SUITES AVEC CoffeeScript

I/ Suites numériques

Avant de parler de suites numériques, il est bon de se demander ce qu'est exactement une suite. La définition suivante vient de l'*Encyclopédie ou dictionnaire raisonné des sciences, des arts et des métiers* (d'Alembert pour la partie mathématique) de 1751 :

> *se dit d'un ordre ou d'une progression de quantités, qui croissent, ou décroissent suivant quelques lois. Lorsque la suite va toujours en s'approchant de plus en plus de quelque quantité finie (...) on l'appelle une suite convergente, et si on la continue à l'infini, elle devient enfin égale à cette quantité.*

Cette définition est visiblement restreinte aux suites numériques infinies, mais on peut aussi définir une suite dont le terme général est choisi dans un ensemble E, comme une fonction de $\mathbb{N}$ dans E [1]. Ce qui permet alors de parler de suites de points, de suites de droites, de suites de cercles, de suites de fonctions, de suites de suites etc.

Dans cet article, on se restreindra aux suites numériques telles que définies par d'Alembert ci-dessus, puisqu'elles se prêtent à des questions intéressantes sur la convergence. D'ailleurs les premiers exemples de suites étaient numériques et étaient utilisés pour des calculs numériques :

- L'algorithme de Heron (connu des babyloniens[2]) pour calculer des racines carrées consiste essentiellement à produire une suite convergeant vers la racine carrée et calculer ses termes successifs jusqu'à ce qu'on soit suffisamment proche de la limite ;
- Plusieurs calculs de géométrie ont été menés par Archimède de façon analogue : Par exemple, pour calculer π, Archimède construit une suite convergeant vers π et calcule ses termes successifs[3] ;
- En Inde, Aryabatha calculait les premières tables trigonométriques (à moins de 4° près[4]) en considérant les nombres qu'elles contiennent, comme termes successifs de suites récurrentes ;
- Al Tusi résolvait des équations du troisième degré par une méthode similaire[5], que par la suite l'équipe d'Al-Kashi à Samarcande a perfectionnée au point de pouvoir faire une table trigonométrique au degré près, la première au monde de ce genre[6] ;
- Raphson et son contemporain Newton découvrent indépendamment l'un de l'autre une méthode pour résoudre presque n'importe quelle équation, par une suite qui converge vers l'une de ses solutions. Elle généralise d'ailleurs la méthode de Heron ; on va traiter cet exemple plus bas ;
- Briggs propose à Neper un algorithme pour calculer itérativement des logarithmes ; cet

---

1 L'existence d'un tel objet s'appelle « axiome du choix » ; elle est indécidable...

2 Voir sa description ici : http://fr.wikipedia.org/wiki/M%C3%A9thode_de_H%C3%A9ron

3 L'histoire est contée ici : http://www.math.uha.fr/Pi/archimede.html Le site internet tout entier vaut le détour d'ailleurs.

4 Voir à ce sujet l'article de Jean Lefort : http://www.apmep.asso.fr/Aryabhata-et-la-table-des-sinus ainsi que l'article sur wikipedia en anglais : http://en.wikipedia.org/wiki/%C4%80ryabha%E1%B9%ADa%27s_sine_table

5 Décrite ici : http://fr.wikipedia.org/wiki/M%C3%A9thode_de_Ruffini-Horner#Valeur_approch.C3.A9e_d.27une_racine

6 Quel est le rapport entre les équations du troisième degré et les tables trigonométriques ? En fait, si on connaît le sinus et le cosinus de 3θ, on trouve ceux de θ par résolution des équations $4\cos^3\theta - 3\cos\theta = \cos(3\theta)$ et $3\sin\theta - 4\sin^3\theta = \sin(3\theta)$ qui sont bien du troisième degré ; à partir des sinus et cosinus de 9°, on trouve successivement ceux de 3° puis de 1°.

algorithme[7], ancêtre du CORDIC moderne, consiste encore une fois à itérer une suite ;

- Simpson calcule des intégrales en additionnant des termes successifs de suites ;
- De Moivre et Stirling, dans un travail collaboratif, ont trouvé une suite qui approche celle des factorielles ; leur travail implique donc que la factorielle est une suite d'entiers...
- Euler découvre la constante qui porte son nom en comparant la suite $(\ln n)_{n\geqslant 1}$ avec la série harmonique $\sum_{k=1}^{n}\frac{1}{k}$; il résout des équations différentielles par la méthode d'Euler qui est aussi basée sur une itération de suite numérique.

Bref, d'Alembert avait à sa disposition de nombreux exemples d'utilisation des suites pour résoudre approximativement des problèmes de mathématiques, que ce soit pour résoudre des équations, calculer des intégrales ou résoudre des équations différentielles... Cela s'est confirmé avec notamment Gauss qui étudiait le comportement asymptotique de la répartition des nombres premiers[8], Cauchy, Riemann etc.

Mais le premier exemple de programme informatique[9] était un algorithme de calcul des nombres de Bernoulli, qui porte donc sur une suite entière. Le voici en CoffeeScript (pour l'outil alcoffeethmique ci-joint) :

```
B = [1]
for n in [1..10]
   somme = 0
   somme += combinaison(n+1,k)*B[k] for k in [0..n-1]
   B.push -somme/(n+1)

affiche B
```

Et lorsque dans les années 1930, Jacques Herbrand et ses émules[10] se demandaient ce qu'est un calcul, leurs définitions différentes mais équivalentes revenaient toutes à des suites d'entiers[11]. Par exemple, calculer π, c'est calculer ses décimales, qui forment une suite d'entiers. Par conséquent, le lien entre suites et algorithmes est très fort, et on peut aller jusqu'à définir une suite ainsi :

> Une suite est ce que produit un algorithme basé sur une boucle ; le nombre n de passages dans la boucle est l'indice, et l'objet produit au n -ième passage dans la boucle est le terme d'indice n de la suite.

Ensuite, on précise que si l'objet est un nombre, on parle de suite numérique ; que si de plus il est entier, on parle de suite entière, que si il est de plus en plus grand à chaque passage dans la boucle, la suite est croissante, que s'il se rapproche de plus en plus d'un nombre donné[12] la suite est convergente etc.

---

7 Dans alcoffeethmique, cliquer sur l'exemple « Briggs » pour tester l'algorithme.
8 Voir plus bas aussi pour deux suites adjacentes dues à Gauss et qui permettent de calculer π
9 Rédigé par Ada Lovelace au milieu du XIXe siècle pour la machine analytique de Babbage
10 Gödel (son correspondant), Church, Post, Turing, Kleene etc
11 En fait, plus généralement, des suites de suites finies d'entiers ; mais le plus souvent, ce sont des suites d'entiers
12 Très précisément, on dira que la suite converge vers l si, pour tout $p\in\mathbb{N}$, l'algorithme obtenu à partir du précédent en remplaçant sa boucle par `jusqu'à ce que` $|u_n-l|<10^{-p}$ s'arrête un jour. Cette définition est correcte pour les suites monotones

Aussi, dans cet article, on va voir ce que le langage CoffeeScript[13] apporte à l'étude des suites du point de vue algorithmique. Dans le reste de cette première partie, on va se concentrer sur les suites réelles, puis dans la seconde partie, on évoquera les suites entières (ainsi que les suites sur un corps fini comme $\mathbb{Z}/p\mathbb{Z}$). La troisième partie sera consacrée aux séries et enfin la quatrième partie va vers des élargissements.

Pour tester les algorithmes décrits en CoffeeScript, on peut aller sur le site www.coffeescript.org et là, cliquer sur « try CoffeeScript » puis entrer les algorithmes et cliquer sur « run » pour les lancer. Mais dans ce cas, il faut remplacer tous les « affiche » par des « alert ». On peut donc préférer l'outil joint à cet article, en lançant le fichier alcoffeethmique.html puis en programmant les exemples dans sa fenêtre de programmation[14]. C'est d'ailleurs ainsi que ce document a été rédigé.

Un exemple pour se mettre en bouche :

- La boule de rayon 1, en dimension 1, est le segment [-1;1] ; son contenu monodimensionnel est sa longueur qui vaut 2.
- La boule de rayon 1 dans le plan, est le disque de rayon 1 ; son contenu bidimensionnel est l'aire du disque, qui vaut $\pi \times 1^2 \simeq 3{,}14$
- La boule de rayon 1 dans l'espace de dimension 3 est la boule au sens où on l'entend habituellement (points situés à moins d'une unité de distance, du centre de la boule). Son contenu tridimensionnel est son volume, qui vaut $\frac{4}{3}\pi \times 1^3 \simeq 4{,}19$
- On peut définir, à défaut de visualiser, une boule en dimension 4, 5 etc. et son contenu n-dimensionnel. Cela définit une suite qui, au vu de ses premiers termes, est croissante. Se pose alors la question de sa limite (finie ou infinie) mais également de ses variations : Est-elle vraiment croissante ?

Il existe une formule[15] pour calculer le contenu n-dimensionnel d'une boule euclidienne. En CoffeeScript on peut la traduire ainsi :

```
volumes = [0]
p = 1
for n in [1..16]
   if n%2 is 0
      volumes.push puissance(pi,n/2)/factorielle(n/2)
   else
      p *= n
      volumes.push puissance(2,(n+1)/2)*puissance(pi,(n-1)/2)/p

affiche "Le #{n}-volume de la boule de dimension #{n} est
#{volumes[n]}" for n in [1..8]
```

On verra plus bas qu'il est commode de stocker les termes successifs d'une suite dans un tableau appelé ici **volumes**. La variable **p** contient le produit des entiers impairs successifs (initialement 1). On boucle sur **n** allant de 1 à 16 ; selon la parité de **n**, on pousse (« push ») dans le tableau, ou bien

13 Choisi pour sa concision et le fait qu'il tourne en ligne sur navigateur internet

14 On peut aussi cliquer sur un exemple en bas de page pour le voir remplacer le contenu de la fenêtre de programmation, et il se trouve que beaucoup d'exemples concernent les suites, à commencer par le premier d'entre eux qui est la suite de Collatz

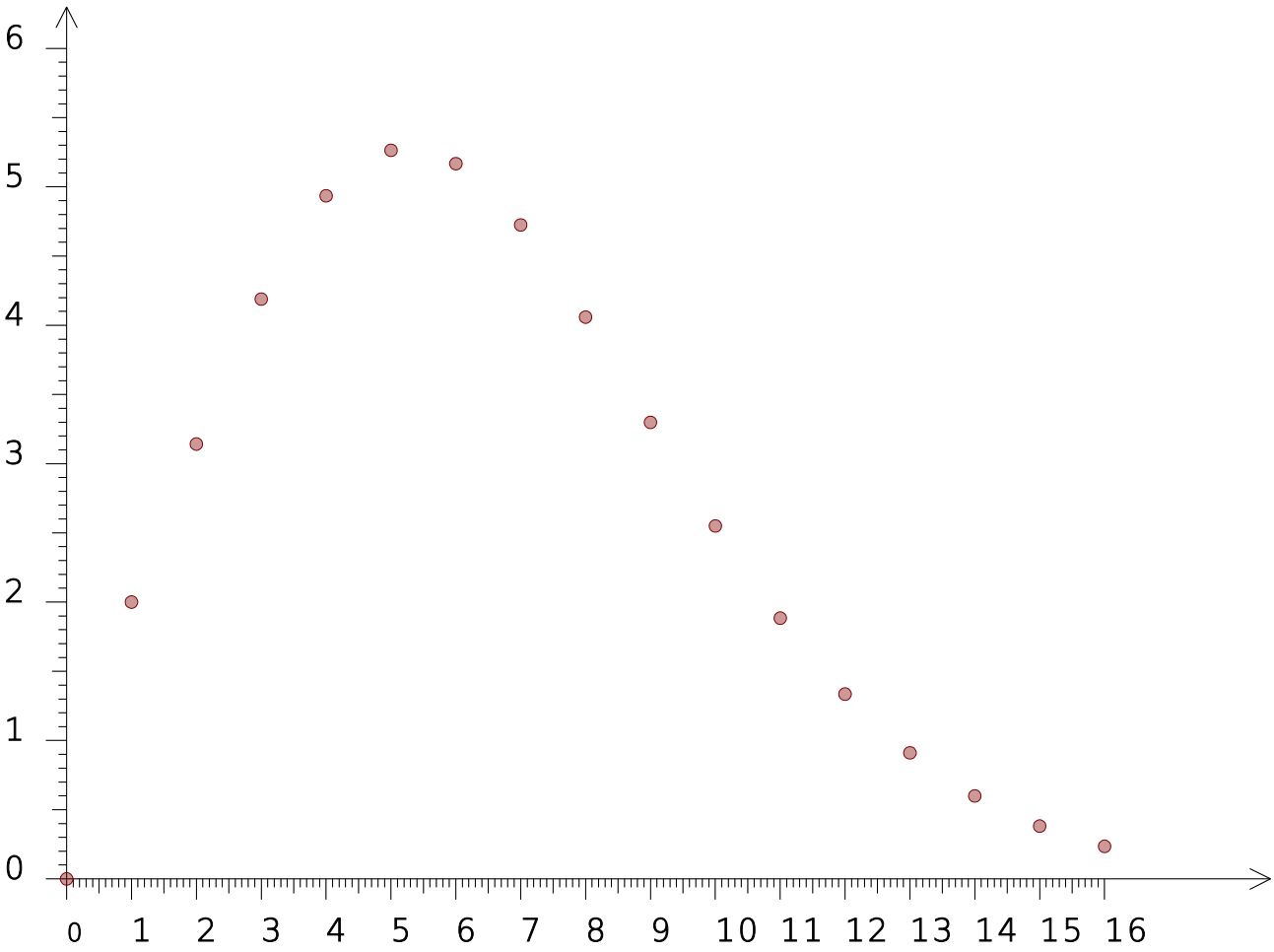
15 Trouvée ici : http://fr.wikipedia.org/wiki/Calcul_du_volume_de_l%27hypersph%C3%A8re

$\frac{\pi^{\frac{n}{2}}}{\frac{n}{2}!}$ si **n** est pair, ou bien $\frac{\left(\sqrt{2}\right)^{\frac{n+1}{2}} \times \pi^{\frac{n-1}{2}}}{p}$ si **n** est impair, et on met à jour la variable **p**.

Après l'avoir exécuté, l'affichage d'***alcoffeethmique*** devient quelque chose comme ceci :

```
Algorithme lancé
Le 1-volume de la boule de dimension 1 est 2
Le 2-volume de la boule de dimension 2 est 3.141592653589793
Le 3-volume de la boule de dimension 3 est 4.1887902047863905
Le 4-volume de la boule de dimension 4 est 4.934802200544679
Le 5-volume de la boule de dimension 5 est 5.263789013914324
Le 6-volume de la boule de dimension 6 est 5.167712780049969
Le 7-volume de la boule de dimension 7 est 4.724765970331401
Le 8-volume de la boule de dimension 8 est 4.058712126416768

Algorithme exécuté en 105 millisecondes
```

Surprise :



La dimension 5 se distingue : Au-delà, les contenus n-dimensionnels ne font que décroître et la suite des contenus n-dimensionnels, loin de tendre vers l'infini, tend au contraire vers 0[16]. Pour faire la

16 Le nombre de polytopes réguliers convexes, lui aussi, est une suite (entière) un peu surprenante : Il y a une infinité de polygones réguliers en dimension 2, il y a 5 solides de Platon en dimension 3, il y a 6 polytopes réguliers en

figure ci-dessus, le script suivant a été utilisé :

```
volumes = [0]
p = 1
for n in [1..16]
    if n%2 is 0
        volumes.push puissance(pi,n/2)/factorielle(n/2)
    else
        p *= n
        volumes.push puissance(2,(n+1)/2)*puissance(pi,(n-1)/2)/p

dessineSuite volumes, 16, 0, 6, 3, 'maroon'
```

1) Suites récurrentes et suites définies explicitement

Commençons par un exemple de suite définie explicitement pour $n \geqslant 1$: $u_n = \frac{1}{n(n+1)}$. On dit que la suite est définie explicitement si la formule qui permet de calculer son terme général ne fait intervenir que l'indice n et non sa valeur précédente u ; dans le cas contraire, on dit que la suite est récurrente. Il peut y avoir des suites qui ne sont ni l'un ni l'autre, et il est parfois possible de définir explicitement certaines suites récurrentes. On en verra très bientôt l'intérêt[17].

On peut transformer cette suite en une « boucle à nombre prédéfini d'exécutions » :

```
u = (n) -> 1/(n*(n+1))
for indice in [1..10]
    termeGénéral = u(indice)
    affiche "Le terme de rang #{indice} est #{termeGénéral}"
```

Après exécution de cette boucle pour l'indice allant de 1 à 10, on a ceci dans la fenêtre d'affichage :

```
Algorithme lancé
Le terme de rang 1 est 0.5
Le terme de rang 2 est 0.16666666666666666
Le terme de rang 3 est 0.08333333333333333
Le terme de rang 4 est 0.05
Le terme de rang 5 est 0.03333333333333333
Le terme de rang 6 est 0.023809523809523808
Le terme de rang 7 est 0.017857142857142856
Le terme de rang 8 est 0.013888888888888888
Le terme de rang 9 est 0.011111111111111112
Le terme de rang 10 est 0.00909090909090909

Algorithme exécuté en 31 millisecondes
```

dimension 4, et dans toutes les dimensions supérieures il y a 3 polytopes réguliers seulement.

17 Par exemple, la suite $u_{n+1} = \arctan(n) u_n (1 - u_n)$ qui ressemble à la suite logistique mais avec un paramètre variant selon n ; cette suite tend vers une limite proche de 0,363.

L'avantage de la définition explicite d'une suite, du moins lorsqu'il n'y a pas de forme indéterminée, est qu'on peut avoir la limite avec cette petite variante du script :

```
u = (n) -> 1/(n*(n+1))
for indice in [10, 100, 1000, 1000000, Infinity]
    termeGénéral = u(indice)
    affiche "Le terme de rang #{indice} est #{termeGénéral}"
```

Le résultat obtenu est éloquent :

```
Algorithme lancé
Le terme de rang 10 est 0.00909090909090909
Le terme de rang 100 est 0.00099009900990099902
Le terme de rang 1000 est 9.99000999000999e-7
Le terme de rang 1000000 est 9.99999000001e-13
Le terme de rang Infinity est 0

Algorithme exécuté en 70 millisecondes
```

De façon similaire, si on définit récursivement une suite arithmétique par $u_{n+1}=u_n+r$, on ne peut pas calculer la limite de cette suite avec CoffeeScript, mais si on la définit de façon explicite avec $u_n=u_0+n\times r$, CoffeeScript permet de calculer la limite comme terme d'indice infini u_∞ comme ci-dessus. Et la réponse est celle du cours, que la raison soit positive, nulle ou négative ! De même, une suite géométrique, dès lors qu'elle est définie explicitement, affiche correctement sa limite comme u_∞ quelle que soit la valeur de sa raison, alors que la définition récurrente ne le permet pas. Enfin, la somme des termes d'une suite géométrique peut se calculer algorithmiquement par une boucle, mais la limite de cette somme ne s'obtient par la méthode ci-dessus, que si on peut la calculer explicitement.

Ceci dit, une suite étant la donnée d'une quantité dénombrable de réels, même si ces réels sont des petits entiers qui occupent peu de place dans la mémoire de l'ordinateur, celle-ci est finie et on ne peut donc pas stocker dans l'ordinateur toute la suite. Alors on ne boucle qu'un nombre fini de fois comme on l'a vu sur les exemples ci-dessus, et on stocke les termes calculés dans un tableau de nombres, ce qui simplifie le traitement numérique ou graphique sur le résultat. Par exemple :

```
u = (n) -> 1/(n*(n+1))
tableau = []
for indice in [1..10]
    termeGénéral = u(indice)
    tableau.push termeGénéral

affiche tableau
```

Chaque fois qu'on a calculé un terme de la suite dans la boucle, on le pousse (« push ») sur le tableau, qui contient donc à la fin la liste des termes calculés. Alors ***alcoffeethmique*** permet de calculer la somme des termes avec laSommeDe tableau, la moyenne des termes du tableau avec laMoyenneDe tableau, son écart-type avec lEcartTypeDe tableau, sa médiane avec laMédianeDe tableau, ou représenter graphiquement les premiers termes de la suite avec dessineSuite tableau, 9, 0, 0.5, 3, 'green' (mais dans le cas présent, il y a un décalage lié au fait que la suite n'est définie qu'à partir de 1 alors que traditionnellement, les tableaux sont indexés à partir de 0).

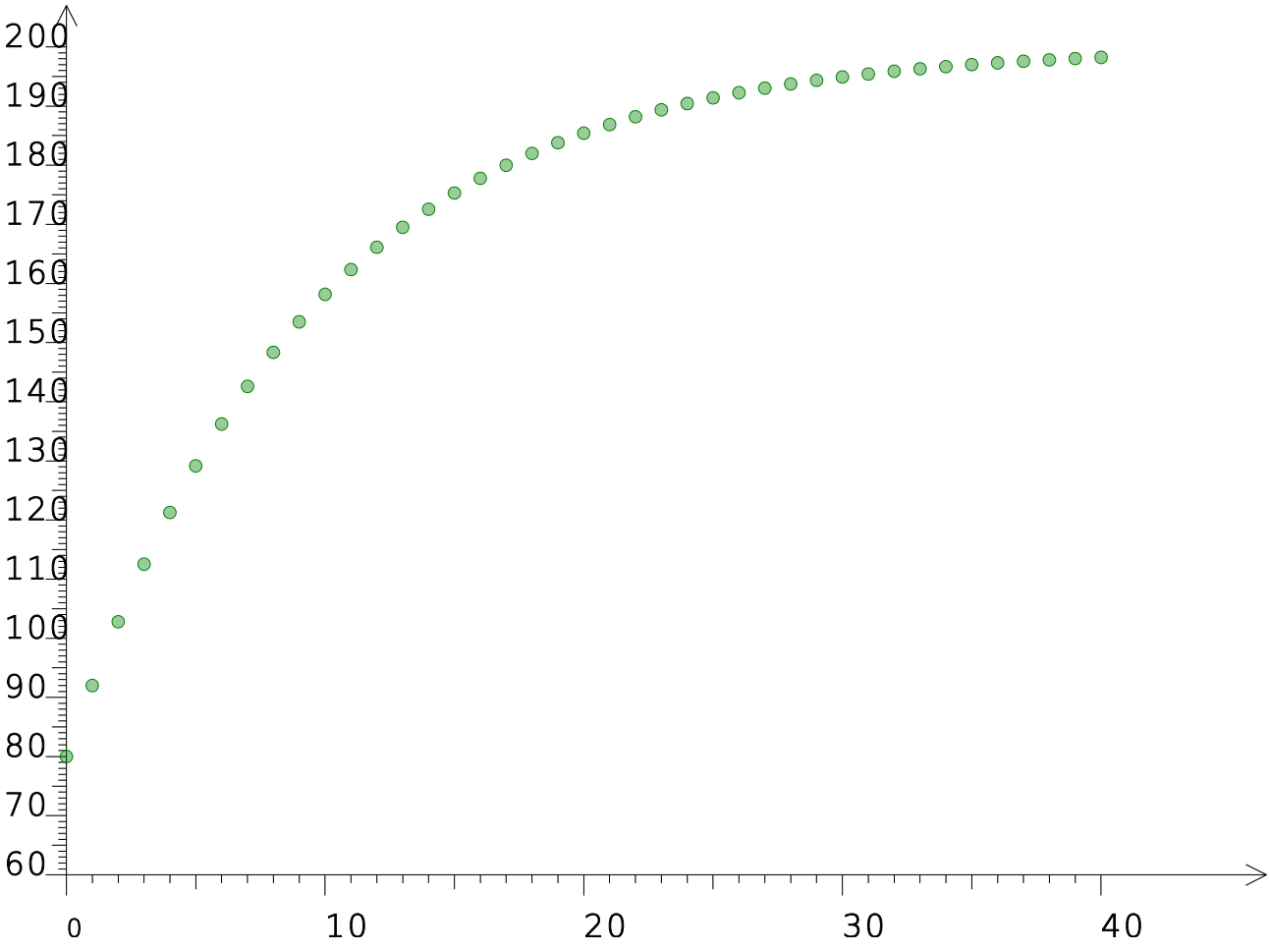
Par exemple, voici un extrait du sujet de Bac ES Antilles-Guyane septembre 2013 :

> En 2005, année de sa création, un club de randonnée pédestre comportait 80 adhérents. Chacune des années suivantes
> on a constaté que :
> — 10 % des participants ne renouvelaient pas leur adhésion au club ;
> — 20 nouvelles personnes s'inscrivaient au club.
> On suppose que cette évolution reste la même au fil des ans.

La suite donnant le nombre d'adhérents est donc arithmético-géométrique. Elle est facilement décrite par $a_{n+1}=0{,}9\,a_n+20$ (avec $a_0=80$, l'indice 0 correspondant donc à l'année 2005). En CoffeeScript :

```
a = 80
tableau = [a]
for n in [1..100]
    a = 0.9*a + 20
    tableau.push a

dessineSuite tableau, 40, 60, 200, 3, 'green'
```

L'algorithme du sujet incite à mettre a=troncature(0.9*a+20) comme affectation, mais cela fait perdre son caractère arithmético-géométrique à la suite. En plus, la limite de la suite devient alors 191 au lieu de 200. Voici la représentation graphique obtenue avec le script ci-dessus :



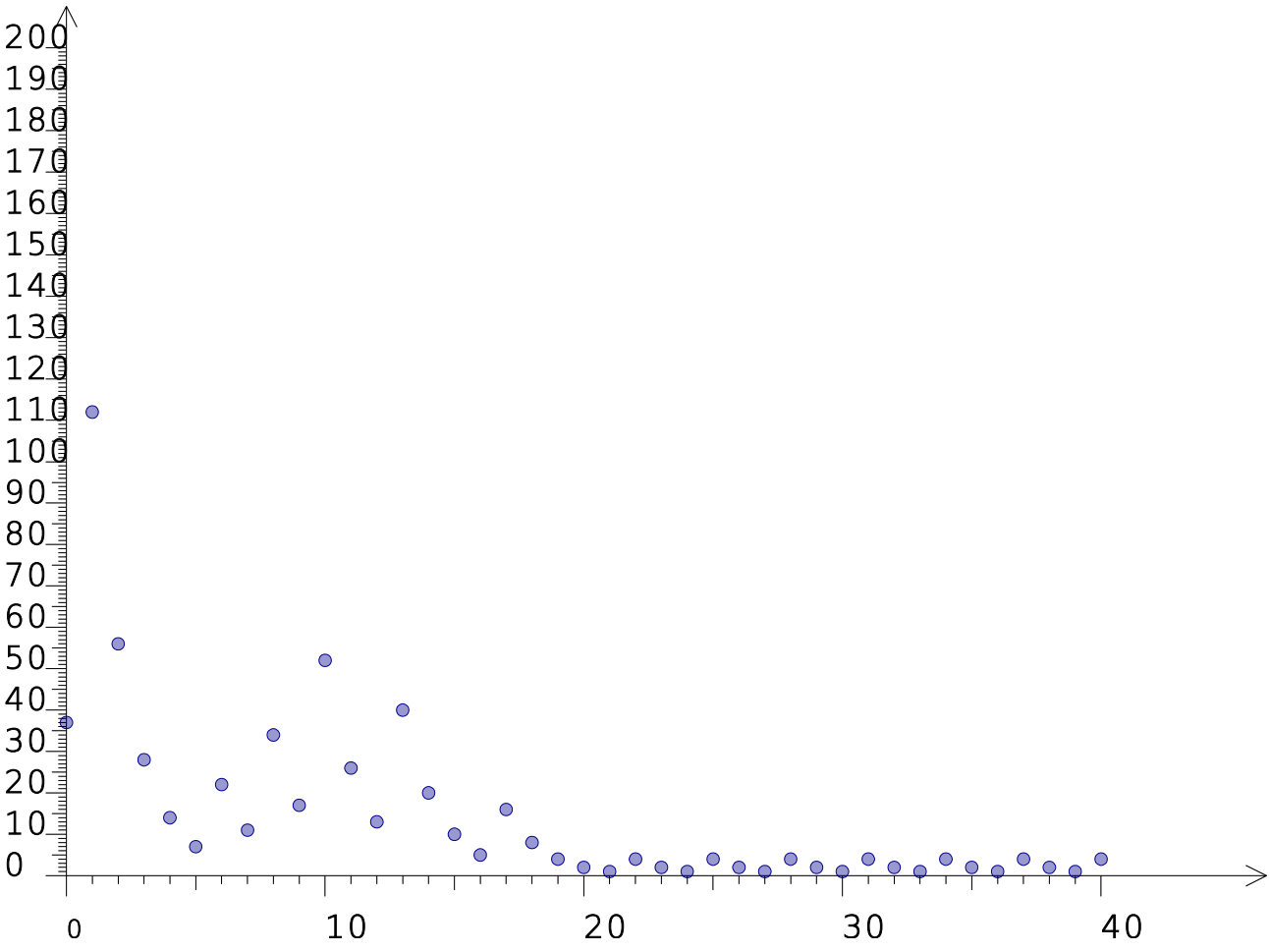
La suite de l'exercice consiste essentiellement à prouver la formule explicite suivante pour la suite $a_n = 200 - 120 \times 0{,}9^n$. Elle permet à CoffeeScript de donner la limite de a_n :

```
a = (n) -> 200-120*puissance(0.9,n)

affiche a(Infinity)
```

La limite est bien 200 :

```
Algorithme lancé
200

Algorithme exécuté en 157 millisecondes
```

Un autre exemple de suite récurrente connu est la suite de Collatz $u_{n+1} = \frac{u_n}{2}\cos^2\left(\frac{\pi u_n}{2}\right) + (3u_n + 1)\sin^2\left(\frac{\pi u_n}{2}\right)$ qui, lorsque $u_0 \in \mathbb{N}$, est une suite entière, qui apparemment, finit toujours par cycler sur 4, 2, 1 :



Cette suite ne converge donc pas[18]. Pour dessiner la représentation graphique ci-dessus, le script

18 D'ailleurs on ne lui connaît pas de formule explicite.

utilisé a été celui-ci :

```
f = (x) ->
    carré(cos(pi*x/2))*x/2+carré(sin(pi*x/2))*(3*x+1)

u = 37
t=[u]
for n in [1..40]
    u = f(u)
    t.push u

dessineSuite t, 40, 0, 200, 3, 'darkblue'
```

2) Recherche de seuils

Voici un extrait du sujet de bac ES Nouvelle-Calédonie novembre 2013 (exercice obligatoire-L : Encore une suite arithmético-géométrique) :

> Le premier janvier 2014, Monica ouvre un livret d'épargne sur lequel elle dépose 6 000 euros.
> Elle décide de verser 900 euros sur ce livret chaque premier janvier à partir de 2015 jusqu'à atteindre le plafond autorisé de 19 125 euros.
> On suppose dans tout cet exercice que le taux de rémunération du livret reste fixé à 2,25 % par an et que les intérêts sont versés sur le livret le premier janvier de chaque année.

On s'en doute, le but de l'exercice est de chercher à partir de quelle année Monica aura atteint ou dépassé le plafond. Or dans la description de l'algorithme plus bas le sujet utilise non pas le mot « jusqu'à » mais le mot « tant que » :

```
Variables : MONTANT est un réel
            ANNÉE est un entier
Initialisation : Affecter à MONTANT la valeur 6 000
                 Affecter à ANNÉE la valeur 2014
Traitement : Tant que MONTANT < 19125
             Affecter à MONTANT la valeur 1,0225 × +900
             Affecter à ANNÉE la valeur ANNÉE +1
Sortie : Afficher « Le plafond du livret sera atteint en . . . »
             Afficher ANNÉE
```

Pourquoi ce soudain revirement sémantique ? En effet, l'expression « jusqu'à » ne nécessitant pas de négation, tend à être plus facile à interpréter par les élèves[19], et visiblement plus facile à verbaliser par les concepteurs de sujets de Bac...

19 Le langage Pascal possédait une boucle « until » mais les langages de programmation les plus répandus actuellement n'en ont plus (c, Java, Python, JavaScript notamment). Il en est de même pour les calculatrices graphiques qui n'ont que le « while ». C'est dans un souci de simplification que Jeremy Askhenas a réintroduit la boucle « until ». Merci Monsieur Askhenas !

Voici le corrigé du sujet de Bac précédent en CoffeeScript :

```
montant = 6000
annee = 2014
until montant > 19125
   montant = montant * 1.0225 + 900
   annee += 1

affiche "Le montant est #{arrOdg(montant,2)} euros en #{annee}"
```

On y apprend que

```
Algorithme lancé
Le montant est 20078.3 euros en 2026

Algorithme exécuté en 112 millisecondes
```

3) suites adjacentes

L'une des premières utilisations des suites était le calcul de π par Archimède[20]. Pour ce faire, Archimède encadrait π entre les périmètres de deux polygones de plus en plus proches d'un cercle. Ce qui revient en fait à créer non pas une suite récurrente, mais deux, et qui sont adjacentes. L'avantage dans ce cas est qu'il est facile de savoir quand arrêter le calcul : Dès que l'écart entre les deux suites est suffisamment petit.

Or pour les suites adjacentes, CoffeeScript présente un intérêt hérité de Python : La possibilité d'affecter simultanément les deux variables a et b : En effet, si on a $a_{n+1}=f\left(a_n,b_n\right)$ et $b_{n+1}=g\left(a_n,b_n\right)$ on a en général besoin d'une variable temporaire pour stocker le résultat du calcul de $a_{n+1}=f\left(a_n,b_n\right)$ puisqu'à ce stade on a encore besoin de l'ancienne valeur de a_n et on ne peut donc pas « écraser » la variable a avant d'avoir recalculé b . Il faut avouer que l'affectation simultanée est plus facile à rédiger en Python :

```
a, b = f(a,b), g(a,b)
```

En CoffeeScript, c'est à peine plus compliqué[21] :

```
[a, b] = [f(a,b), g(a,b)]
```

20 En fait l'algorithme de Heron se prête aussi à un calcul par deux suites, mais elles ne sont pas adjacentes car pas monotones.

21 Cette affectation simultanée a convaincu Brendan Eich (inventeur de JavaScript) au point qu'il a fini par l'introduire dans JavaScript. Il est donc déjà possible de programmer avec des affectations simultanées en JavaScript

Cela s'applique aussi à certaines suites non adjacentes comme cet exemple issu d'un autre sujet de bac : Bac S Nouvelle-Calédonie novembre 2013 :

> Soient deux suites (u_n) et (v_n) définies par $u_0=2$ et $v_0=10$ et pour tout entier naturel n ,
>
> $$u_{n+1}=\frac{2u_n+v_n}{3} \quad \text{et} \quad v_{n+1}=\frac{u_n+3v_n}{4}$$

L'objet de l'exercice est de déterminer la limite commune de ces deux suites. Mais l'algorithme fourni avec l'énoncé est très long :

```
Variables :   N est un entier
              U ,V,W sont des réels
              K est un entier
Début :       Affecter 0 à K
              Affecter 2 à U
              Affecter 10 à V
              Saisir N
              Tant que K < N
                   Affecter K + 1 à K
                   Affecter U à W
                   Affecter (2U+V)/3 à U
                   Affecter (W+3V)/4 à W
              Fin tant que
              Afficher U
              Afficher V
Fin
```

Pour comparaison, la version CoffeeScript :

```
[u,v] = [2,10]
tableau = [v]
for n in [1..20]
   [u,v] = [(2*u+v)/3,(u+3*v)/4]
   affiche "#{u} et #{v}"
```

L'affichage permet de voir la convergence commune des deux suites :

```
Algorithme lancé
4.666666666666667 et 8
5.777777777777779 et 7.166666666666667
6.240740740740741 et 6.819444444444445
6.433641975308642 et 6.674768518518519
6.514017489711935 et 6.61448688271605
6.547507287379973 et 6.589369534465021
6.561461369741655 et 6.578903972693759
6.5672755707256885 et 6.5745433219557325
```

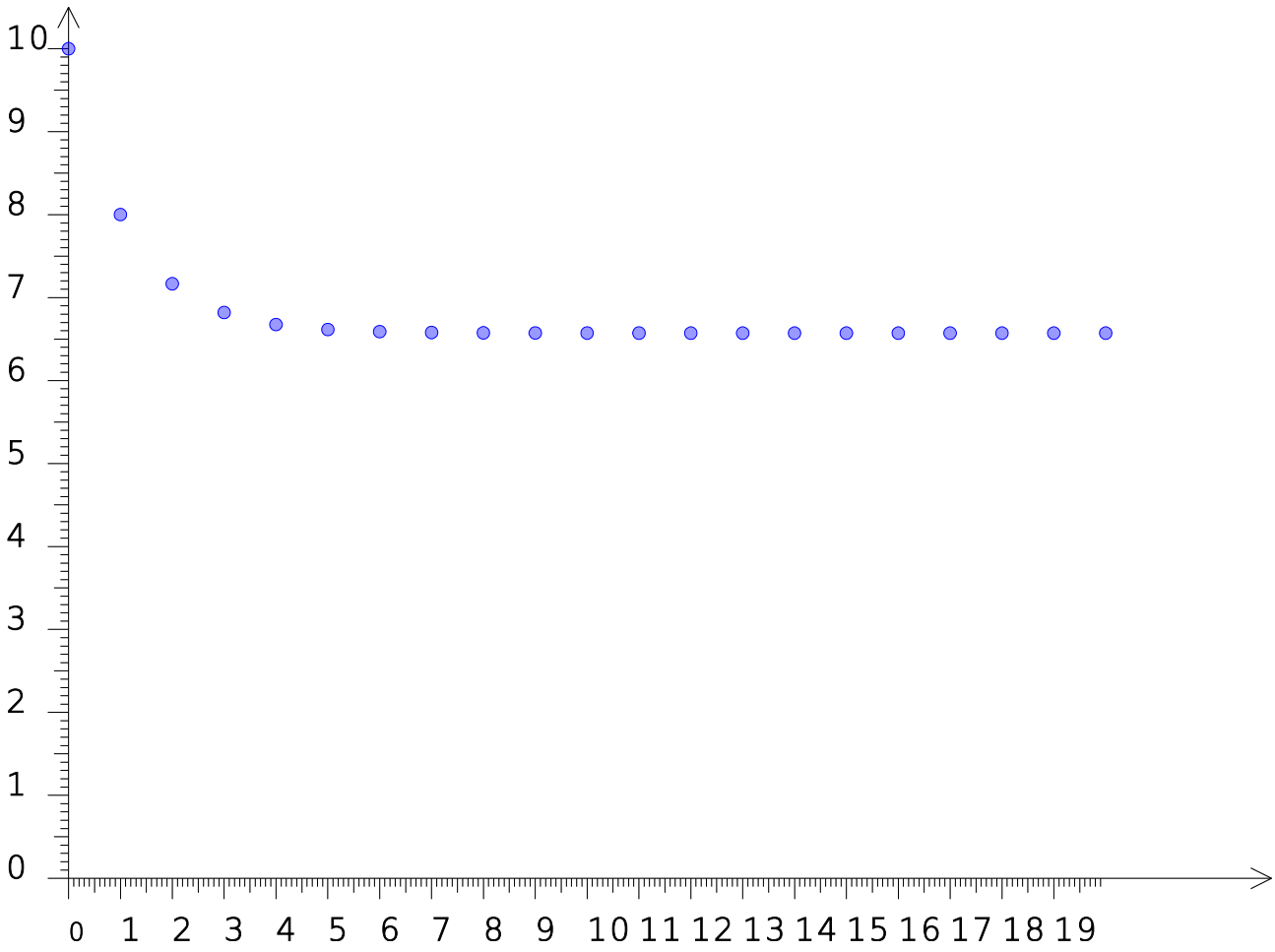
```
6.5696981544690365 et 6.572726384148221
6.570707564362098 et 6.571969326728425
6.57112815181754 et 6.571653886136843
6.571303396590641 et 6.5715224525570175
6.5713764152461 et 6.571467688565423
6.571406839685874 et 6.571444870235592
6.57141951653578 et 6.571435362598162
6.571424798556574 et 6.571431401082567
6.571426999398572 et 6.571429750451069
6.57142791641607 et 6.571429062687945
6.571428298506695 et 6.571428776119976
6.5714284577111215 et 6.571428656716656

Algorithme exécuté en 81 millisecondes
```

Un script à peine plus long permet de représenter graphiquement la suite (v_n) :

```
[u,v] = [2,10]
tableau = [v]
for n in [1..20]
    [u,v] = [(2*u+v)/3,(u+3*v)/4]
    tableau.push v

dessineSuite tableau, 20, 0, 10, 3, 'blue'
```

En fait, au lieu d'afficher v, on l'empile (« push ») dans le tableau, qu'il suffit ensuite de dessiner (20 termes, les ordonnées allant de 0 à 10, les points représentés par des cercles de rayon 3, en bleu)

La représentation graphique montre bien la convergence (et sa rapidité), et la valeur approchée de la limite :



Remarque : On peut aussi traiter cet exemple du point de vue matriciel : En posant $X_n=\begin{pmatrix}u_n\\v_n\end{pmatrix}$ on a $X_{n+1}=\begin{pmatrix}\frac{2}{3} & \frac{1}{3}\\ \frac{1}{4} & \frac{3}{4}\end{pmatrix}X_n$: On peut considérer que la suite de vecteurs X_n est une suite géométrique dont la raison est la matrice $M=\begin{pmatrix}\frac{2}{3} & \frac{1}{3}\\ \frac{1}{4} & \frac{3}{4}\end{pmatrix}$. Or les puissances de cette matrice tendent vers $\begin{pmatrix}\frac{3}{7} & \frac{4}{7}\\ \frac{3}{7} & \frac{4}{7}\end{pmatrix}$ et $\begin{pmatrix}\frac{3}{7} & \frac{4}{7}\\ \frac{3}{7} & \frac{4}{7}\end{pmatrix}\begin{pmatrix}2\\10\end{pmatrix}=\begin{pmatrix}\frac{46}{7}\\ \frac{46}{7}\end{pmatrix}$ d'où la limite commune des deux suites.

Un exemple intéressant de suites adjacentes est fourni par la notion de ***moyenne arithmético-géométrique*** de deux nombres positifs a_0 et b_0 : En posant que $a_{n+1}=\frac{a_n+b_n}{2}$ est la moyenne arithmétique de a_n et b_n et que $b_{n+1}=\sqrt{a_n\times b_n}$ est la moyenne géométrique de a_n et b_n, alors les deux suites a_n et b_n sont adjacentes et leur limite commune est la moyenne arithmético-géométrique de a_0 et b_0. Cette notion, due à Gauss, est intéressante pour deux raisons :

- Les suites convergent extrêmement vite[22]
- La moyenne arithmético-géométrique de 1 et $\frac{1}{\sqrt{2}}$ est liée à une intégrale elliptique

Combinées entre elles, ces deux raisons permettent d'utiliser la moyenne arithmético-géométrique ci-dessus pour calculer rapidement un grand nombre de décimales de π grâce à une amélioration due à Salamin : En posant $c_n = \frac{a_n - b_n}{2}$ alors il suffit de diviser le carré de la moyenne arithmético-géométrique par $1 - \sum_{k=0}^{\infty} 2^n c_n^2$ pour avoir le quart de π.

L'algorithme de Gauss-Salamin se traduit ainsi en CoffeeScript :

```
[a,b,c,s,p]=[1,Math.SQRT2/2,Math.SQRT2/2-1,1,2]
for j in [1..8]
   p *= 2
   [a,b,c] = [(a+b)/2,racine(a*b),(a-b)/2]
   s -= p*c*c

affiche 4*a*a/s
```

La valeur de π ainsi calculée est proche de la valeur interne à $8{,}881784197001252 \times 10^{-16}$ près.

Pour aller plus loin, on utilise une bibliothèque JavaScript appelée big[23] ; on commence par fixer sa variable DP à 1000 (c'est le nombre de décimales auquel chaque étape intermédiaire du calcul est approchée). Ensuite on remplace chaque nombre par son « big » équivalent (par exemple Big(1) au lieu de 1) et chaque opération par sa « big » équivalente (par exemple, .plus(1) au lieu de +1 ; cette notation évite de mettre les numérateurs entre parenthèses et permet de rédiger même cette version avec concision). Le script modifié donne en une vingtaine de secondes les 1000 premières décimales de π :

```
Big.DP = 1000
[a,b,c,s,p]=[Big(1),(Big(1).div(2)).sqrt(),
(Big(1).div(2)).sqrt().minus(1),Big(1),Big(2)]
for j in [1..10]
   p = p.times 2
   [a,b,c] = [(a.plus(b)).div(2),(a.times(b)).sqrt(),(a.minus b).div 2]
   s = s.minus(p.times(c).times(c))

affiche Big(4).times(a).times(a).div(s)
```

4) Méthode de Newton

Pour résoudre assez rapidement des équations, l'algorithme de Raphson est efficace. Il consiste à réécrire l'équation $f(x)=0$ sous la forme $g(x)=x$ où la dérivée de g s'annule en la solution cherchée. Alors la seconde équation a pour solution la limite de la suite récurrente $u_{n+1}=g(u_n)$. La vitesse de convergence est déterminée par la dérivée, d'où le choix de la dérivée nulle qui

22 La vitesse est dite quadratique : http://mathworld.wolfram.com/Brent-SalaminFormula.html ; L'algorithme est décrit ici: http://fr.wikipedia.org/wiki/Formule_de_Brent-Salamin

23 Téléchargeable ici : https://github.com/MikeMcl/big.js mais il est inclus dans l'utilitaire ***alcoffeethmique*** ci-joint

maximise cette vitesse de convergence.

> *La petite île de Delos dans la Grèce antique, était célèbre pour la perfection de l'autel placé dans le temple d'Apollon : Un cube parfait. Mais un jour, malgré la perfection de cet autel, une épidémie de peste vint à frapper l'île. Ses habitants, ignorant quel sort pourrait conjurer la peste, envoyèrent une délégation à Delphes pour demander à la Pythie (la prêtresse d'Apollon qui disait l'avenir sous forme d'énigmes prononcées dans un nuage de fumées toxiques) ce qu'Apollon souhaitait. Elle répondit alors qu'ils devaient doubler l'autel. Les architectes tentèrent alors, sans succès, de construire un nouveau cube dont le volume serait le double de celui du cube actuel.*

(légende rapportée par Eratosthène)

Le problème de la duplication du cube consiste à construire géométriquement $\sqrt[3]{2}$. Dans l'article « duplication » de l'Encyclopédie, d'Alembert écrit

> *Il en est de ce problème comme de celui de la quadrature du cercle, qu'on peut résoudre sinon rigoureusement, du moins aussi exactement qu'on veut, & dont une solution exacte & absolue seroit plus curieuse qu'elle n'est nécessaire.*

Un résultat de Wantzel montre que cette construction est impossible. Mais Raphson peut donner un coup de main aux architectes de Delos, en montrant comment on peut calculer 100 décimales de la racine cubique de 2.

Pour calculer $\sqrt[3]{2}$ qui est solution de $x^3-2=0$, on prend $f(x)=x^3-2$. On calcule $f'(x)=3x^2$ et on réécrit l'équation successivement : $x^3-2=0$, puis $\frac{x^3-2}{3x^2}=0$ puis $x-\frac{x^3-2}{3x^2}=x$: $g(x)=x-\frac{x^3-2}{3x^2}$. On vérifie ensuite, en calculant la dérivée, que si $x^3-2=0$ alors $g'(x)=0$. Enfin, toujours avec un peu de calcul formel, on simplifie $g(x)=\frac{2x^3+2}{3x^2}$. L'algorithme s'en déduit :

- On choisit la valeur de u_0 qu'on veut
- On calcule les termes de la suite $u_{n+1}=\frac{2u_n^3+2}{3u_n^2}$

En CoffeeScript :

```
u = 1
for n in [1..8]
   u = (2*u*u*u+2)/(3*u*u)
   affiche u
```

On vérifie la convergence :

```
Algorithme lancé
1.3333333333333333
```

```
1.2638888888888888
1.2599334934499768
1.2599210500177698
1.259921049894873
1.2599210498948732
1.2599210498948732
1.2599210498948732

Algorithme exécuté en 115 millisecondes
```

L'étape suivante est d'utiliser la bibliothèque big.js pour calculer $\sqrt[3]{2}$ à 100 décimales :

```
Big.DP = 100
u = Big(1)
for n in [1..10]
    u = Big(2).times(u).times(u).times(u).plus(2).div(Big(3).times(u).times(u))

affiche u
```

II/ Suites entières

On a déjà vu un exemple de suite entière : Collatz. On va jouer l'originalité en évitant de parler de la suite de Fibonacci. Mais on va traiter un exemple voisin avec la suite de Lucas. Mais d'abord, un problème de combinatoire où intervient une suite récurrente :

1) Problème de monnaie

De combien de manières différentes peut-on constituer 89 centimes avec uniquement des pièces rouges ?

Pour répondre à cette question, on peut déjà regarder quelques cas plus simples, comme par exemple 8 centimes, que l'on peut constituer de 7 manières différentes :

	5+2+1 = 8
	5+1+1+1 = 8
	2+2+2+2 = 8
	2+2+2+1+1 = 8
	2+2+1+1+1+1 = 8
	2+1+1+1+1+1+1 =8
	1+1+1+1+1+1+1 +1=8

Ainsi, en notant u_n le nombre de manières de constituer n centimes avec des pièces rouges, on a $u_8=7$. Or, dénombrer de cette manière u_{89} est rédhibitoirement long ; on va donc généraliser le problème, et définir toute la suite u_n récursivement pour avoir un algorithme permettant de calculer tous ses termes.

Premiers termes :

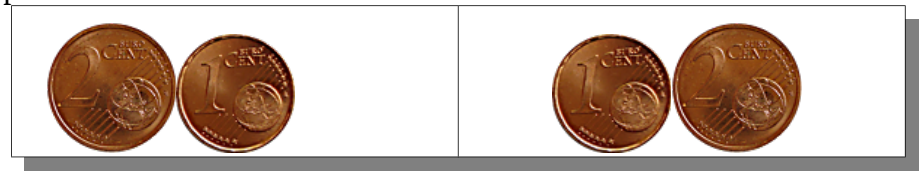
1. $u_0=1$: Il n'y a qu'une manière d'avoir un porte-monnaie vide...
2. $u_1=1$: Avec une seule pièce de 1 centime, on constitue 1 centime et c'est la seule manière
3. $u_2=2$ en effet on a deux manières de constituer 2 centimes :

Mais la première a déjà été vue précédemment ; plus généralement, on retrouve les manières de constituer n centimes comme somme de u_{n-1} et d'autres termes. En effet, il suffit d'ajouter une pièce de 1 à une des décompositions de $n-1$ centimes en pièces rouges, pour avoir une décomposition de n centimes en pièces rouges. Il reste donc à chercher quels sont les autres moyens d'avoir n centimes.

4. Par exemple, il y a également deux manières de faire 3 centimes avec des pièces rouges, on les obtient en ajoutant une pièce de 1 centime à chaque case du tableau ci-dessus

5. Pourtant il y a 3 manières de faire 4 centimes : Outre les deux précédentes (auxquelles on a ajouté 1 centime), il y a aussi 2+2 ; plus généralement, l'un des termes constituant u_n est u_{n-2} (obtenu en ajoutant 2 centimes à chaque décomposition de $n-2$ centimes en pièces rouges)

Mais dans ce cas, pourquoi n'y a-t-il pas aussi 3 manières de faire 3 centimes, au lieu de 2 seulement ? En effet, 3=1+2, et on ajoute aussi une pièce de 2 centimes à l'unique centime de u_1 pour faire 3 centimes. Seulement, ce faisant, on compte deux fois la même décomposition :



Ainsi, lorsqu'on additionne u_{n-1} et u_{n-2} pour avoir u_n , on compte deux fois u_{n-3} et on doit donc le soustraire : Jusque là, $u_n=u_{n-1}+u_{n-2}-u_{n-3}$

6. Vérification avec $u_4=u_3+u_2-u_1=3+2-2=3$: Il y a bien 3 manières de faire 4 centimes avec des pièces rouges : 4=1+1+1+1=2+1+1=2+2.
7. Mais il y a une manière supplémentaire de faire 5 centimes, outre les 1+1+1+1+1=2+1+1+1=2+2+1 obtenues en ajoutant un centime aux trois manières de faire 4 centimes : On peut aussi, tout simplement, mettre une pièce de 5 centimes !
8. Pour constituer 6 centimes, on peut faire 1+1+1+1+1+1, 2+1+1+1+1, 2+2+1+1, 2+2+2 ou 5+1. Plus généralement, on ajoute le terme u_{n-5} aux précédents pour avoir u_n , à partir du rang 5. Mais comme 5+1=1+5, on doit comme ci-dessus soustraire u_{n-6} qui a été compté deux fois.
9. Pour constituer 7 centimes, on a

7=1+1+1+1+1+1+1=2+1+1+1+1+1=2+2+1+1+1=2+2+2+1=5+1+1=5+2 mais on a compté deux fois 5+2=2+5 et on doit donc aussi soustraire u_{n-7} pour avoir u_n .

10. Enfin, comme on a vu précédemment, $u_8=7$ et là encore, 5+2+1=5+1+2=2+1+5=1+2+5 a été compté une quatrième fois dans la somme précédente : On a donc soustrait u_{n-8} une fois de trop pour avoir u_n , et on doit l'additionner à nouveau.

Finalement, u_n est décrite par la relation de récurrence suivante :

- $u_0=1$
- $u_1=1$
- $u_2=2$
- $u_3=2$
- $u_4=3$
- $u_5=4$
- $u_6=5$
- $u_7=6$
- $u_n=u_{n-1}+u_{n-2}-u_{n-3}+u_{n-5}-u_{n-6}-u_{n-7}+u_{n-8}$ pour $n\geqslant 8$

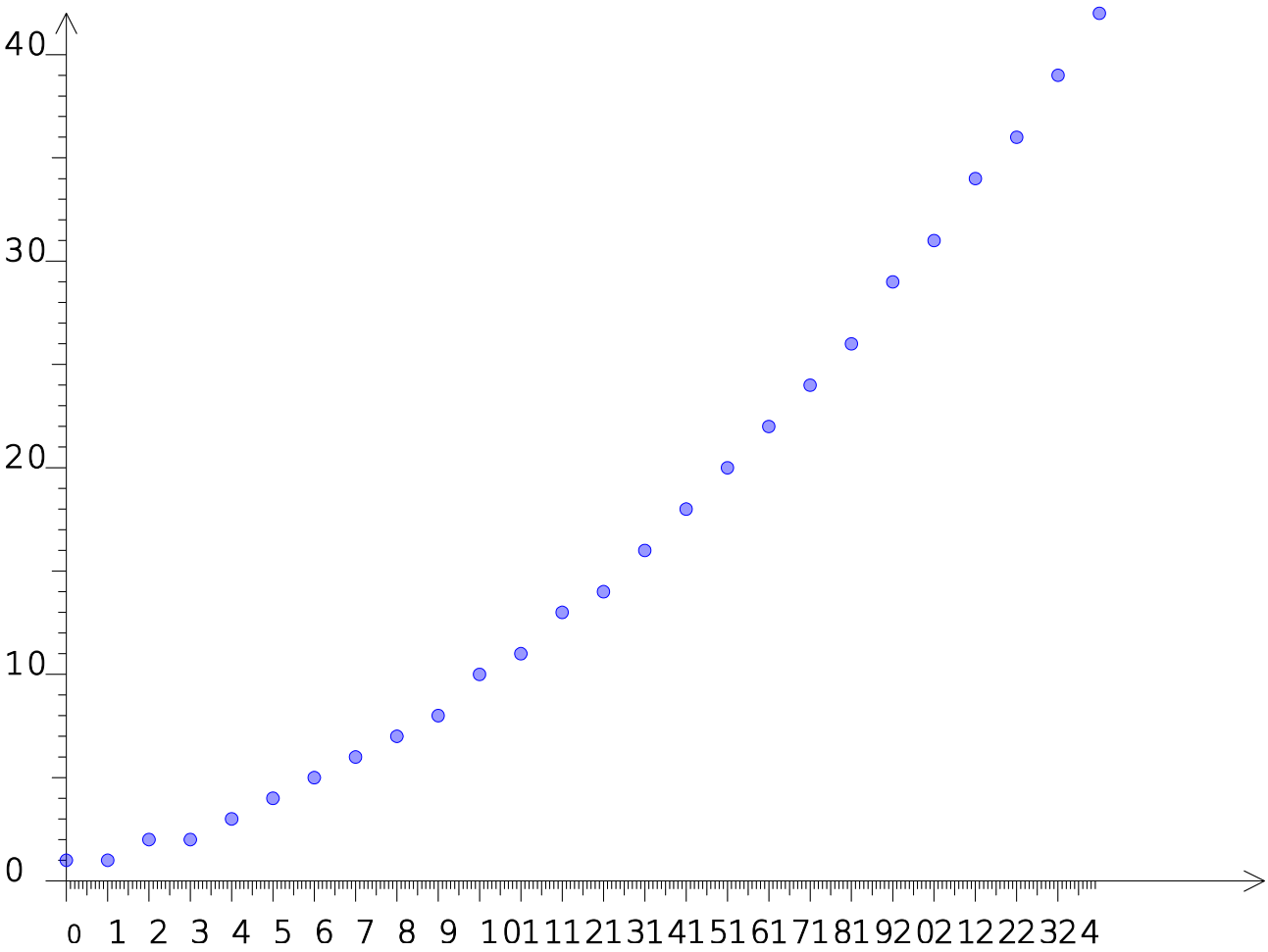
Ceci suffit pour calculer rapidement u_n pour tout n suffisamment petit (y compris à l'aide d'un tableur d'ailleurs). Pour avoir u_{89} il suffit par exemple de faire

```
u =  [1,1,2,2,3,4,5,6,7]
for n in [8..100]
   u[n]=u[n-1]+u[n-2]-u[n-3]+u[n-5]-u[n-6]-u[n-7]+u[n-8]

affiche u[89]
```

Comme les termes successifs ont été stockés dans le tableau u pour n allant jusqu'à 100, on peut représenter graphiquement la suite avec cette variante :

```
u =  [1,1,2,2,3,4,5,6,7]
for n in [8..100]
   u[n]=u[n-1]+u[n-2]-u[n-3]+u[n-5]-u[n-6]-u[n-7]+u[n-8]



dessineSuite u, 25, 0, 1000, 3, 'blue'
```

Ce script produit le graphique suivant :



Les points semblent presque disposés sur une parabole. D'où l'idée (inédite jusqu'ici semble-t-il) de chercher l'équation de cette parabole. Une régression donne l'expression explicite de u_n suivante :

$u_n = E(0{,}05\,n^2 + 0{,}4\,n + 1)$ où $E(x)$ désigne la partie entière de x . Ce qui donne un algorithme encore plus rapide que le précédent pour calculer u_{89} :

```
f= (x) -> troncature(.05*x*x+.4*x+1)

affiche f(89)
```

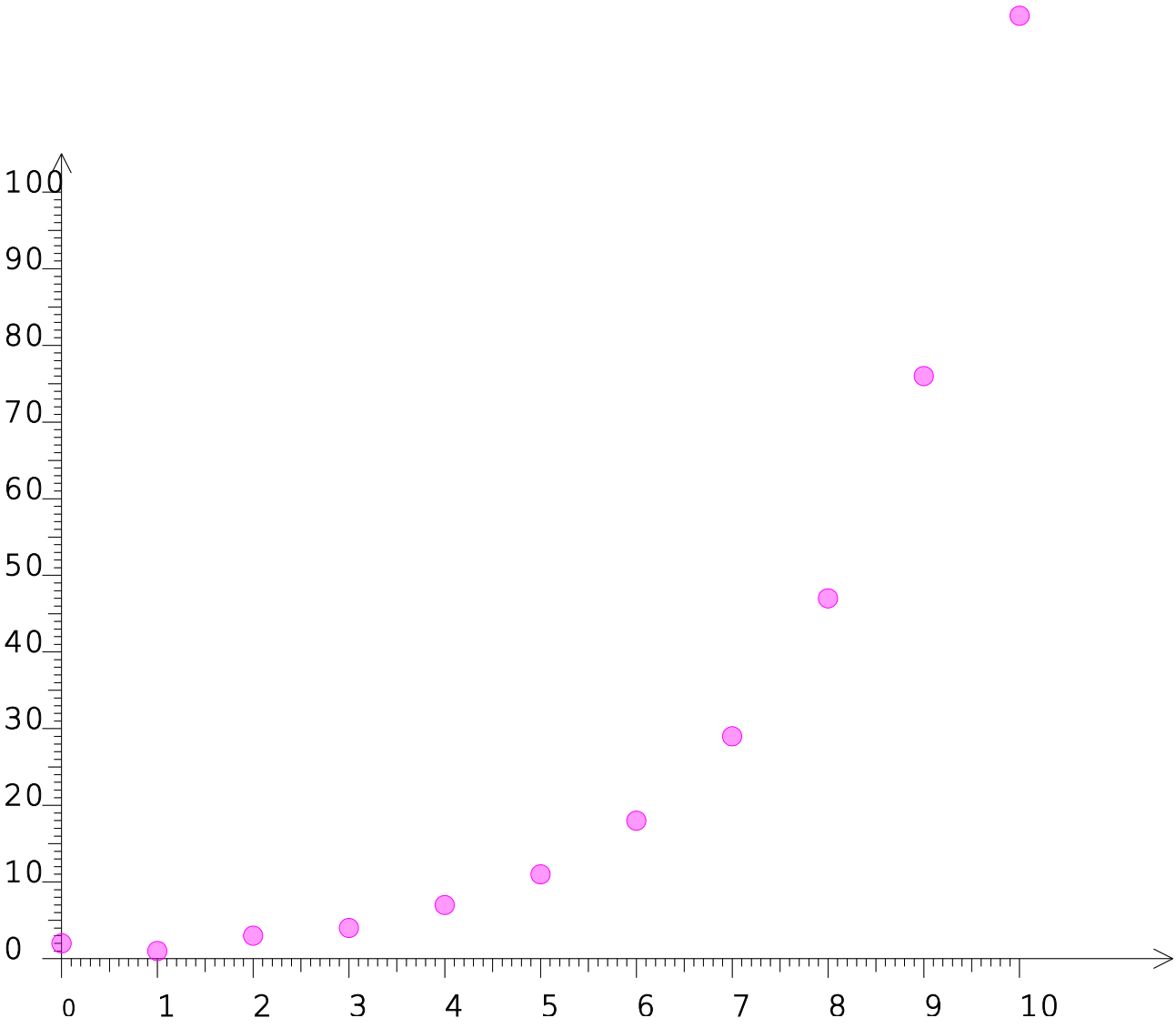
2) Nombres de Fermat

En fait, toute partie de $\mathbb{N}$ est énumérable donc peut être considérée comme une suite croissante d'entiers. Par exemple, les nombres premiers, le nombre de nombres premiers inférieurs à n, etc. Un exemple particulièrement simple (mais qui tend rapidement vers l'infini) est celui des nombres de Fermat, définis comme la suite $F_n = 2^{2^n} + 1$. Ils ont des propriétés intéressantes, comme par exemple le fait que leurs diviseurs sont congrus à 1 modulo les nombres de Fermat qui leur sont inférieurs, et le fait que les 5 premiers d'entre eux sont premiers. Pour les calculer et afficher :

```
for n in [0..4]
    F = puissance(2,puissance(2,n))+1
    affiche F
```

3) Suite de Lucas

La suite de Lucas est définie comme celle de Fibonacci, mais le premier terme est différent puisqu'il vaut 2 au lieu de 1. La suite de Lucas ne diffère pas beaucoup, notamment du point de vue de son comportement asymptotique, de la suite de Fibonacci, comme le montre sa représentation graphique :



D'ailleurs le quotient de deux termes successifs tend toujours vers le nombre d'or :

```
[a,b] = [2,1]
for n in [2..10]
    [a,b] = [b,a+b]
    affiche b/a
```

Ce script donne l'affichage des quotients successifs :

```
Algorithme lancé
3
```

```
1.3333333333333333
1.75
1.5714285714285714
1.6363636363636365
1.6111111111111112
1.6206896551724137
1.6170212765957446
1.618421052631579

Algorithme exécuté en 73 millisecondes
```

Alors quel est l'intérêt de cette suite ? Édouard Lucas propose, comme ça, pour voir, de regarder les éléments de sa suite modulo les valeurs successives de l'indice. En CoffeeScript, comme dans beaucoup de langages de programmation, le symbole de pourcentage (« % ») désigne l'opération « modulo » ; la modification suivante du script précédent permet de tester l'idée de Lucas :

```
[a,b] = [2,1]
for n in [2..20]
    [a,b] = [b,a+b]
    affiche "#{b} modulo #{n} vaut #{b%n}"
```

L'affichage est intéressant à étudier :

```
Algorithme lancé
1 modulo 1 vaut 0
3 modulo 2 vaut 1
4 modulo 3 vaut 1
7 modulo 4 vaut 3
11 modulo 5 vaut 1
18 modulo 6 vaut 0
29 modulo 7 vaut 1
47 modulo 8 vaut 7
76 modulo 9 vaut 4
123 modulo 10 vaut 3
199 modulo 11 vaut 1
322 modulo 12 vaut 10
521 modulo 13 vaut 1
843 modulo 14 vaut 3
1364 modulo 15 vaut 14
2207 modulo 16 vaut 15
3571 modulo 17 vaut 1
5778 modulo 18 vaut 0
9349 modulo 19 vaut 1
15127 modulo 20 vaut 7

Algorithme exécuté en 48 millisecondes
```

Si cela ne suffit pas, on peut se concentrer sur le cas où le reste modulo l'indice est 1 :

```
[a,b] = [2,1]
for n in [2..20]
    [a,b] = [b,a+b]
    if b%n is 1
        affiche "#{b} modulo #{n} vaut #{b%n}"
```

Regardez bien les valeurs de l'indice ; intéressant non ? Bien entendu, Édouard Lucas a démontré le résultat conjecturé (la réciproque par contre, est fausse). On en déduit un test que voici (avec des calculs modulo pour éviter les dépassements de capacité) :

```
test = (n) ->
    [a,b] = [2,1]
    for k in [2..n]
        [a,b] = [b,(a+b)%n]
    b%n is 1

affiche (n for n in [2..200] when test n)
```

Les entiers inférieurs à 200 qui réussissent ce test sont les suivants :

```
Algorithme lancé
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,8
9,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,17
3,179,181,191,193,197,199

Algorithme exécuté en 70 millisecondes
```

Ce test, amélioré dans les années 1930 par Derrick Lehmer, est encore abondamment utilisé de nos jours. Avec l'inconvénient qu'il y a des faux positifs, et l'intervention des probabilités dans l'arithmétique...

4) La suite de Conway

Cette suite est célèbre comme « test d'intelligence » (deviner le prochain terme) :

```
Algorithme lancé
1
11
21
1211
111221
312211
13112221
1113213211
31131211131221
13211311123113112211

Algorithme exécuté en 65 millisecondes
```

Certes, si on devine comment la suite est calculée, on devinera aussi le terme suivant ; mais il est surprenamment difficile de deviner la procédé de construction de cette suite d'entiers. Sauf si on ne craint pas le ridicule en la lisant à voix haute. Bon sang mais c'est bien sûr ! Ce ne sont pas des entiers, ce sont des chaînes de caractères ! Le procédé de construction, en CoffeeScript, est celui-ci :

```
toutou = /(\d)\1*/g
u = "1"
for n in [1..10]
    affiche u
    u = u.replace toutou, (trouvé, symbole) ->
        trouvé.length.toString()+symbole
```

Arrivé là, on est en droit de se demander si cet algorithme explique quoique ce soit sur la construction de la suite... On va essayer ci-dessous de le décrire.

Comme il s'agit finalement de chaînes de caractères, on va se servir de l'outil le plus pratique qui soit pour agir sur des chaînes de caractères : Une ***expression régulière***, ou ***RegEx***. Attendez ! Ne courez pas comme ça, ça ne mord pas ! Enfin presque, on peut comparer une RegEx à un chien renifleur[24], dressé pour chasser une sous-chaîne de caractères et la modifier. La première opération est donc de dresser le chien à trouver une suite de chiffres identiques. Dans le langage des RegEx, un chiffre est représenté par « \d » (comme « décimale » ; le backslash sert à distinguer une décimale de la lettre « d »). Le tout entre parenthèses pour dire à la RegEx de conserver le résultat dans sa mémoire, afin de le modifier plus tard. En bref, « (\d) » permet au chien-RegEx de se souvenir du chiffre en le stockant dans une mémoire qui s'appelle « $1 ». Pour rappeler le contenu de cette mémoire, c'est « \1 » (backslash pour ne pas confondre avec le chiffre 1). Et l'étoile veut dire que le chiffre « \d » peut être suivi par des copies de lui-même en nombre indéterminé. Donc « (\d)\1* » désigne, comme on le voulait, une suite de chiffres identiques entre eux. Pour finir le dressage du chien, on met le tout entre « slashs » et on fait suivre de la lettre g (comme « global ») pour lui demander de faire une recherche exhaustive. Le toutou, une fois dressé, contient donc l'expression régulière **/(\d)\1*/g**. C'est abscons mais c'est concis[25]...

Le premier terme est entre guillemets pour préciser à CoffeeScript que c'est une chaîne de caractères. Dans la boucle, on lance le chien à l'attaque de cette chaîne, qu'il va parcourir caractère par caractère en reniflant. On le lance avec la tâche « replace » qui va remplacer la chaîne trouvée par le résultat d'une fonction anonyme[26] de deux variables appelées trouvé (la chaîne de caractère ramenée par le toutou) et symbole (le caractère obtenu). Alors trouvé.length est le nombre de caractères de la chaîne, qui est converti en chaîne de caractères avec toString() puis on lui ajoute le symbole. Alors « 444 » devient « 34 ».

La conversion inverse (de chaîne de caractères en entiers) se fait avec parseInt. Ce qui permet de vérifier que le quotient de deux termes successifs a une limite, appelée constante de Conway[27].

---

24 En bio-informatique, les caractères sont A, C, G et T (bases nucléiques) et la RegEx simule en fait le comportement d'un ribosome qui est beaucoup plus petit qu'un chien. Mais le principe reste le même, et les RegEx servent à chercher des codons, par exemple dans les tests d'ADN.

25 Néologisme autoréférentiel : C'est « absconcis »...

26 Comme elle ne sert qu'une fois, pas besoin de lui donner un nom ; on appelle ça une « lambda fonction » ; pour toute réclamation sur ce choix de vocabulaire, s'adresser à Stephen Kleene

27 Pour calculer sa limite il faut faire appel au module « big » à cause de la taille des entiers.

5) Générateur congruentiel de Lehmer

Encore une suite d'entiers, mais modulo un nombre premier : Le retour de la suite arithmético-géométrique, promue par Derrick Lehmer[28] dont on a déjà parlé plus haut, mais dans un corps fini. Alors la notion d'inverse modulo p permet de faire des divisions et de résoudre des équations. Or, lorsqu'on étudie une suite arithmético-géométrique, la limite s'obtient par résolution d'une équation, qui fait intervenir une division. Sauf qu'ici, il n'est plus question de limite, puisqu'on évolue dans un espace discret. D'ailleurs, comme il n'y a qu'un nombre fini d'éléments dans le corps des nombres modulo p (il y en a p, dont p-1 sont inversibles), toute suite récurrente est nécessairement périodique[29], et pour une suite récurrente à l'ordre 1 comme c'est le cas pour une suite arithmético-géométrique, la période ne peut excéder p. C'est tout le génie de Lehmer d'avoir trouvé comment faire pour que la période soit p-1, ce qui donne l'illusion du hasard si on choisit bien les coefficients. Le générateur congruentiel de Lehmer est l'un des plus utilisés pour fabriquer des nombres pseudo-aléatoires. On appelle « graine » le premier terme de la suite, et chaque appel à la fonction « alea » fait remplacer la graine par son image par une permutation affine, ce qui revient à passer d'un terme au suivant dans la suite arithmético-géométrique. Pour que le hasard soit bien simulé, il faut

- que p soit premier (maximise la période)
- que le coefficient directeur de la fonction affine itérée soit une racine primitive de l'unité modulo p (là encore, pour que la période soit p-1)
- que ce coefficient directeur soit assez grand pour cacher le caractère arithmético-géométrique de la suite
- que l'ordonnée à l'origine ne montre aucune relation simple avec les autres paramètres.

Par exemple, comme on a vu plus haut que 257 est premier, on peut tester par une boucle sur 256 indices que 19 est une racine primitive 256ème de 1 modulo 257 :

```
p = 257
a = 19
u = a
tableau = [u]
until u%p is 1
    u = (u*a)%p
    tableau.push u
affiche tableau.length
```

L'affichage confirme que la période est 1 :

```
Algorithme lancé
256

Algorithme exécuté en 46 millisecondes
```

Comme 64 n'a pas de relation trop évidente ni avec 257 ni avec 19, on va essayer la suite $u_{n+1}=\left(19\times u_n+64\right) modulo\, 257$ comme générateur congruentiel de Lehmer. Le générateur pseudo-aléatoire fonctionne comme ceci (avec 23 comme graine) :

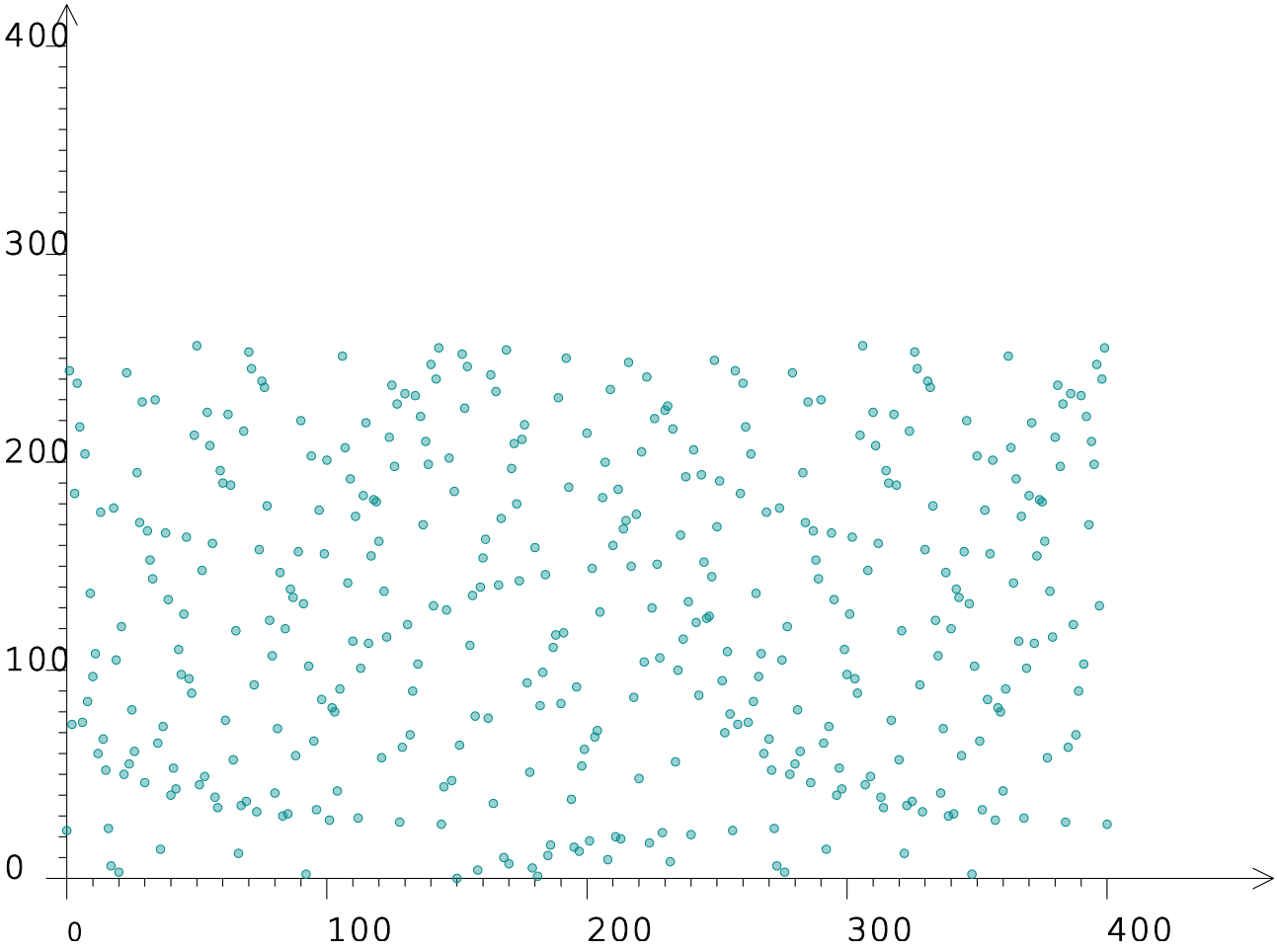
---

28 Qui était donc ce mathématicien au nom d'inspecteur de police allemand et indolent ? C'était lui : http://fr.wikipedia.org/wiki/Derrick_Lehmer

29 Ou prépériodique, c'est-à-dire qu'elle devient périodique à partir d'un certain rang, comme par exemple la suite de Collatz

```
u = 23
tableau = [u]
for n in [1..500]
    u = (19*u+64)%257
    tableau.push u

dessineSuite tableau, 400, 0, 400, 2, 'darkcyan'
```

Le graphique obtenu montre à la fois la périodicité et l'apparence de hasard :



III/ Séries

Chez d'Alembert, les termes d'une suite ont vocation à être additionnés (l'article de l'encyclopédie est titré « série ou suite ») :

Ainsi, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, $\frac{1}{64}$ *, &c forment une suite qui s'approche toujours de la quantité 1, & qui lui devient enfin égale, quand cette suite est continuée à l'infini.*

Bien entendu, la suite géométrique de raison $\frac{1}{2}$ tend vers 0 et non vers 1 ; d'Alembert avait donc en tête la série (somme des termes de la suite) dont la valeur explicite est $\sum_{k=1}^{n}\left(\frac{1}{2}\right)^k=\frac{1-\left(\frac{1}{2}\right)^{n+1}}{1-\frac{1}{2}}-1$ ce qui permet de voir que sa limite est 1 : C'est la série qui tend vers 1, pas la suite.

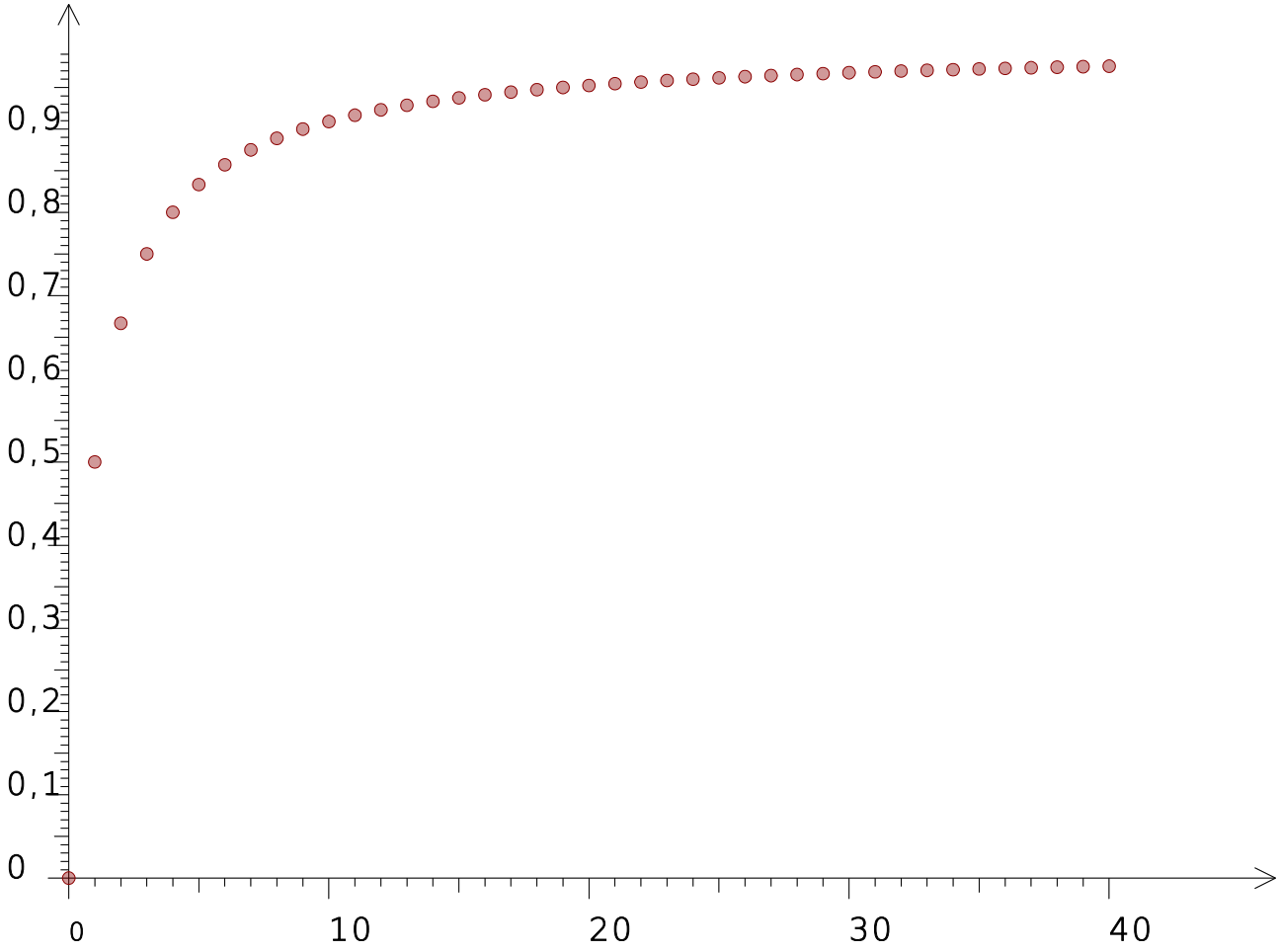
Calculer une somme partielle de série est une chose qu'on peut faire avec un algorithme d'addition et qui peut se montrer utile à bien des calculs. L'idée est, à partir d'une suite u_n , d'en définir une autre par $S_n=\sum_{k=0}^{n} u_k$; en espérant qu'elle converge...

1) Exemple

On a vu plus haut la suite $u_n=\frac{1}{n(n+1)}$ définie à partir du rang 1. Pour calculer la série correspondante $S_n=\sum_{k=1}^{n}\frac{1}{k(k+1)}$ on peut utiliser un algorithme de sommation :

```
u = (n) -> 1/(n*(n+1))
S = 0
tableau = [S]
for k in [1..40]
    S += u(k)
    tableau.push S

dessineSuite tableau, 40, 0, 1, 3, 'darkred'
```

Le graphique obtenu montre la convergence de la série :



Une autre façon de calculer la somme des termes est de mettre d'abord les termes dans un tableau, puis d'appliquer à ce tableau la fonction **laSommeDe** qui effectue la somme :

```
u = (n) -> 1/(n*(n+1))
for n in [1..10]
    S = laSommeDe (u(k) for k in [1..n])
    affiche "La somme des #{n} premiers termes vaut #{S}"
```

L'algorithme est moins rapide que le précédent :

```
Algorithme lancé
La somme des 1 premiers termes vaut 0.5
La somme des 2 premiers termes vaut 0.6666666666666666
La somme des 3 premiers termes vaut 0.75
La somme des 4 premiers termes vaut 0.8
La somme des 5 premiers termes vaut 0.8333333333333334
La somme des 6 premiers termes vaut 0.8571428571428572
La somme des 7 premiers termes vaut 0.8750000000000001
La somme des 8 premiers termes vaut 0.888888888888889
```

```
La somme des 9 premiers termes vaut 0.9
La somme des 10 premiers termes vaut 0.9090909090909091

Algorithme exécuté en 222 millisecondes
```

Comme la série semble converger vers 1, il est logique d'afficher la différence entre 1 et la somme :

```
u = (n) -> 1/(n*(n+1))
for n in [1..10]
    S = laSommeDe (u(k) for k in [1..n])
    affiche 1-S
```

L'affichage montre des inverses d'entiers, alors on peut émettre une conjecture en affichant l'inverse de 1-S :

```
u = (n) -> 1/(n*(n+1))
for n in [1..10]
    S = laSommeDe (u(k) for k in [1..n])
    affiche 1/(1-S)
```

Ceci permet d'émettre une conjecture que l'on peut ensuite démontrer par récurrence, et comme récompense on a une forme explicite de la somme, ce qui permet de faire calculer la limite par CoffeeScript :

```
S = (n) -> 1-1/(n+1)
affiche S(n) for n in [1..10]
affiche "and the limit is ... #{S(Infinity)}"
```

Pour les sommes de termes de suites arithmétiques ou géométriques, les formules du cours permettent aussi de calculer la limite par ce procédé.

2) Nombre de Champernowne

Un autre exemple de nombre défini par une somme de série, est le nombre de Champernowne[30], qui est en fait défini par une chaîne de caractères :

```
champer = '0.'
for n in [1..100]
    champer += n
affiche champer
```

Remarque surprenante : L'inverse du nombre de Champernowne est proche de 8,1 ; pour le vérifier on peut utiliser la bibliothèque Big :

30 Voir ici : http://fr.wikipedia.org/wiki/Constante_de_Champernowne

```
champer = '0.'
for n in [1..100]
    champer += n
champer = Big(champer)
affiche Big(1).div champer
```

3) Exponentielle de 1

Une première méthode pour calculer e est de chercher la limite de la suite $\left(1+\frac{1}{n}\right)^n$; mais cette suite converge assez lentement :

```
for n in [10,100,1000,10000,100000,1000000]
    affiche puissance(1+1/n,n)
```

On voit la convergence vers e mais aussi sa lenteur :

```
Algorithme lancé
2.593742460100002
2.704813829421529
2.7169239322355203
2.718145926824356
2.7182682371975284
2.7182804691564275

Algorithme exécuté en 50 millisecondes
```

La série suivante converge nettement plus vite :

```
for n in [0..10]
    affiche laSommeDe (1/factorielle(k) for k in [0..n])
```

En effet

```

Algorithme lancé
1
2
2.5
2.6666666666666665
2.708333333333333
2.7166666666666663
2.7180555555555554
2.7182539682539684
2.71827876984127
```

```
2.7182815255731922
2.7182818011463845

Algorithme exécuté en 50 millisecondes
```

Pour calculer e à 100 décimales près on peut utiliser ce script :

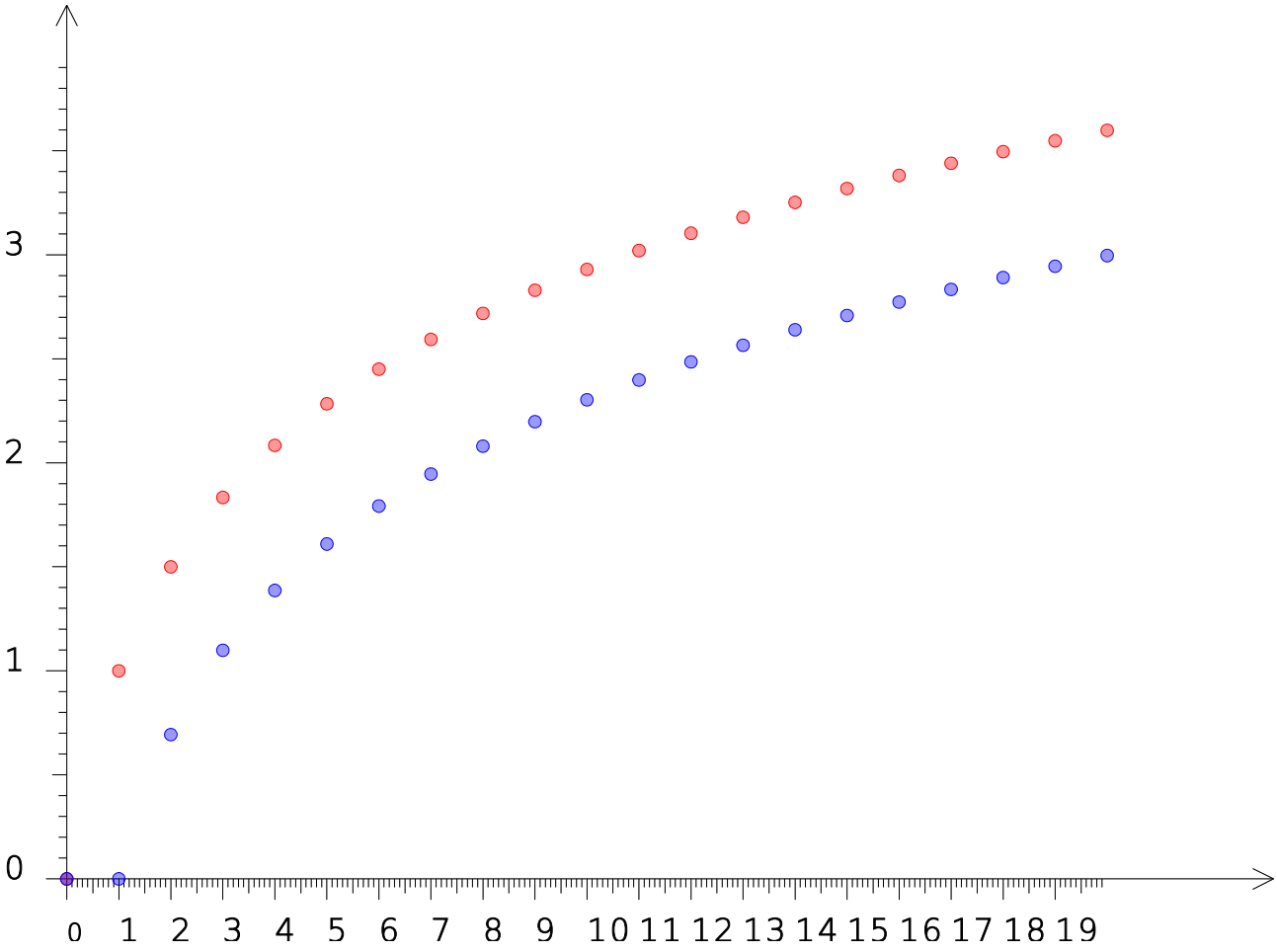
```
Big.DP = 100
e = Big(0)
p = Big(1)
for n in [1..80]
    e = e.plus(Big(1).div(p))
    p = p.times(n)

affiche e
```

4) Constante d'Euler

La suite $\frac{1}{n}$ tend vers 0, mais pas la somme de ses termes, qui tend (quoique lentement) vers l'infini. On peut la représenter graphiquement ainsi :

```
s = 0
tableau = [s]
for n in [1..20]
    s += 1/n
    tableau.push s

dessineSuite tableau, 20, 0, 4, 3, 'red'
```

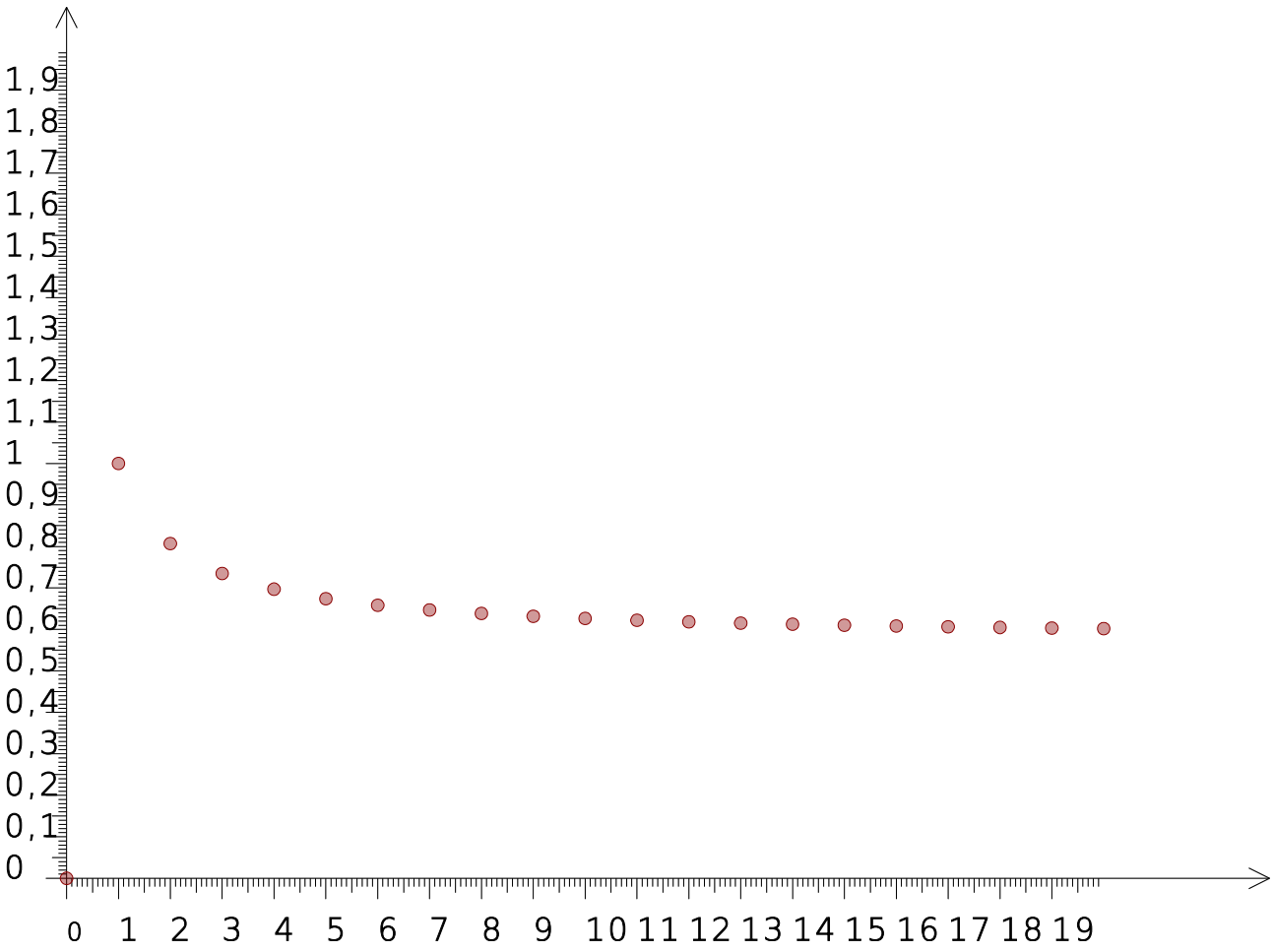
La suite est représentée en rouge ; pour comparaison, on a ajouté sur le graphique, en bleu, la représentation graphique de la suite $\ln(n)$:



Cela suggère que la différence entre ces deux suites converge vers une limite qui est la ***constante d'Euler***. On peut représenter graphiquement la différence :

```
s = 0
tableau = [s]
for n in [1..20]
    s += 1/n
    tableau.push s-ln(n)

dessineSuite tableau, 20, 0, 2, 3, 'darkred'
```

La convergence est lente :



IV/ Autres suites

1) Nombres complexes

Un exemple de suite numérique complexe (les complexes étant considérés comme des nombres) :

```
a = new Complexe 0.9, 0.2
b = new Complexe 0.4, 0.1
z = new Complexe 1, 0
for n in [1..2000]
    z =  z.fois(a).plus(b)

affiche z
```

On voit que la suite (elle est arithmético-géométrique bien entendu) converge vers 0,4+1,8i.

Un exercice du bac S avec une suite récurrente sur $\mathbb{C}$: Antilles-Guyane 2013 (encore!) :

On considère la suite (z_n) à termes complexes définie par $z_0=1+i$ et, pour tout entier naturel n, par

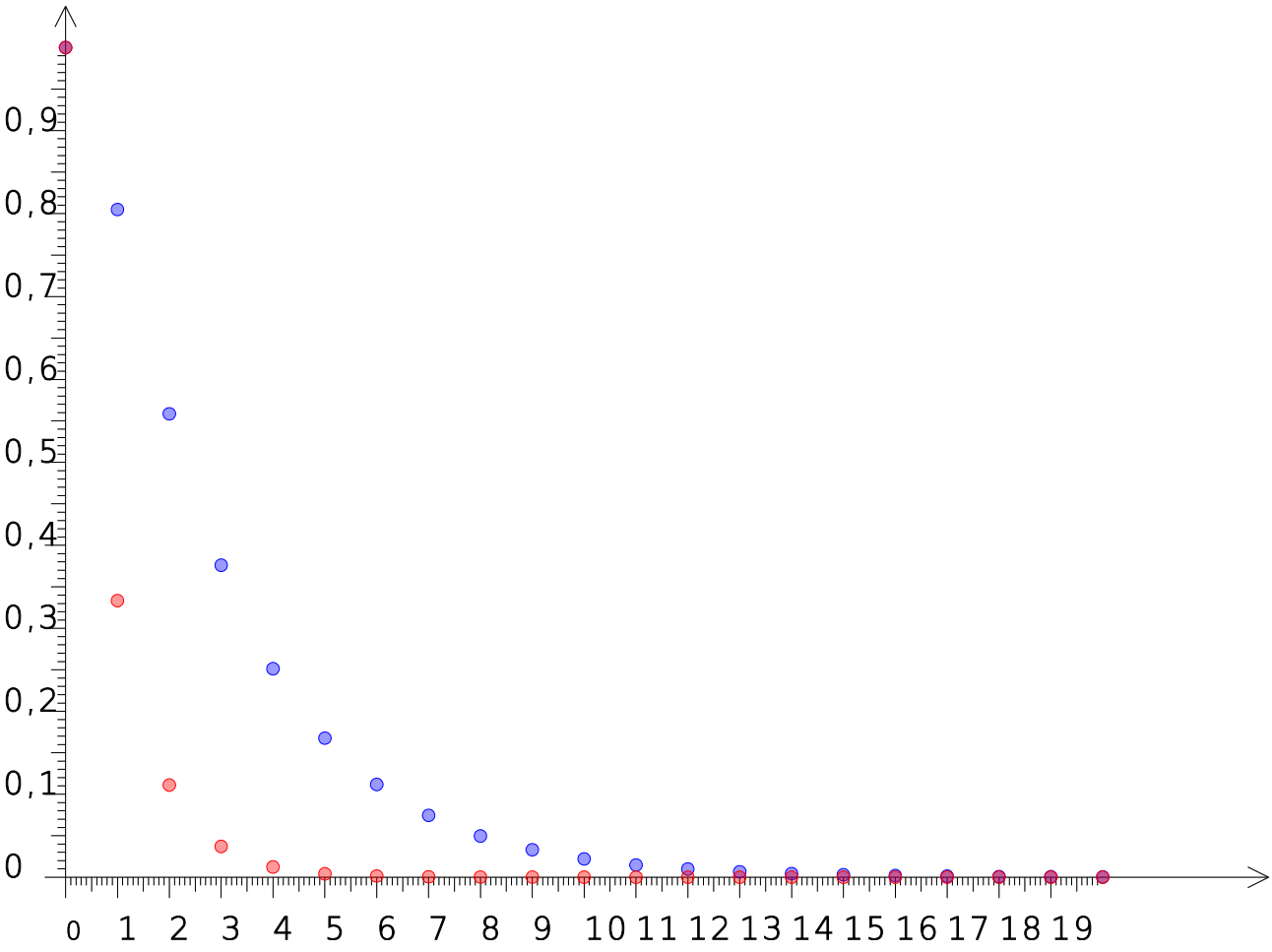
$$z_{n+1}=\frac{z_n+|z_n|}{3}$$

Le but de l'exercice est d'étudier la convergence des parties réelle et imaginaire de z_n . Cela revient en fait à étudier la convergence de z_n elle-même. On peut le faire expérimentalement avec les nombres complexes d'alcoffeethmique :

```
z = new Complexe 1, 1
trois = new Complexe 3, 0
for n in [1..20]
    z = z.plus(new Complexe z.module()).sur(trois)
    affiche "z(#{n}) = #{z}"
```

La difficulté vient de ce que, contrairement à Python, CoffeeScript ne convertit pas automatiquement les réels en complexes pour par exemple additionner un réel avec un complexe. On doit donc faire la conversion comme par exemple à la deuxième ligne.

Le graphique (partie réelle en bleu, partie imaginaire en rouge) montre bien la convergence vers 0 :

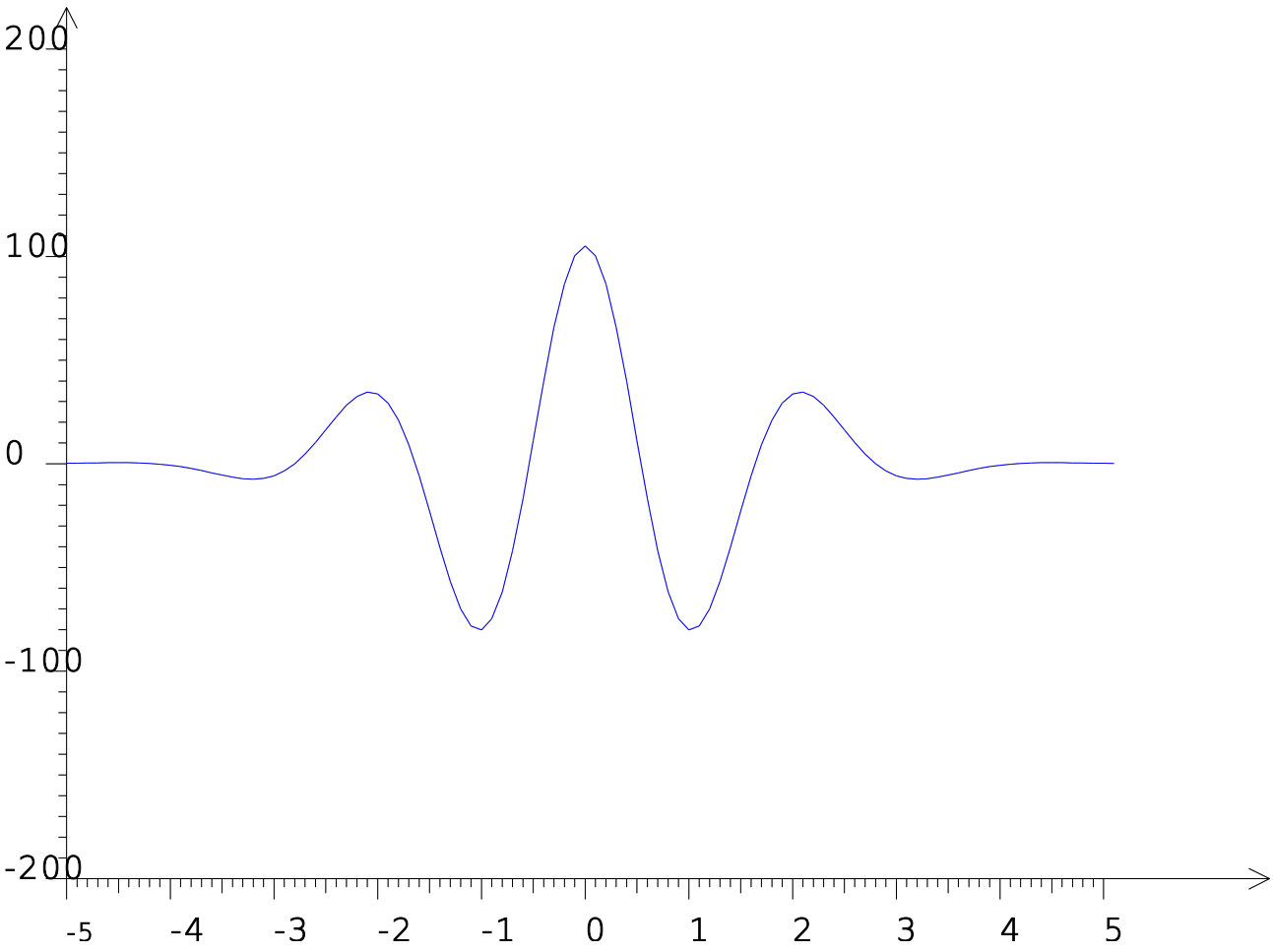


2) Suites de fonctions

Les suites de fonctions peuvent être considérées comme des fonctions à deux variables, la première des deux étant entière ; ou des familles de fonctions, l'entier étant cette fois-ci un paramètre. Par exemple, la suite des approximations polynomiales de l'exponentielle $u_n(x)=\left(1+\frac{x}{n}\right)^n$ peut être

considérée comme la fonction $u(n,x)=\left(1+\frac{x}{n}\right)^n$ des deux variables n et x .

Un exemple de suite de fonctions définie par récurrence est celle des polynômes de Hermite[31] : La dérivée n-ième de la densité de probabilité φ d'une variable aléatoire normale centrée et réduite, est le produit de celle-ci par un polynôme de degré n . Par linéarité, il en est de même de $e^{\frac{-x^2}{n}}$ et on pose $\frac{d}{dx^n}e^{\frac{-x^2}{2}}=(-1)^n H_n(x)e^{\frac{-x^2}{2}}$ où H_n est le n-ième polynôme de Hermite. Voici par exemple la représentation graphique de $H_8(x)e^{\frac{-x^2}{2}}$ qui est donc la dérivée huitième de $e^{\frac{-x^2}{2}}$:



Or les polynômes de Hermite peuvent être calculés par récurrence avec des formules comme $H'_n=n\times H_{n-1}$ ou $H_n(x)=x\times H_{n-1}(x)-(n-1)H_{n-2}(x)$:

```
H = (n,x) ->
   if n < 2
      if n < 1 then 1 else x
   else
      x*H(n-1,x) - (n-1)*H(n-2,x)
```

Les polynômes de Tchebyshev[32] peuvent être calculés de manière similaire :

31 http://fr.wikipedia.org/wiki/Polyn%C3%B4me_d%27Hermite ; voir aussi l'aide en ligne du logiciel Xcas à ce sujet
32 http://fr.wikipedia.org/wiki/Polyn%C3%B4me_de_Tchebychev

```
T = (n,x) ->
   if n < 2
      if n < 1 then 1 else x
   else
      2*x*T(n-1,x) - T(n-2,x)
```

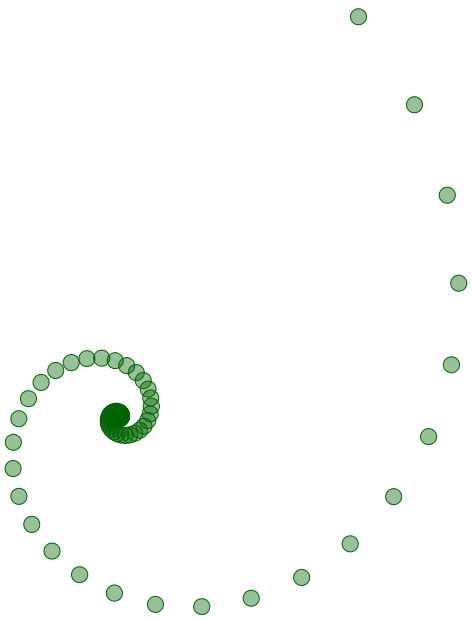
3) Suites de points

Une suite récurrente peut très bien se prêter à du graphisme, pour peu que la fonction à itérer soit une transformation du plan[33] et qu'on représente les termes de la suite (des points) sous forme d'un nuage de points. Ce genre d'activités infographiques mène à la notion d'ensembles de Julia et d'attracteurs étranges. On terminera donc cet article par ces notions.

Tout d'abord, on peut revisiter l'exemple de la suite arithmético-géométrique sur $\mathbb{C}$ vue au IV, cette fois-ci en dessinant sous forme de petits cercles, les images des nombres complexes, plutôt que d'afficher les affixes :

```
effaceDessin()
a = new Complexe 0.9, 0.2
b = new Complexe 0.4, 0.1
z = new Complexe 1, 0
for n in [1..80]
   z =  z.fois(a).plus(b)
   dessineCercle 100+100*z.Re, 100*z.Im, 3, 'darkgreen'
```

En fait, on « calcule » une suite itérée de points, en itérant une transformation du plan, en l'occurrence une similitude directe. Le nuage de points est alors une spirale :



33 Ou de l'espace...

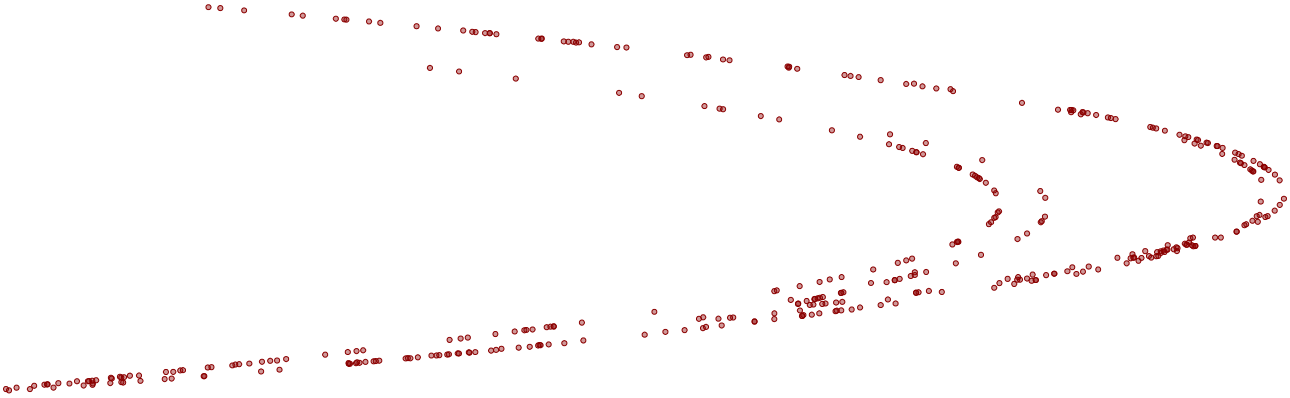
D'autres transformations donnent des nuages de points plus complexes, comme les « attracteurs étranges » présentés ci-dessous.

Attracteur de Hénon

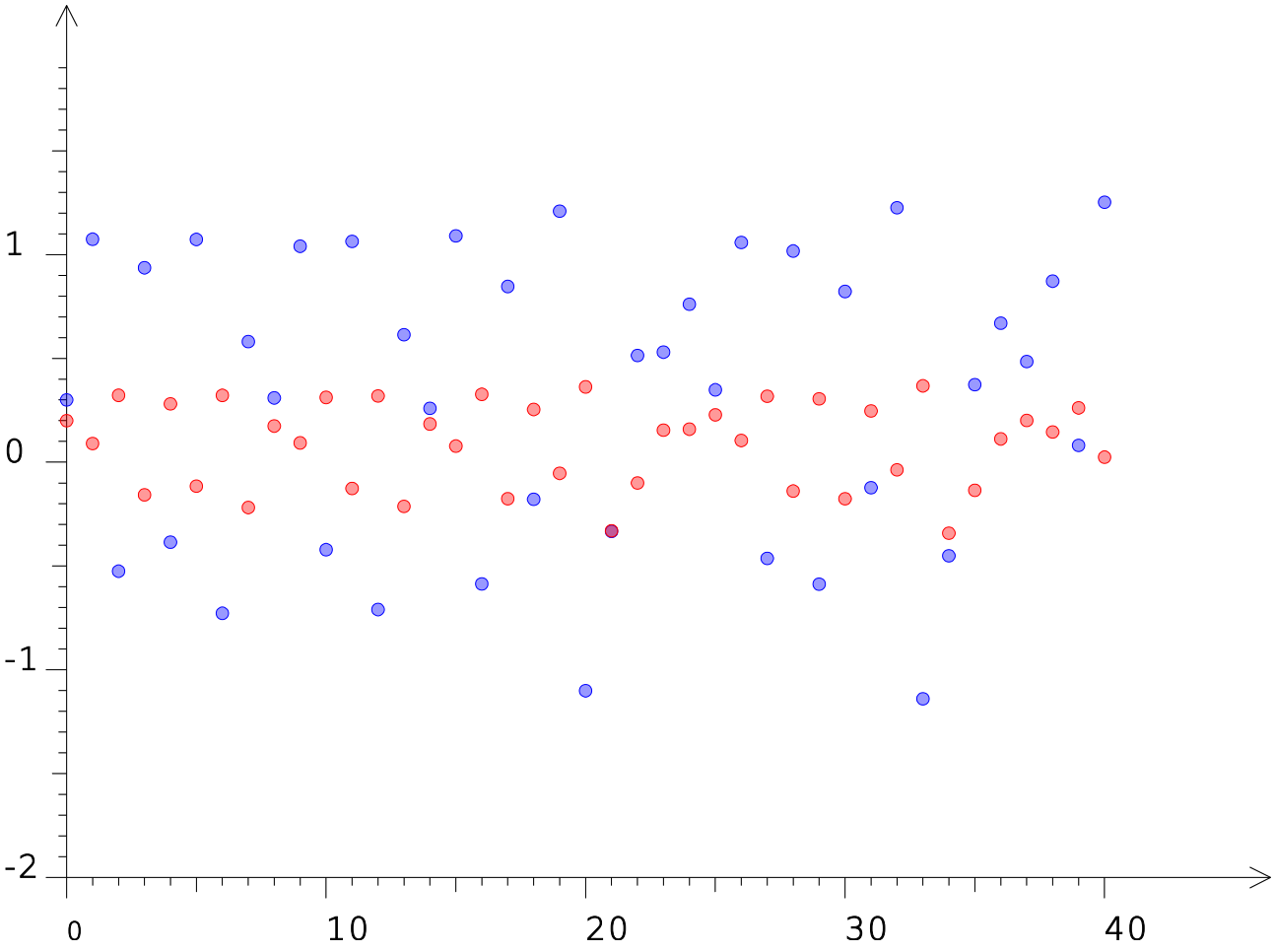
Michel Hénon a découvert cet attracteur il y a plusieurs dizaines d'années, et on ne sait toujours pas s'il est vraiment « étrange » :

```
effaceDessin()
[a, b] = [1.4, 0.3]
[x, y] = [0.3, 0.2]
for n in [1..400]
    [x, y] = [y+1-a*x*x, b*x]
    dessineCercle 320+200*x, 120+200*y, 1, 'darkred'
```

On remarquera la concision permise par l'affectation simultanée de x et y : On met vraiment en œuvre une transformation du plan. Le nuage de points dessiné en rouge foncé est ici :



La représentation graphique des abscisses (en bleu) et des ordonnées (en rouge) montre que ces deux suites ont l'air aléatoires, alors que le nuage de points précédent[34] montrait qu'ils ne sont pas si aléatoires que ça :



Le bonhomme de pain d'épices

En ces temps de Saint-Nicolas[35], il est temps de ressortir[36] ce système dynamique très simple du point de vue des calculs, mais pouvant donner lieu, pour certains choix du point initial, à des choses compliquées comme le remplissage non uniforme d'une surface polygonale ayant des trous.

Voici donc la recette du petit bonhomme en pain d'épices, façon systèmes dynamiques :

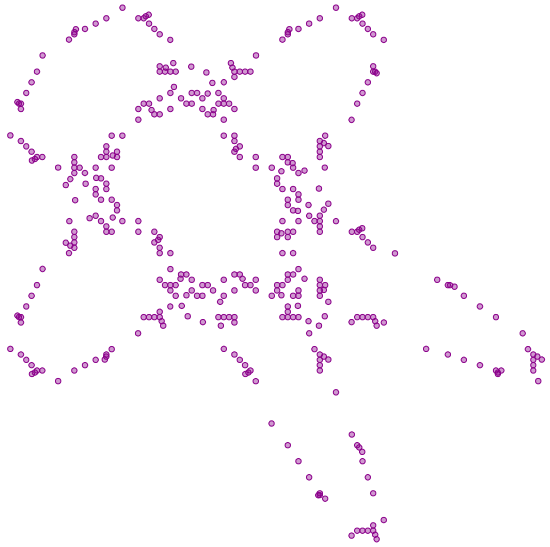
```
effaceDessin()
[x, y] = [0.1, 1.8]
for n in [1..400]
    [x, y] = [1-y+abs(x), x]
    dessineCercle 320+20*x, 240+20*y, 1, 'darkmagenta'
```

34 Voir aussi http://fr.wikipedia.org/wiki/Attracteur_de_H%C3%A9non

35 Une coutume alsacienne consiste à manger, le jour de la Saint-Nicolas, une brioche héritée du Saint-Nicolas en pain d'épices, et ayant la forme d'un petit bonhomme appelé « manala » en dialecte. Cette coutume, aux accents quelque peu anthropophages, est gastronomiquement très agréable, l'excellente brioche étant souvent accompagnée d'un chocolat chaud. C'était ma séquence nostalgie...

36 Voir en anglais : http://en.wikipedia.org/wiki/Gingerbreadman_map mais aussi, la version française et animée : http://db-maths.nuxit.net/CaRMetal/user_galery/diapos/ginger

La couleur « magenta foncé » n'est peut-être pas très appétissante, mais on distingue bien le petit bonhomme dans le nuage de points :



La fée Clochette

Après la Saint-Nicolas, Noël et ses fées, ainsi que les promotions des produits Walt Disney : Par analogie avec la trajectoire de Clochette autour du château de Louis II que l'on voit à chaque générique de Walt Disney, les créateurs de cet attracteur étrange[37] lui ont donné le nom de « Tinkerbell » qui est précisément le nom original de Clochette. Voici la traduction en CoffeeScript de l'algorithme[38] :

```
effaceDessin()
[a, b, c, d] = [0.9, -0.6013, 2, 0.5]
[x, y] = [-0.72, -0.4]
for n in [1..400]
    [x, y] = [x*x-y*y+a*x+b*y, 2*x*y+c*x+d*y]
    dessineCercle 320+100*x, 240+100*y, 1, 'darkgreen'
```

37 Voir ici : http://en.wikipedia.org/wiki/Tinkerbell_map

38 En vert parce que c'est la couleur de la robe de la fée

Et le nuage obtenu :

Alain Busser
LPO Roland-Garros
Le Tampon