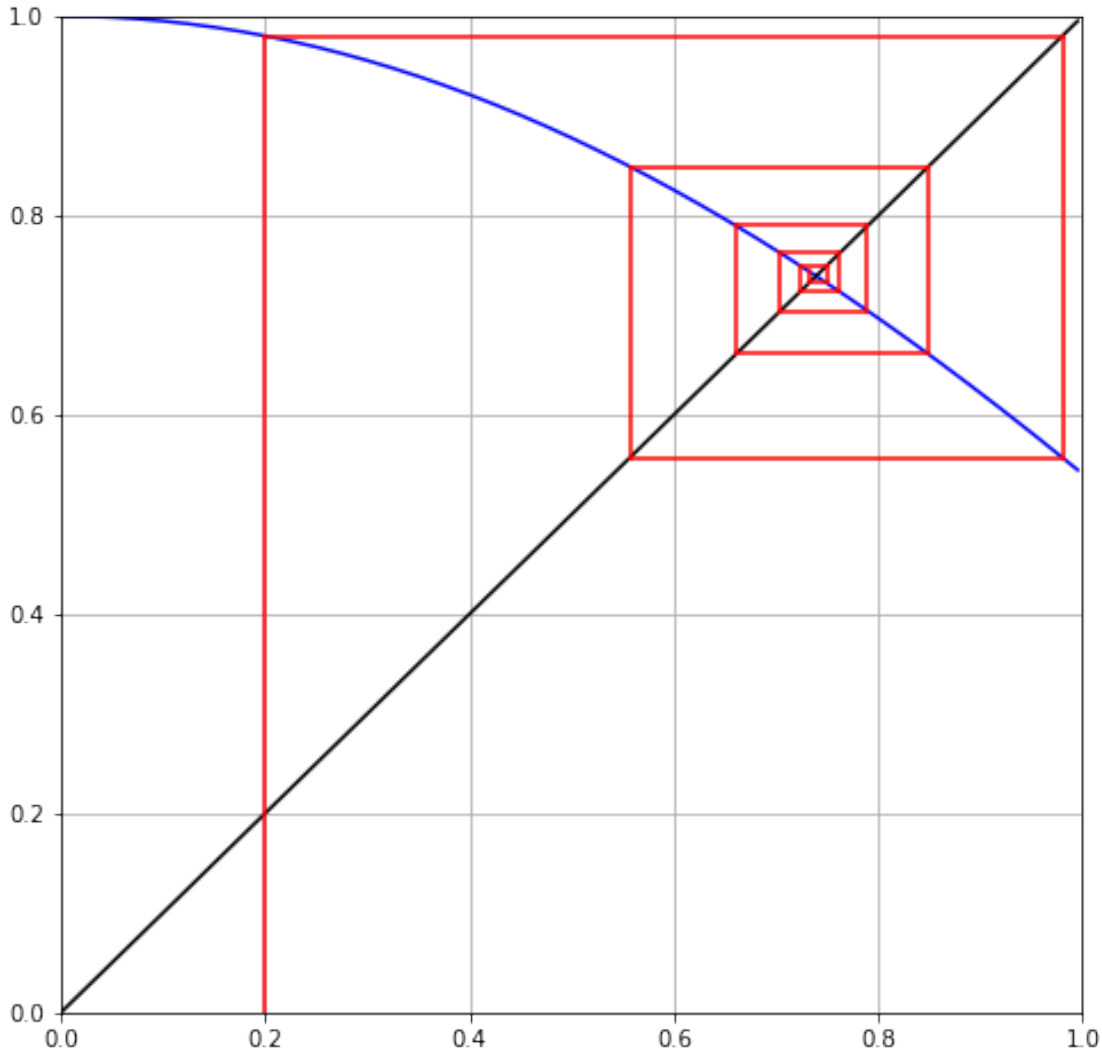
Petite fonction renvoyant une subdivision régulière du segment $[a, b]$. Bien pratique.

```
In [8]: def subdivision(a, b, n):
            d = (b - a) / n
            return [a + k * d for k in range(n)]
```

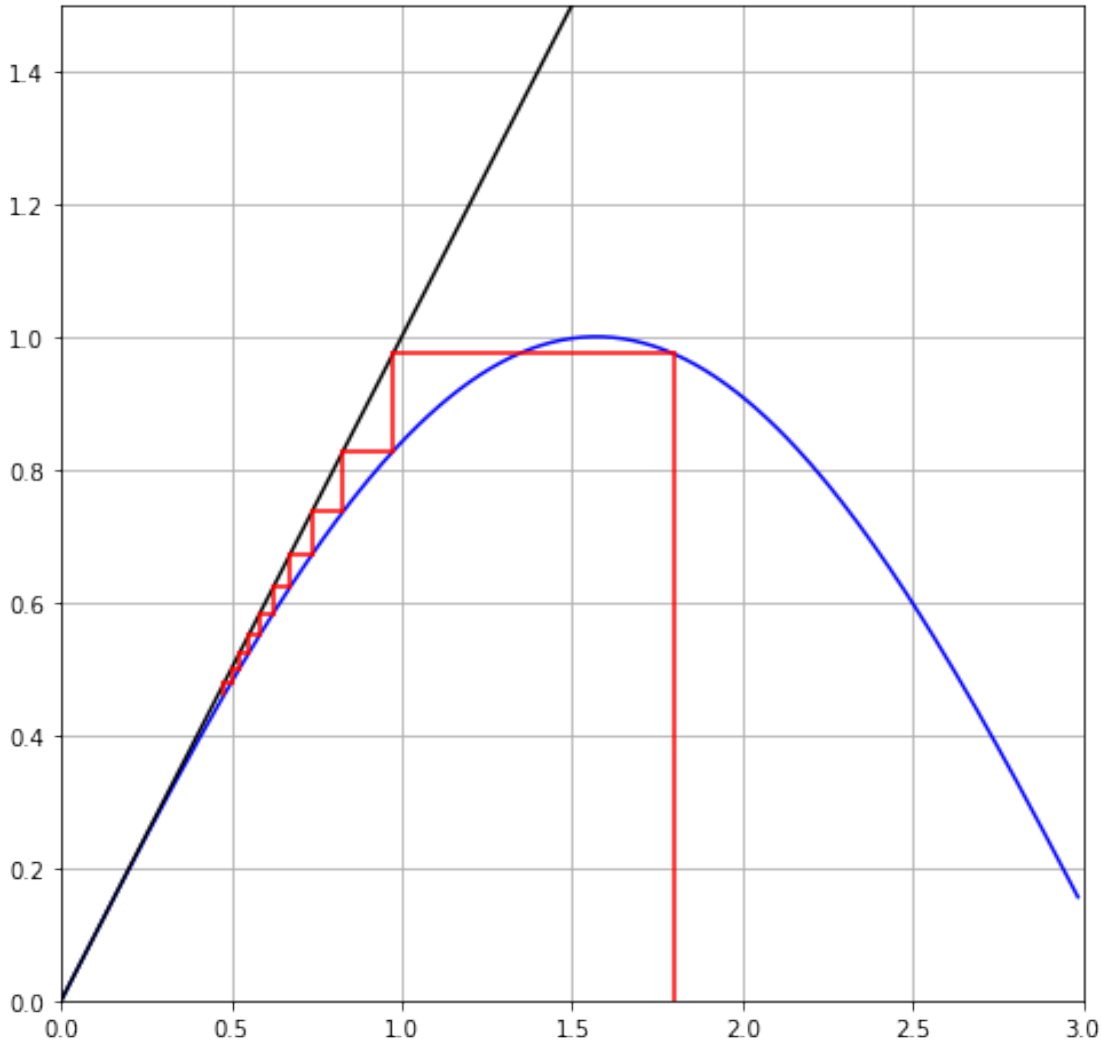
La fonction `escalier` dessine la courbe d'une fonction f sur le segment $[a, b]$, ainsi que la droite d'équation $y = x$. Puis elle visualise les différents termes de la suite de premier terme x_0 définie par $x_{n+1} = f(x_n)$. Vite écrite, mal écrite. On peut clairement améliorer la présentation.

```
In [9]: def escalier(f, a, b, c, d, x0, n):
            xs = subdivision(a, b, 200)
            ys = [f(x) for x in xs]
            pad = 0.
            plt.axis([a - pad, b + pad, c - pad, d + pad])
            plt.plot(xs, ys, 'b')
            plt.plot(xs, xs, 'k')
            x = x0
            coul = 'r'
            plt.plot([x0, x0], [0, f(x0)], coul)
            for k in range(n):
                plt.plot([x, f(x)], [f(x), f(x)], coul)
                plt.plot([f(x), f(x)], [f(x), f(f(x))], coul)
                x = f(x)
            plt.grid()
            plt.show()

In [10]: escalier(cos, 0, 1, 0, 1, 0.2, 10)
```




```
In [11]: escalier(lambda x: sin(x), 0, 3, 0, 1.5, 1.8, 10)
```



1.2.4 2.4 Une condition suffisante

À quelles conditions (suffisantes mais non triviales) une fonction possède-t-elle un point fixe ? L'ensemble I ci-dessous est un intervalle fermé de $\mathbb{R}$.

Proposition : Soit $f : I \to I$ une application k-lispchitzienne où $k \in [0,1[$. Alors f admet un unique point fixe ℓ. Mieux, pour tout $x_0 \in I$, la suite définie par $x_{n+1} = f(x_n)$ converge vers le point fixe ℓ.

Démonstration : Soient ℓ_1 et ℓ_2 deux points fixes de f. On a $|\ell_1 - \ell_2| = |f(\ell_1) - f(\ell_2)| \leq k|\ell_1 - \ell_2|$ d'où $(1-k)|\ell_1 - \ell_2| \leq 0$. Mais $1 - k > 0$, donc $|\ell_1 - \ell_2| \leq 0$. D'où $\ell_1 = \ell_2$ et l'unicité.

Passons à l'existence. On a pour tout entier $n \geq 1$, $|x_{n+1} - x_n| = |f(x_n) - f(x_{n-1})| \leq k|x_n - x_{n-1}|$. On déduit par une récurrence facile que pour tout $n \geq 0$, $|x_{n+1} - x_n| \leq k^n|x_1 - x_0|$. Le terme général de la série $\sum(x_{n+1} - x_n)$ est donc majoré en valeur absolue par le terme général d'une série géométrique convergente. Ainsi, la série $\sum(x_{n+1} - x_n)$ est absolument convergente, donc convergente. Mais la $(n-1)$ième somme partielle de cette série est $S_{n-1} = \sum_{k=0}^{n-1}(x_{k+1} - x_k) =$

$x_n - x_0$ par téléscopage. On en déduit que x_n converge vers un réel ℓ lorsque n tend vers l'infini. Comme l'intervalle I est fermé, $\ell \in I$. Prenons l'égalité $x_{n+1} = f(x_n)$, valable pour tout entier n. Un passage à la limite (f est lipschitzienne sur I, donc continue en ℓ) donne $\ell = f(\ell)$. On a donc l'existence et la limite adéquate.

Remarque : On a pour tout entier $n \geq 1$, $|x_{n+1} - \ell| = |f(x_n) - f(\ell)| \leq k|x_n - \ell|$. On déduit par une récurrence facile que pour tout $n \geq 0$, $|x_n - \ell| \leq k^n|x_0 - \ell|$. On a donc une majoration explicite de l'écart entre le $n$ième terme de la suite, x_n, et sa limite ℓ. On constate également que plus le coefficient de Lischitz k est petit, plus la suite converge vite vers sa limite.

1.2.5 2.5 Application à la recherche de racines

Soit $f : I \to \mathbb{R}$. Pour trouver une racine de f, une idée consiste à considérer la fonction $g : x \mapsto x - f(x)$. En effet, pour tout $x \in I$, on a $f(x) = 0$ si et seulement si $g(x) = x$. Pour des raisons qui apparaîtront plus tard, on va plutôt prendre un réel μ et s'intéresser à la fonction $g : x \mapsto x - \mu f(x)$. En effet, le paragraphe précédent nous dit qu'en abaissant le coefficient de Lispchitz de g la suite (x_n) tendra plus vite vers sa limite. En glissant un paramètre dans la fonciton g, on peut peut-être maîtriser le coefficient de Lipschitz.

Regardons un peu ce qui se passe avec différentes valeurs de μ. On va s'apercevoir que pour certains μ il y a divergence de l'algorithme, et pour d'autres convergence. De plus, selon le choix de μ, la convergence est plus ou moins rapide.

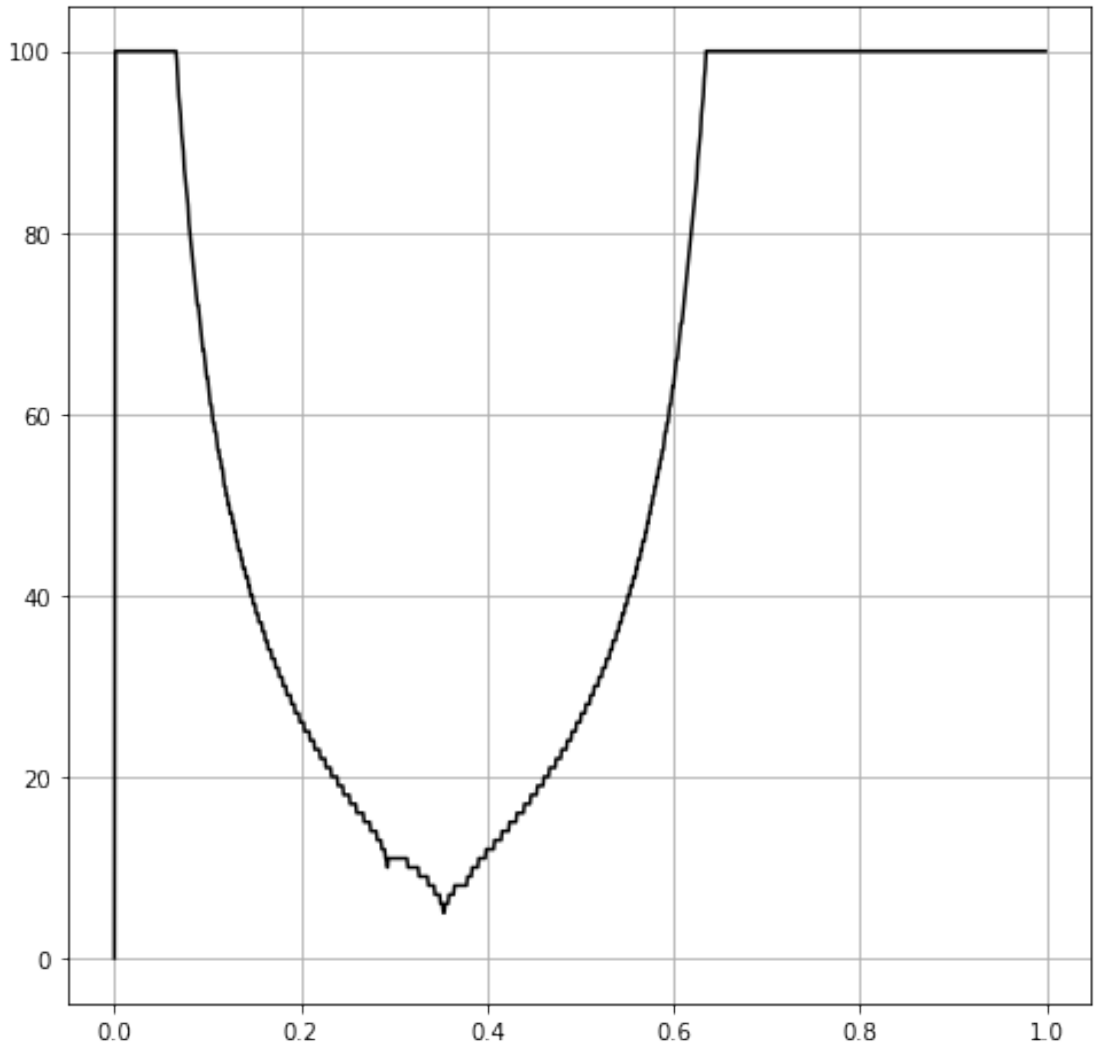
```
In [12]: def racine2(f, x0, mu, epsilon):
             g = lambda x: x - mu * f(x)
             return point_fixe(g, x0, epsilon)

In [13]: racine2(lambda x: x ** 2 - 2, 2, 0.2, 1e-10)

Out[13]: (1.4142135625141385, 26)
```

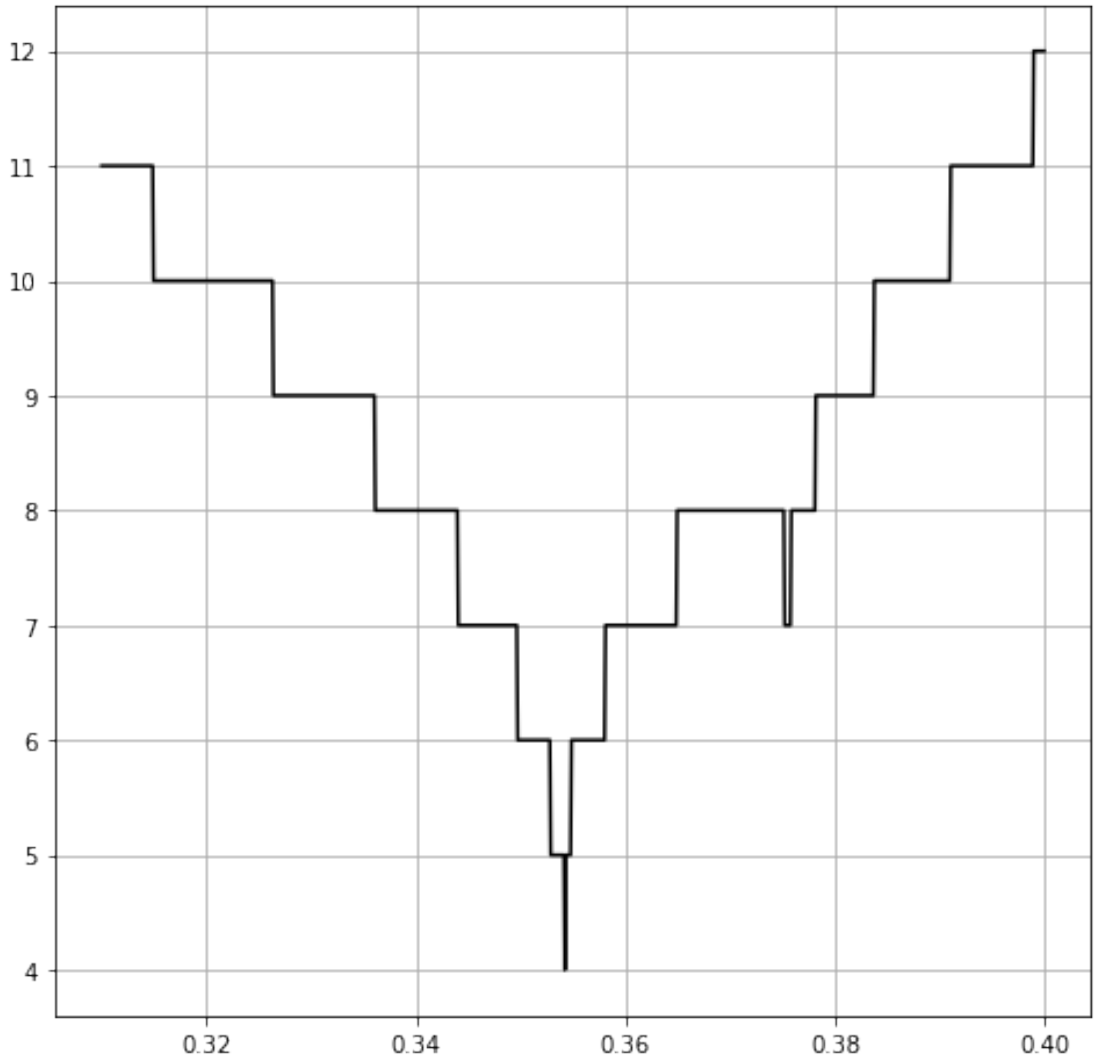
Prenons des valeurs de μ entre -1 et 1 et regardons le nombre d'itérations nécessaires à la terminaison de notre algorithme.

```
In [14]: s = subdivision(0, 1, 1000)
         cs = [racine2(lambda x: x ** 2 - 2, 2, mu, 1e-10)[1] for mu in s]
         plt.plot(s, cs, 'k')
         plt.grid()
         plt.show()
```



Trop beau. Faisons un zoom dans la zone ou le nombre d'itérations est minimal ...

```
In [15]: s = subdivision(0.31, 0.4, 1000)
         cs = [racine2(lambda x: x ** 2 - 2, 2, mu, 1e-10)[1] for mu in s]
         plt.plot(s, cs, 'k')
         plt.grid()
         plt.show()
```



Une valeur de μ quelque part entre 0.35 et 0.36 a l'air optimale. Il s'avère que cette valeur est $\frac{1}{2\sqrt{2}}$. Mais pourquoi ?

```
In [16]: racine2(lambda x: x ** 2 - 2, 2, 1 / (2 * sqrt(2)), 1e-10)
```

```
Out[16]: (1.4142135623406868, 4)
```

L'inégalité des accroissements finis nous dit que si $|g'|$ est majorée par un réel k alors g est k-lipschitzienne. Pour accélérer la convergence de nos suites il suffit donc de minimiser g', au moins au voisinage de ℓ. Si on prend $g'(\ell) = 0$, $|g'|$ sera k-lipschitzienne avec k aussi petit qu'on veut pourvu qu'on se restreingne à un voisinage suffisamment petit de ℓ.

Pour tout réel x, $g'(x) = 1 - \mu f'(x)$. Le réel $\mu = \frac{1}{f'(\ell)}$ paraît donc très prometteur.

Dans l'exemple que nous avons regardé, $f(x) = x^2 - 2$, $\ell = \sqrt{2}$ et, justement, $f'(\ell) = \frac{1}{2\sqrt{2}}$!

Il semblerait que l'on approche du Graal : la suite à considérer serait donc $x_{n+1} = g(x_n)$ où $g(x) = x - \frac{f(x)}{f'(\ell)}$, ℓ étant la limite de la suite. Euh oui, mais on a un gros souci : on ne connaît pas ℓ ! C'est toujours comme ça avec le Graal, chaque fois qu'on croit le tenir il nous échappe :-).

Soyons rusés, en prenant un paramètre μ variable qui tend vers $\frac{1}{f'(\ell)}$ lorsque n tend vers l'infini. Lequel ? eh bien $\mu = \frac{1}{f'(x_n)}$. LA suite à étudier est donc la suite

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Il s'avère que cette suite est justement à la base d'un algorithme connu depuis des siècles, l'algorithme de Newton-Raphson. L'idée a été ébauchée par Newton et formalisée par Raphson. C'est le but du prochain paragraphe.

1.3 4. L'algorithme de Newton-Raphson

1.3.1 4.1 Dérivation approchée

Voici une fonction renvoyant la valeur approchée de la dérivée d'une fonction f en un point x.

```
In [17]: def deriv(f, x, h=1e-5):
             return (f(x + h) - f(x - h)) / (2 * h)

In [18]: deriv(lambda x: exp(x), 0)

Out[18]: 1.0000000000121023
```

1.3.2 4.2 La méthode de Newton

Soit $f : I \to \mathbb{R}$ une fonction dérivable en point $x \in I$. L'équation de la tangente à la courbe représentative de f au point $A = (x, f(x))$ est

$$(T)Y = f(x) + (X - x)f'(x)$$

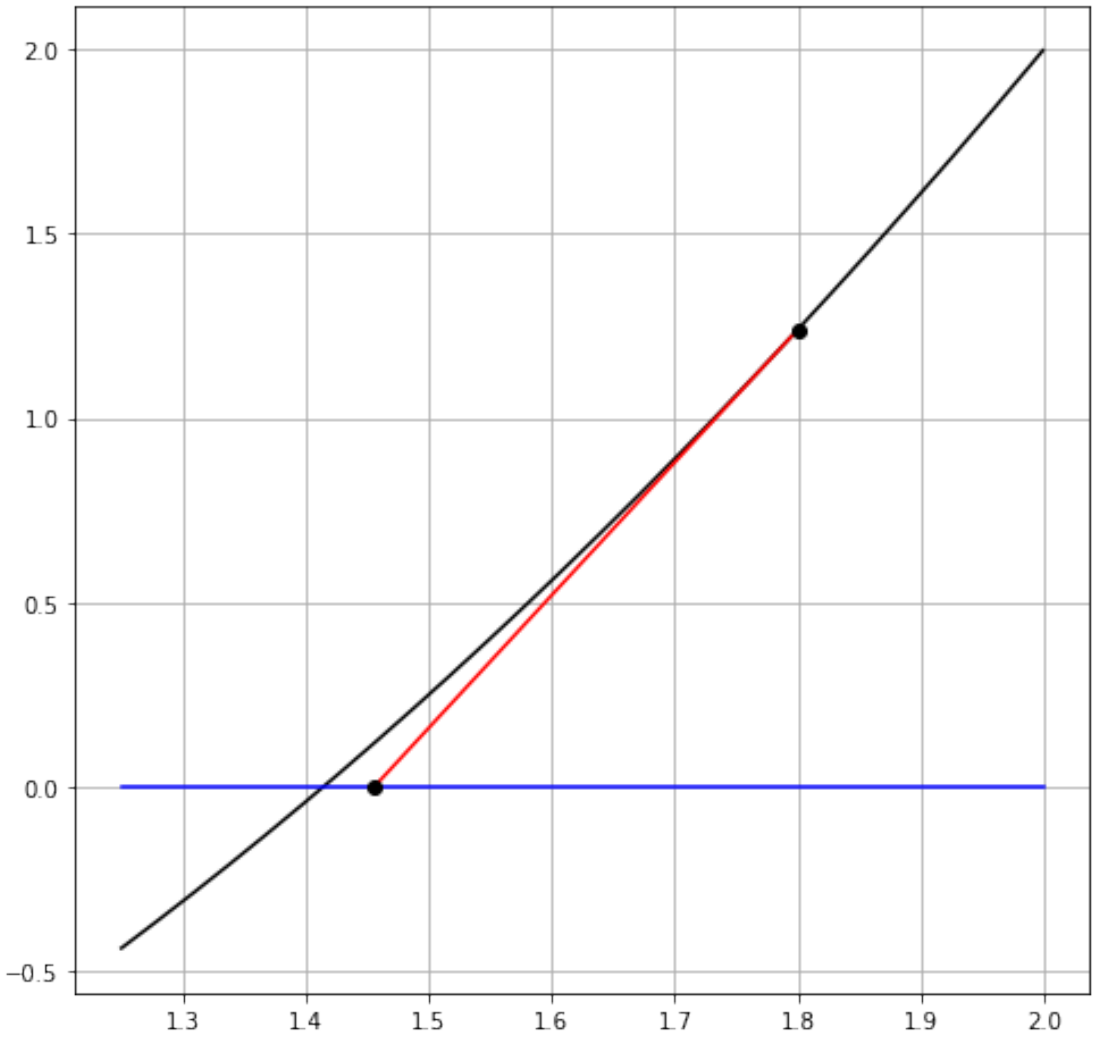
Quelle est l'intersection de T et de l'axe Ox ? Il suffit d'annuler Y dans l'équation de T. Il vient

$$X = x - \frac{f(x)}{f'(x)}$$

Surprise ... La suite récurrente que nous nous proposons d'étudier a donc une interprétation géométrique simple. Pour passer de x_n à x_{n+1}, on trace la tangente à la courbe au point d'abscisse x_n. L'intersection de cette tangente avec l'axe Ox a alors pour abscisse x_{n+1}.

Voici un dessin simpliste.

```
In [19]: xs = subdivision(1.25, 2, 1000)
         f = lambda x: x ** 2 - 2
         ys = [f(x) for x in xs]
         plt.plot(xs, ys, 'k')
         plt.plot(xs, [0 for x in xs], 'b')
         x = 1.8
         x1 = x - f(x) / deriv(f, x)
         plt.plot([x, x1], [f(x), 0], 'r')
         plt.plot([x],[f(x)],'ko')
         plt.plot([x1],[0],'ko')
         plt.grid()
         plt.show()
```



Newton se contentait de prendre pour valeur approchée de la racine de f l'intersection du segment bleu et du segment rouge. Nous devons à Raphson l'idée que l'on pouvait recommencer avec cette nouvelle valeur. Enfin, oui, j'avoue, Newton a quand même un peu itéré aussi :-)

1.3.3 4.3 L'algorithme

Il est immédiat. Il n'y a qu'à utiliser ce que nous avons déjà fait.

```
In [20]: def newton(f, x0, epsilon):
             g = lambda x: x - f(x) / deriv(f, x)
             return point_fixe(g, x0, epsilon)
```

Voici $\sqrt{2}$ en 4 itérations avec 10 chiffres après la virgule.

```
In [21]: newton(lambda x: x ** 2 - 2, 2, 1e-10 )
```

```
Out[21]: (1.4142135623746899, 4)
```

Et voici π en 3 itérations avec 15 chiffres.

```
In [22]: newton(lambda x:sin(x), 3, 1e-15)
```

```
Out[22]: (3.141592653589793, 3)
```

Pour bien mesurer à quel point la méthode de Newton est efficace, travailler avec 10 chiffres après la virgule ne suffit pas.

```
In [23]: from sympy import *
```

Calculons $\sqrt{2}\ldots$ avec 2000 chiffres après la virugule ?

```
In [24]: prec = 2000
         newton(lambda x: x ** 2 - 2, N(2, prec + 1), S(10) ** (- prec))
```

```
Out[24]: (1.414213562373095048801688724209698078569671875376948073176679737990732478462107038850
          12)
```

Eh oui. En 12 itérations. L'algorithme de Newton-Raphson est TRÈS performant. Vous n'avez pas confiance, vous n'y croyez pas ? Comparons le résultat r obtenu par Newton à une valeur approchée de $\sqrt{2}$, x, demandée à `sympy`.

```
In [25]: prec = 2000
         r, c  = newton(lambda x: x ** 2 - 2, N(2, prec + 1), S(10) ** (- prec))
         x = N(sqrt(2), prec)
         print(r - x)
```

```
-5.65703448066293155378268732104691836995986456277450895432770655136455707114733344047036961544
```

Demandez 10000 chiffres si cela vous tente ... Il faudra peut-être attendre 1 ou 2 secondes mais c'est normal. Faire une simple division avec autant de chiffres prend quand même un certain temps, et `sympy` n'a pas été construit pour cela. Je l'ai lancé avec 100000, mais je n'ai pas osé le million.

1.3.4 4.4 Et sur $\mathbb{C}$?

Est-ce que la méthode de Newton permet de trouver une racine **non réelle** de $x^3 - 1$?

Exécutons `newton` avec $x_0 = i$ ...

```
In [26]: newton(lambda x: x ** 3 - 1, 0+1J, 1e-15)
```

```
Out[26]: ((-0.5+0.8660254037844387j), 6)
```

Super, ça converge vers j, la racine cubique de 1.

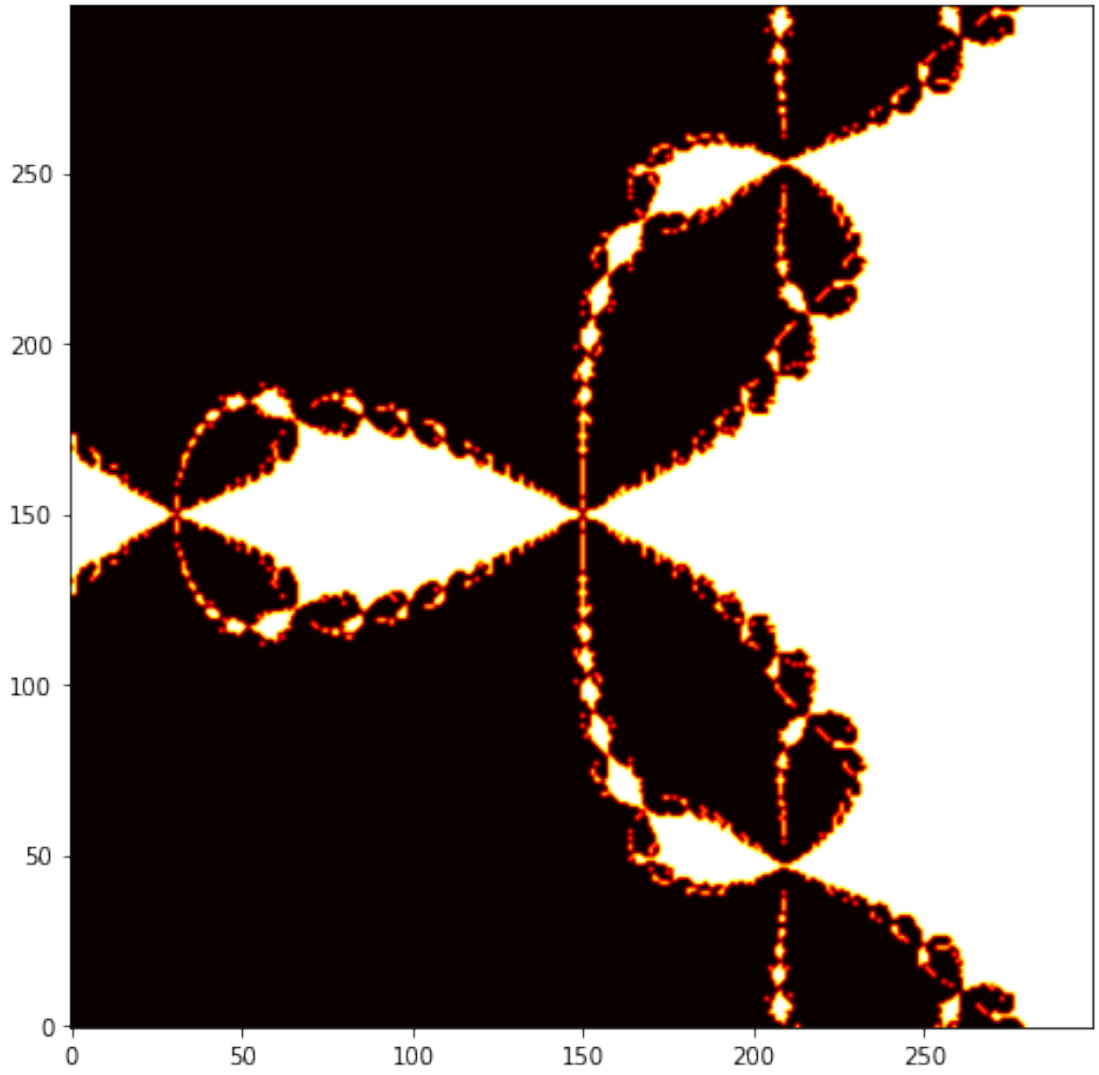
Maintenant ouvrons la boîte de Pandore. Il y a TROIS racines cubiques de 1. Quelles sont les valeurs de x_0 pour lesquelles la suite converge vers 1 ? Vers j ? Vers j^2 ? Ci-dessous, on affiche en blanc les points $x_0 \in \mathbb{C}$ du carré $[-1,1] \times [-1,1]$ tels que la suite converge vers 1. Remarquer que pour $f(x) = x^3$, on a $x - \frac{f(x)}{f'(x)} = \frac{2x^3+1}{3x^2}$.

Ne vous inquiétez pas pour la ligne ci-dessous. Vous me remercierez si vous faites l'exercice posé un peu plus bas ...

In [27]:
```
from math import sqrt
```

In [28]:
```
def attracteur():
    N = 300
    n = 3
    f = lambda w : ((n - 1) * w ** n + 1) / (n * w ** (n - 1))
    #f = lambda w: ((2 * w ** 3 + 1) / (3 * w ** 2))
    A = N * [None]
    for i in range(N): A[i] = N * [0]
    for i in range(N):
        for j in range(N):
            x = -1 + 2 * i / N
            y = -1 + 2 * j / N
            z = x + y * 1j
            for k in range(32):
                if abs(z) > 1e-10: z = f(z)
            if abs(z - 1) < 1e-10: A[j][i] = 1
    plt.imshow(A, interpolation='bicubic', cmap='hot', origin='lower')
    plt.savefig('newton.png')
```

In [29]:
```
attracteur()
```



Exercice : Modifier 1 ligne dans le code ci-dessus pour afficher en blanc les points pour lesquels la suite converge vers j.

L'ensemble cherché est **TRÈS** compliqué, donc très intéressant. Imitons Pandore et refermons la boîte en y laissant l'espérance ... qu'un jour nous pourrons comprendre ce dessin.

1.3.5 4.5 La théorie

4.5.1 Hypothèses Plaçons nous dans un cas particulier : $f : I \to \mathbb{R}$ est de classe $\mathcal{C}^2$. $\ell \in I$ est une racine de f et n'est pas la borne supérieure de I. On suppose $f'' > 0$, $f'(\ell) > 0$. Et on prend $x_0 > \ell$. Ça a l'air très particulier et ça l'est ... mais l'exposé sera plus clair.

On considère la suite définie par la relation de récurrence $x_{n=1} = g(x_n)$ où $g(x) = x - \frac{f(x)}{f'x)}$.

4.5.2 Convergence **Proposition** : la suite (x_n) est décroissante et minorée par ℓ.

Démonstration : Soit $n \in \mathbb{N}$. Supposons $\ell < x_n$. On a $x_{n+1} - \ell = x_n - \ell - \frac{f(x_n)}{f'(x_n)} = x_n - \ell - \frac{f(x_n)-f(\ell)}{f'(x_n)} = (x_n - \ell)(1 - \frac{f'(c_n)}{f'(x_n)})$ où $\ell < c_n < x_n$ par le théorème des accroissements finis. Mais f' croît strictement, donc, $0 < f'(c_n) < f'(x_n)$, donc $0 < \frac{f'(c_n)}{f'(x_n)} < 1$ d'où $0 < x_{n+1} - l < x_n - \ell$. Ainsi, $x_{n+1} > \ell$ et $x_{n+1} < x_n$.

Proposition : $x_n \to \ell$ lorsque n tend vers l'infini.

Démonstration : (x_n) est décroissante et minorée, donc converge vers $\ell' \in \mathbb{R}$. Comme les x_n appartiennent à $[\ell, x_0]$, la limite ℓ' de la suite aussi et cette limite vérifie $g(\ell') = \ell'$, c'est à dire $f(\ell') = 0$. Mais seul ℓ vérifie cela sur $[\ell, x_0]$. Donc $\ell' = \ell$.

4.5.3 Vitesse de convergence Appliquons maintenant le théorème des accroissements finis à g entre ℓ et x_{n+1}. Il existe c_n (pas le même que ci-dessus)), $\ell < c_n < x_n$ tel que $x_{n+1} - \ell = g(x_n) - g(\ell) = (x_n - \ell)g'(c_n)$. Que vaut g' ? Pour tout $x \in I$, $g'(x) = \frac{f(x)f''(x)}{f'(x)^2}$ (immédiat). Ainsi, $|g'(c_n)| = \frac{f(c_n)f''(c_n)}{f'(c_n)^2} = \frac{(f(c_n)-f(\ell))f''(c_n)}{f'(c_n)^2} = \frac{(c_n-\ell)f'(d_n)f''(c_n)}{f'(c_n)^2}$ où $\ell < d_n < c_n$, par le théorème des accroissements finis. Majorons sans vergogne : $|\frac{f'(d_n)f''(c_n)}{f'(c_n)^2}| \leq \frac{f'(x_0)M}{f'(\ell)^2}$, où où $M = \max_{[\ell,x_0]} |f''|$.

Posons $K = \frac{f'(x_0)M}{f'(\ell)^2}$. On a donc $|x_{n+1} - \ell| \leq K|x_n - \ell||c_n - \ell| \leq K|x_n - \ell|^2$. Et par une récurrence sans difficulté, $|x_n - \ell| \leq K^{2^n-1}|x_0 - \ell|^{2^n}$, ou encore

$$|x_n - \ell| \leq \frac{1}{K}(K|x_0 - \ell|)^{2^n}$$

4.5.4 Exemple Cette majoration de l'erreur commise n'a d'intérêt, de façon générale, que si x_0 est assez proche de ℓ. On peut faire mieux mais on ne le fera pas ici. Pour notre exemple fétiche, avec $x_0 = 2$, on a $K = 1$. Coup de chance. Ainsi,

$$|x_n - \sqrt{2}| \leq |2 - \sqrt{2}|^{2^n}$$

Si l'on veut $\sqrt{2}$ avec 2000 chiffres exacts il suffira donc de prendre n tel que $(2 - \sqrt{2})^{2^n} \leq 10^{-2000}$, c'est à dire $n \geq \lg \frac{-2000}{\log(2-\sqrt{2})}$.

```
In [30]: a = log(2 - sqrt(2)) / log(10)
         N(log(-2000 / a) / log(2))
```

```
Out[30]: 13.0719673956134
```

La théorie prouve que 14 itérations sont suffisantes.