Suites récurrentes et visualisation

Calcul formel – TP 4

1. Fractions continues

1. Le calcul de la fraction continue s'implémente facilement à l'aide d'une boucle :

```
def frac_finie_vers_rationnel(myfrac):
     invmyfrac = list(reversed(myfrac))   # Liste inverse
     reel = invmyfrac[0]                  # a0
     for a in invmyfrac[1:]:              # Iteration
         reel = a + 1/reel
     return reel
```

2. On calcule les valeurs de la suite doublement récurrente à l'aide d'une fonction qui d'abord initialise les valeurs de p et q, puis, pour chaque élément a_i $(0 \leqslant i \leqslant n)$, calcule les nouvelles valeurs de p et q, et enfin les ré-initialise en vue de l'itération suivante :

```
def frac_vers_rationnel(myfrac,n):
    p_k_2 = 0
    q_k_2 = 1
    p_k_1 = 1
    q_k_1 = 0
    for a in myfrac[0:n]:
        p_k = a*p_k_1 + p_k_2
        q_k = a*q_k_1 + q_k_2
        p_k_2, q_k_2 =  p_k_1, q_k_1
        p_k_1, q_k_1 =  p_k, q_k
    return p_k/q_k
```

   Noter l'utilisation de la double affectation du type : `x,y = newx,newy`.

   Le réel qui a pour développement en fraction continue $[1,1,1,\ldots]$ est le nombre d'or.

3. L'algorithme du calcul successif des nombres $a_0,\ldots,a_n$ s'écrit en suivant les indications, la seule précaution est de veiller à bien arrêter la boucle si nécessaire pour éviter de diviser par zéro :

```
def reel_vers_frac(x,n):
    y = x
    myfrac = []
    for k in range(n):
        a = floor(y)        # Partie entiere
```

```
        myfrac.append(a)
        if y <> a:
            y = 1/(y-a)      # Formule de recurrence
        else:
            break            # Stop car y entier
    return myfrac
```

La méthode append permet d'ajouter un élément en fin de liste.

4. **Énigme.** On applique notre algorithme avec $x = \sqrt{\pi}$ ou à une approximation (les résultats sont obtenus plus rapidement mais il convient d'être prudent sur la validité du résultat obtenu) : `x = sqrt(pi).n(digits=200)`. On construit le développement en fractions continues, par exemple `myfrac = reel_vers_frac(x,100)`. Il suffit alors de parcourir cette liste pour rechercher l'élément demandé :

```
i=0
while (myfrac[i]!=59):
    i=i+1
print(i)
```

Mais le résultat change (lorsque l'élément est rencontré) suivant le niveau d'approximation du départ et le nombre d'éléments du développement demandé. Il faut, en modifiant ces deux paramètres, s'assurer obtenir le résultat demandé.

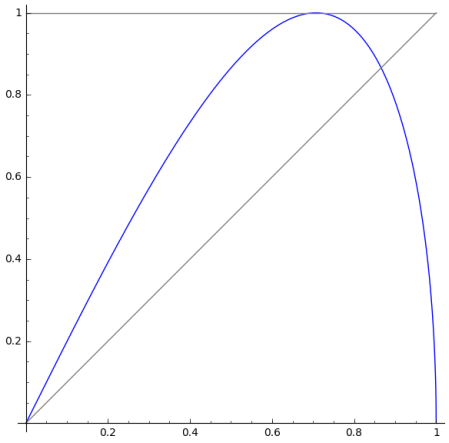
Réponse attendue : 1286.

2. Une suite récurrente

1. (a) `f(x) = 2*x*sqrt(1-x^2)`

   (b) On utilise les commandes `plot` et `show` :

```
G = plot(f,(x,0,1))                        # La fonction
G = G + plot(x,(x,0,1),color="gray")       # La droite (y=x)
G = G + plot(1,(x,0,1),color="gray")       # La droite (y=1)
G.show(aspect_ratio=1)
```

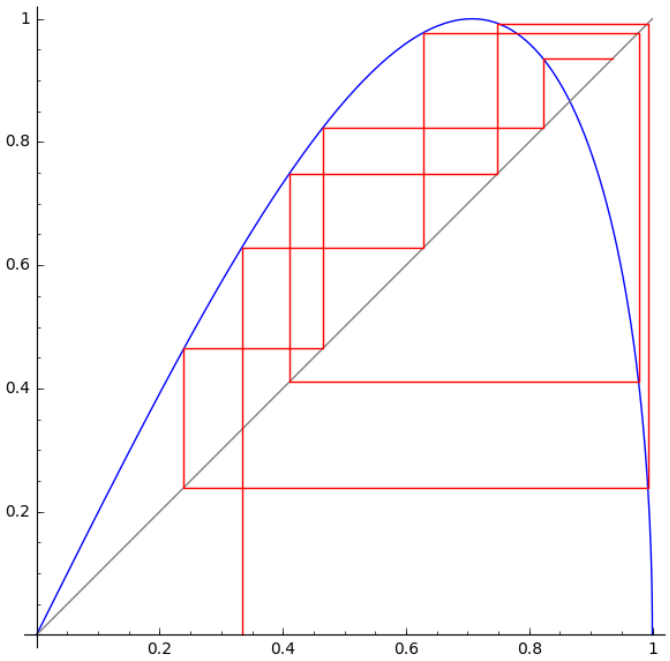


   (c) Une étude de fonction montre que $f([0,1]) \subset [0,1]$. En effet on calcule

$$f'(x) = 2\frac{1-2x^2}{\sqrt{1-x^2}}.$$

La fonction f est donc croissante sur $[0, \sqrt{2}/2]$ et décroissante sur $[\sqrt{2}/2, 1]$. On en déduit que son maximum est $f(\sqrt{2}/2) = 1$.

2. Comme dans le cours :



3. À l'aide de la commande `solve(f==x,x)`, on trouve deux points fixes : 0 et $\sqrt{3}/2$.
4. On suit la démarche de la question précédente, pour $f \circ f$:

```
g = f(f(x))
solve(g==x,x)
```

Toutefois, pour obtenir des solutions explicites, on aura intérêt à élever l'équation au carré par `solve(g^2==x^2,x)`. Dans tous les cas, on trouve 2 points fixes de g qui ne sont pas des points fixes de f. f échange ces deux points fixes de g. Il y a donc un seul cycle d'ordre 2 pour f.

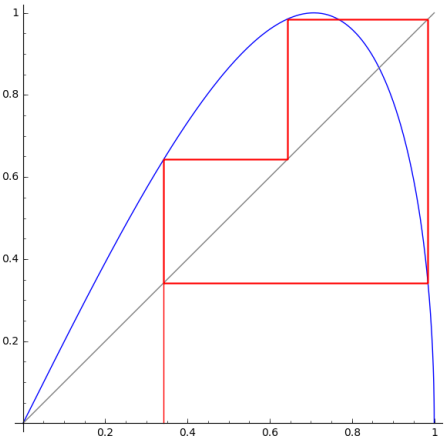
5. **Énigme.** C'est très similaire :

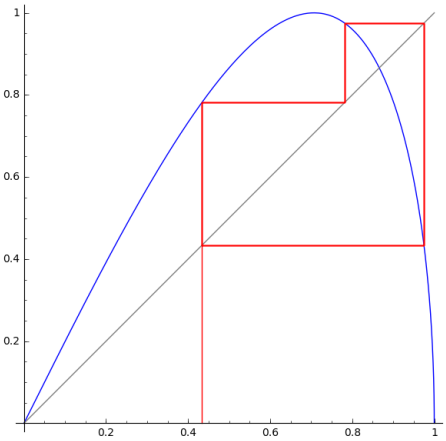
```
h = f(f(f(x)))
solutions = solve(h^2 == x^2,x)
for sol in solutions:
    print sol.rhs().n()
```

Comme les solutions sont renvoyées sous une forme complexe, on calcule leur valeur numérique approchée pour confirmer que les solutions sont en fait réelles. Parmi les 8 solutions, on trouve cette fois-ci 6 points fixes de $f \circ f \circ f$ qui ne sont pas points fixes de f (donc deux cycles d'ordre 3 pour f).

Réponse attendue : 4160.

Voici les deux cycles d'ordre 3 de f.





3. Le *page rank* de Google

Énigme. On commence par définir une liste décrivant les liens vers chacune des pages :

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

On initialise tous les *page rank* à 1 :

```
def init_page_rang(liens):
    pr = [ 1 for page in liens ]
    return pr
```

On définit une fonction calculant pr_{n+1} en fonction de pr_n :

```
def iter_page_rank(liens,pr):
    N = [ len(page[1]) for page in liens ]
    prnew = [0 for pages in liens]
    for page in liens:
        i, sousliens = page
        for k in sousliens:
            prnew[k] = prnew[k] + pr[i]/N[i]
    return prnew
```

Noter qu'on commence par définir une liste `N` contenant le nombre de liens de chaque page. La façon de sommer est un peu déconcertante a priori (mais correcte) puisqu'on regarde la contribution d'une page donnée aux *page rank* de toutes les autres, alors que dans la formule donnée dans l'énoncé, on regarde les contributions de toutes les pages au *page rank* d'une page donnée.
On fait une nouvelle boucle pour calculer pr_n à partir du *page rank* initial :

```
def page_rank(liens,n):
    pr = init_page_rang(liens)
    for i in range(n):
        pr = iter_page_rank(liens,pr)
    return pr
```

Enfin, on affiche les 5 premiers *page rank* et leurs valeurs numériques approchées :

```
for n in range(5):
    pr = page_rank(liens,n)
    print pr
    print 'n␣=␣', n, [p.n() for p in pr]
```

Ceci permet de classer les pages les plus populaires :
Réponse attendue : 301.

4. Attracteur de Hénon

1. On commence par définir H et la liste des points à l'aide d'une fonction contenant une boucle :

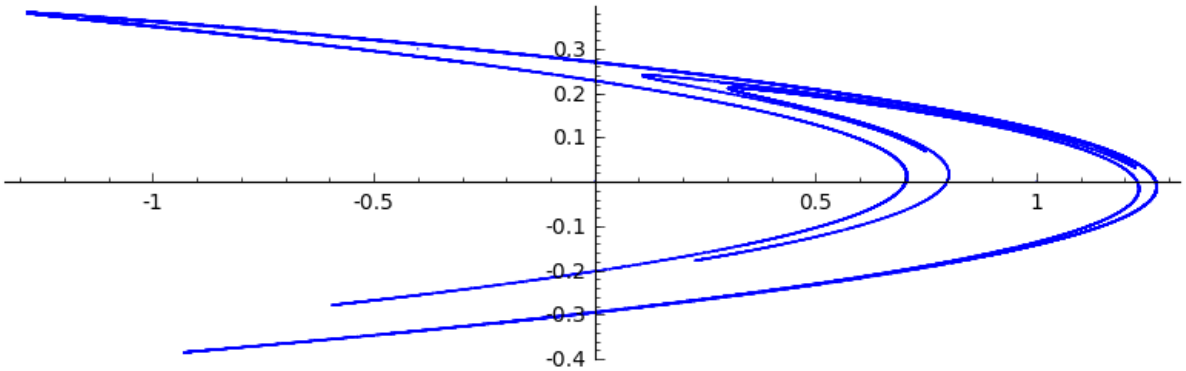
   ```
   def H(x,y):
       return y+1-alpha*x^2, beta*x

   def henon(x0,y0,n,alpha,beta):
       x, y = x0,y0
       mespoints = [(x0,y0)]
       for k in range(n):
           x, y = H(x,y)
           mespoints.append((x,y))
       return mespoints
   ```

   Ensuite, on transforme la liste de points en une "somme" de points grâce à la commande `points`, puis on l'affiche avec la commande `show` :

   ```
   def affiche_henon(x0,y0,n,alpha,beta):
       mespoints = henon(x0,y0,n,alpha,beta)
       G = points(mespoints,pointsize=1)
       G.show(aspect_ratio=1)
   ```

   Il ne reste plus qu'à utiliser cette fonction avec les valeurs données :

   ```
   x0, y0 = 0, 0
   n = 1000
   alpha = 1.4
   beta = 0.3
   affiche_henon(x0,y0,n,alpha,beta)
   ```



2. **Énigme.** On adapte la fonction de la question précédente, mais cette fois on n'ajoute le point calculé à la liste que si ses coordonnées sont dans la fenêtre donnée :

```
def zoom_henon(x0,y0,n,alpha,beta,x_min,x_max,y_min,y_max):
    x, y = x0,y0
    mespoints = []
    for k in range(n):
        x, y = H(x,y)
        if (x_min <= x <= x_max) and (y_min <= y <= y_max):
            mespoints.append((x,y))
    return mespoints
```

Le reste est en tout point identique à la question précédente. On obtient le nombre de points grâce à la commande `len` qui permet de donner la longueur de la nouvelle liste.

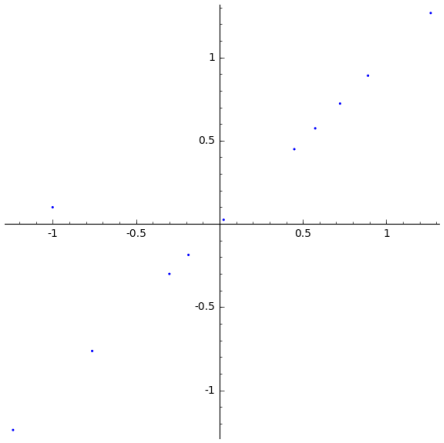
Réponse attendue : 100.

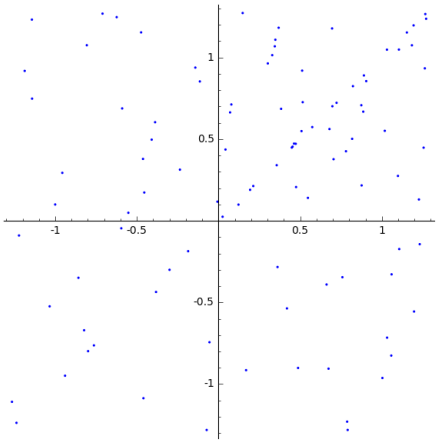
Remarque. Les calculs sont très sensibles à la précision demandée dans les calculse, mais aussi à l'écriture de H. En effet, si on définit H par la formule `y+1-alpha*x*x, beta*x` au lieu de `y+1-alpha*x^2, beta*x` alors les résultats sont différents.

3. Il suffit une nouvelle fois d'adapter la fonction donnant la liste des points :

```
def sensibilite_henon(x0,y0,xx0,yy0,n,alpha,beta):
    x, y = x0,y0
    xx, yy = xx0, yy0
    mespoints = [(x0,yy0)]
    for k in range(n):
        x, y = H(x,y)
        xx, yy = H(xx,yy)
        mespoints.append((x,xx))
    return mespoints
```

Pour $n = 10$ (figure de gauche), les points sont bien alignés le long de la diagonale, donc la sensibilité à la condition initiale n'est pas très forte, par contre ce n'est plus du tout le cas pour $n = 100$ (figure de droite).


1
0.5
-1
-0.5
0.5
1
-0.5
-1


1
0.5
-1
-0.5
0.5
1
-0.5
-1