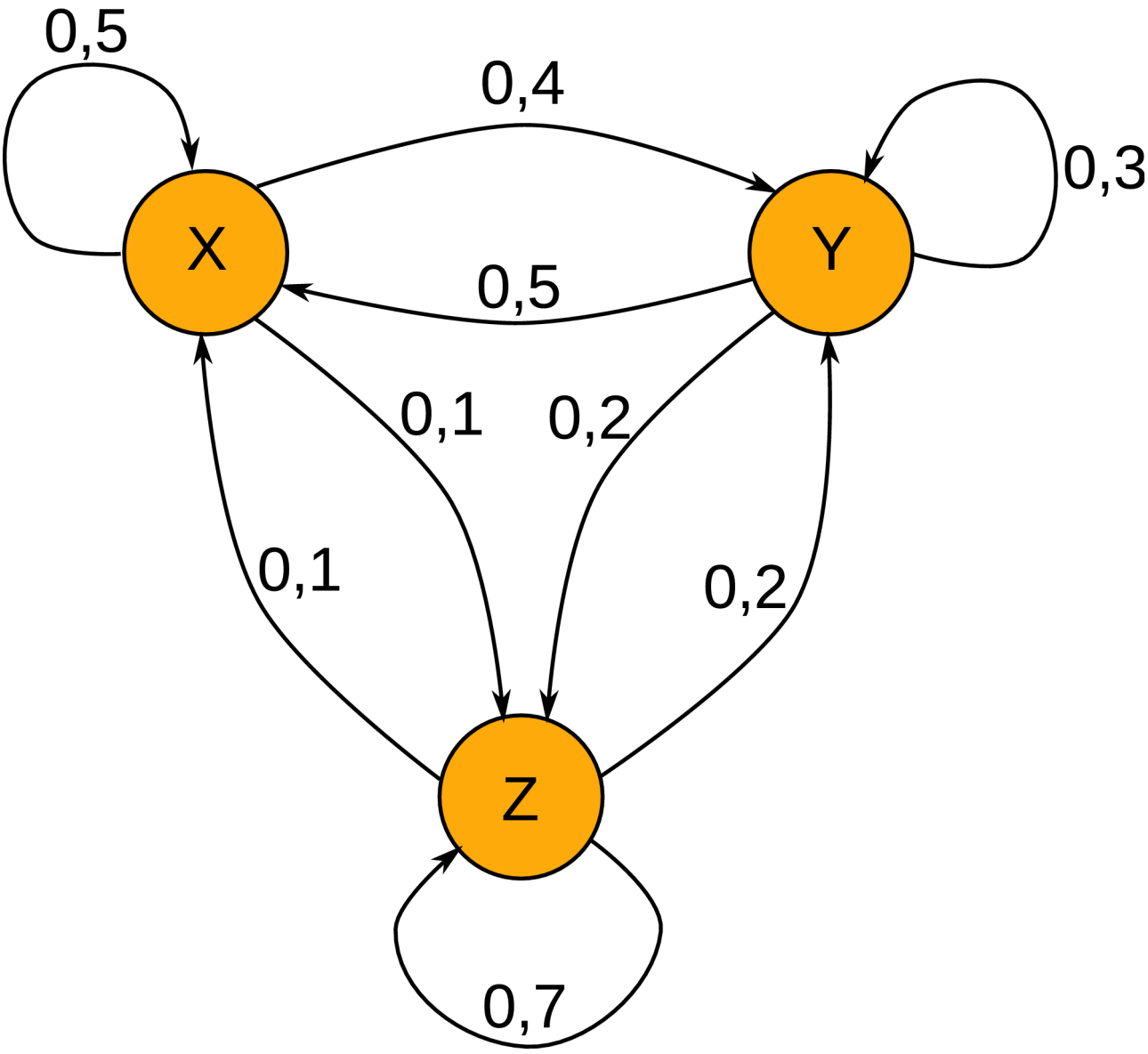
Corrigé du sujet de spécialité bac S Pondichery 2014 avec CoffeeScript

CoffeeScript n'ayant pas de fonctions spéciales pour faire de mathématiques, on va lui adjoindre[1] une petite bibliothèque JavaScript de calcul matriciel, appelée sylvester.js. Bien que cette bibliothèque ait été créée pour faire du graphisme, elle offre toutes les fonctionnalités de calcul matriciel qui sont nécessaires à cet exercice. Dans cet article, le sujet sera traité avec ***alcoffeethmique*** ci-joint, et avec Xcas pour le calcul formel et la théorie spectrale.

I/ Chaîne de Markov

Une chaîne de Markov possède un petit nombre d'états, ici X, Y et Z, et à chaque « top » d'un chronomètre, elle change d'état avec les probabilités conditionnelles données dans l'énoncé. Par exemple, on traduit l'évènement X_n « la marque X est utilisée le mois n » par « le mois n, la machine de Markov est dans l'état X ». Alors la machine à états finis peut se décrire par un graphe ou par la matrice probabiliste de ce graphe. Voici le graphe :



1 C'est le cas de le dire...

Les probabilités ont été lues dans l'énoncé. La matrice de probabilité du graphe est formée par ces probabilités conditionnelles qui ornent les arêtes du graphe.

II/ Question 1

1. Calcul de x_{n+1}

Le graphe aide à répondre à la question 1 : Il y a 3 chemins menant à X :

- le premier venant de X avec une probabilité de 0,5 ;
- le second venant de Y avec une probabilité de 0,5 ;
- le troisième venant de Z avec une probabilité de 0,1.

Par conséquent $x_{n+1}=0{,}5\,x_n+0{,}5\,y_n+0{,}1\,z_n$ [2].

On remarque que 0,5+0,5+0,1 n'est pas égal à 1 : Les probabilités calculées ci-dessus sont bel et bien des probabilités conditionnelles. Par contre, on va voir ci-dessous que les colonnes de la matrice sont bien de somme 1.

2. La matrice de Markov

L'égalité $x_{n+1}=0{,}5\,x_n+0{,}5\,y_n+0{,}1\,z_n$ et les analogues pour y_{n+1} et z_{n+1} peuvent s'écrire matriciellement, en posant $E_n=\begin{pmatrix}x_n\\y_n\\z_n\end{pmatrix}$ (vecteur d'état) et $E_{n+1}=\begin{pmatrix}x_{n+1}\\y_{n+1}\\z_{n+1}\end{pmatrix}=\begin{pmatrix}0{,}5 & 0{,}5 & 0{,}1\\0{,}4 & 0{,}3 & 0{,}2\\0{,}1 & 0{,}1 & 0{,}7\end{pmatrix}\begin{pmatrix}x_n\\y_n\\z_n\end{pmatrix}$.

L'écriture abrégée $E_{n+1}=M\,E_n$ où $M=\begin{pmatrix}0{,}5 & 0{,}5 & 0{,}1\\0{,}4 & 0{,}3 & 0{,}2\\0{,}1 & 0{,}1 & 0{,}7\end{pmatrix}$, permet

- de montrer l'existence d'une distribution de probabilité limite lorsque n devient grand
- de calculer la limite (par résolution d'un système)
- de déterminer explicitement les suites x_n , y_n et z_n par diagonalisation de M.

Sylvester permet aisément d'entrer une matrice avec $M, et de l'afficher :

```
#la matrice du graphe
M = $M [[0.5,0.5,0.1],[0.4,0.3,0.2],[0.1,0.2,0.7]]
affiche M.inspect()
```

2 Une autre démonstration passe par les probabilités conditionnelles, en constatant que l'évènement X_{n+1} est la réunion disjointe des évènements $X_n\cap(X_{n+1}/X_n)$, $Y_n\cap(X_{n+1}/Y_n)$ et $Z_n\cap(X_{n+1}/Z_n)$ où le trait incliné doit être lu « sachant que ». Or la probabilité de $X_n\cap(X_{n+1}/X_n)$ est le produit de la probabilité x_n de X_n par la probabilité conditionnelle $P(X_{n+1}/X_n)=P_{X_n}(X_{n+1})=0{,}5$, soit $0{,}5\,x_n$. Idem pour les probabilités des deux autres évènements, et comme les trois évènements sont disjoints, la probabilité de X_{n+1} est donc la somme des probabilités des trois évènements disjoints, soit $x_{n+1}=0{,}5\,x_n+0{,}5\,y_n+0{,}1\,z_n$.

Le script produit l'effet suivant :

```
Algorithme lancé
[0.5, 0.5, 0.1]
[0.4, 0.3, 0.2]
[0.1, 0.2, 0.7]

Algorithme exécuté en 324
millisecondes
```

3. Puissances de la matrice

De même, l'état initial (donné à la question 2 de l'énoncé) est entré avec « $V » puisque c'est un vecteur :

```
M = $M [[0.5,0.5,0.1],[0.4,0.3,0.2],[0.1,0.2,0.7]]
V = $V [0.5,0.3,0.2]
```

Et pour multiplier la matrice M par le vecteur V, on utilise « .x » (un point suivi d'un x parce que cette lettre ressemble à un signe de multiplication). En le faisant dans une boucle, on répond aux questions 2 et 3 :

```
M = $M [[0.5,0.5,0.1],[0.4,0.3,0.2],[0.1,0.2,0.7]]
V = $V [0.5,0.3,0.2]
for n in [1..8]
    V = M.x V
    affiche V.inspect()
```

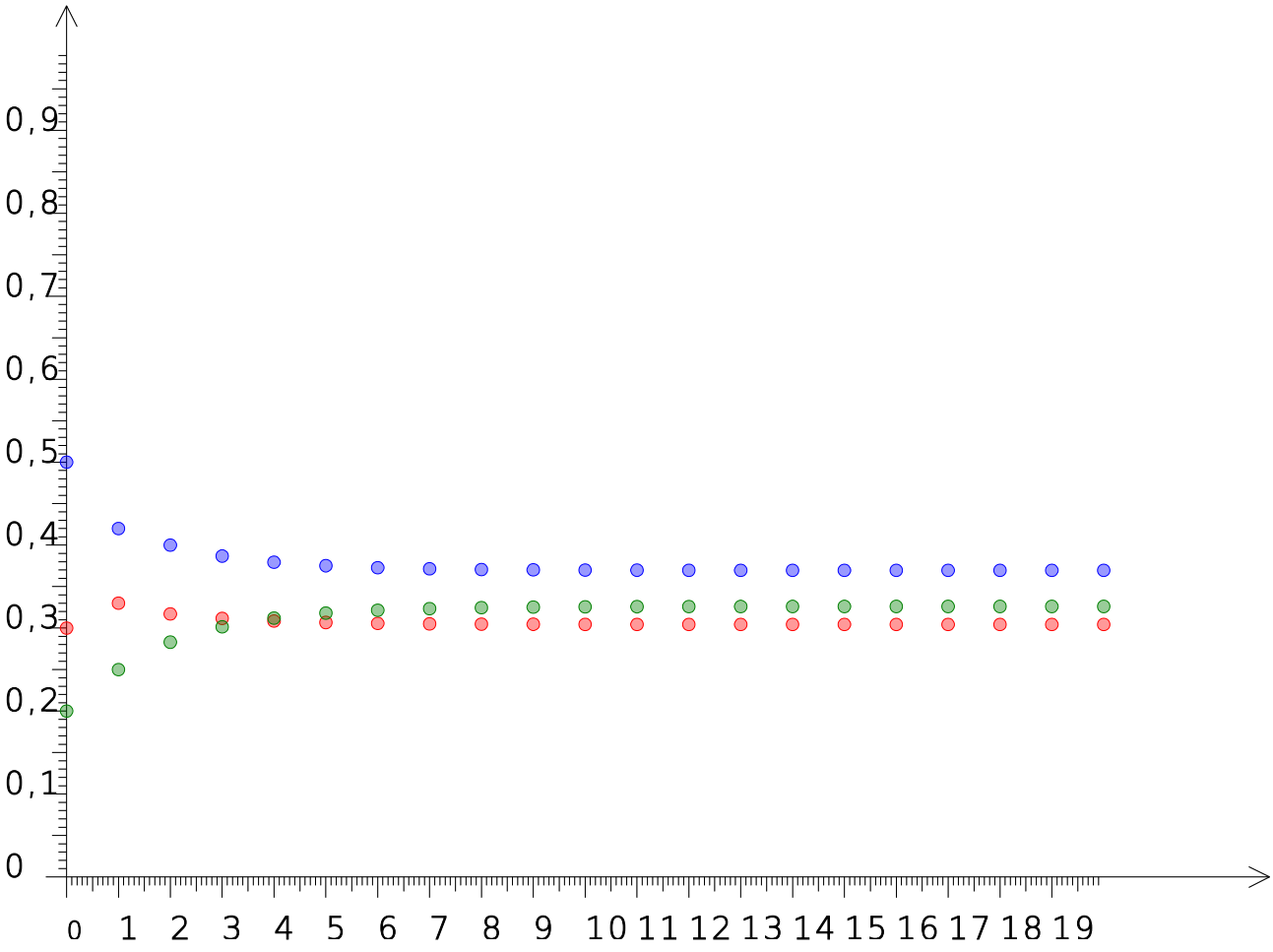
L'affichage des vecteurs successifs montre la convergence rapide vers la probabilité d'équilibre :

```
Algorithme lancé
[0.42, 0.33, 0.25]
[0.4, 0.317, 0.283]
[0.3868, 0.3117, 0.3015]
[0.3794, 0.3085, 0.3121]
[0.3752, 0.3067, 0.3181]
[0.3728, 0.3057, 0.3215]
[0.3714, 0.3051, 0.3235]
[0.3706, 0.3048, 0.3246]

Algorithme exécuté en 86
millisecondes
```

On peut d'ailleurs représenter graphiquement les suites x_n (en bleu), y_n (en rouge) et z_n (en vert). Par exemple x_n est accessible par V.elements[0], les autres étant aux indices respectifs 1 et 2. Le dessin s'obtient par ce genre de script :

```
M = $M [[0.5,0.5,0.1],[0.4,0.3,0.2],[0.1,0.2,0.7]]
V = $V [0.5,0.3,0.2]
x = [0.5]
for n in [1..20]
    V = M.x V
    x.push V.elements[0]
dessineSuite x, 20, 0, 1, 3, "blue"
```

Superposées, les représentations graphiques des suites montrent leur convergence vers des limites différentes entre elles, qu'il reste à trouver :



4. Théorie spectrale

CoffeeScript ne diagonalisant pas les matrices, on va rapidement faire appel à un logiciel le faisant, ici Xcas. On entre donc

```
M := [[0.5,0.5,0.1],[0.4,0.3,0.2],[0.1,0.2,0.7]]
```

Puis

```
eigenvals(M)
```

Pour avoir les trois valeurs propres :

```
(1.0,-0.0701562118716,0.570156211872)
```

On constate que la plus grande valeur propre est 1 et que les autres valeurs propres[3] sont inférieures à 1 en valeur absolue[4]. En mettant les vecteurs propres dans une matrice P, on a donc $M=PDP^{-1}$ et on peut montrer par récurrence que $M^n=PD^nP^{-1}$. Ce qui permet donc d'avoir une forme explicite pour M^n et donc pour x_n , y_n et z_n . Toutefois, cette expression est compliquée[5] et la suite de l'exercice[6] vise à en trouver une autre, plus simple, par une autre méthode.

Une fois qu'on a admis l'existence d'un état d'équilibre vérifiant $ME=E$, on peut le déterminer en résolvant le système $ME=E$ qui peut s'écrire $\begin{pmatrix}0{,}5 & 0{,}5 & 0{,}1\\ 0{,}4 & 0{,}3 & 0{,}2\\ 1 & 1 & 1\end{pmatrix}\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}x\\y\\1\end{pmatrix}$ [7], ou

---

3 Xcas étant un logiciel de calcul formel, fournit les valeurs exactes $r=\frac{5-\sqrt{41}}{20}$ et $s=\frac{5+\sqrt{41}}{20}$

4 Ce phénomène est général, pour toute matrice stochastique c'est-à-dire dont les colonnes sont de somme 1. C'est de là que vient la convergence du vecteur d'état vers une distribution de probabilité limite.

5 Xcas trouve pour x_n : `(-25*x0*sqrt(41)*exp(-n*ln(20)+n*ln(sqrt(41)+5))+25*x0*sqrt(41)*exp(n*ln(-(sqrt(41)-5))-n*ln(20))+1189*x0*exp(-n*ln(20)+n*ln(sqrt(41)+5))+1189*x0*exp(n*ln(-(sqrt(41)-5))-n*ln(20))+1394*x0+205*y0*sqrt(41)*exp(-n*ln(20)+n*ln(sqrt(41)+5))-205*y0*sqrt(41)*exp(n*ln(-(sqrt(41)-5))-n*ln(20))-697*y0*exp(-n*ln(20)+n*ln(sqrt(41)+5))-697*y0*exp(n*ln(-(sqrt(41)-5))-n*ln(20))+1394*y0-163*z0*sqrt(41)*exp(-n*ln(20)+n*ln(sqrt(41)+5))+163*z0*sqrt(41)*exp(n*ln(-(sqrt(41)-5))-n*ln(20))-697*z0*exp(-n*ln(20)+n*ln(sqrt(41)+5))-697*z0*exp(n*ln(-(sqrt(41)-5))-n*ln(20))+1394*z0)/3772` ; et des résultats analogues pour y_n et z_n ...

6 La suite de l'exercice vise aussi à démontrer l'existence d'une distribution limite. Mais déjà avec une matrice stochastique d'ordre 3, on peut le faire : La matrice $D=\begin{pmatrix}1 & 0 & 0\\ 0 & r & 0\\ 0 & 0 & s\end{pmatrix}$ ci-dessus ayant des valeurs inférieures ou égales à 1 en valeur absolue sur sa diagonale, sa puissance n-ième $D^n=\begin{pmatrix}1 & 0 & 0\\ 0 & r^n & 0\\ 0 & 0 & s^n\end{pmatrix}$ tend vers $D^\infty=\begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{pmatrix}$. Alors la matrice $PD^\infty P^{-1}$ est la limite de M^n lorsque n tend vers l'infini. Autre résultat remarquable : $PD^\infty P^{-1}E_0$ ne dépend pas de l'état initial E_0 mais seulement de la somme de ses termes qui vaut 1. La distribution de probabilité limite est donc indépendante de E_0 . Ce qu'on observe algorithmiquement.

7 Les trois équations de $ME=E$ ne sont pas indépendantes, alors on remplace la troisième par $x+y+z=1$

$\begin{pmatrix} -0,5 & 0,5 & 0,1 \\ 0,4 & -0,7 & 0,2 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. En notant $R=\begin{pmatrix} -0,5 & 0,5 & 0,1 \\ 0,4 & -0,7 & 0,2 \\ 1 & 1 & 1 \end{pmatrix}$, la solution de ce système s'obtient en multipliant $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ à gauche par l'inverse de R : $\begin{pmatrix} x \\ y \\ z \end{pmatrix}=R^{-1}\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. Le script suivant fait l'affaire :

```
R = $M [[-0.5,0.5,0.1],[0.4,-0.7,0.2],[1,1,1]]
affiche (R.inverse().x $V([0,0,1])).inspect()
```

Il donne la distribution d'équilibre en valeurs approchées[8] :

```
Algorithme lancé
[0.36956521739130443,
0.3043478260869566,
0.3260869565217391]

Algorithme exécuté en 13
millisecondes
```

Petite remarque géométrique : La suite des vecteurs d'état (x_n, y_n, z_n) peut être représentée graphiquement comme un nuage de points par un logiciel de géométrie dynamique en 3D, surtout si celui-ci est muni d'un langage de script. Mais comme $x_n+y_n+z_n=1$, les points de ce nuage sont tous coplanaires, et même presque alignés (voir ci-dessous). La troisième dimension n'apporte donc pas grand-chose en terme de représentations. Alors, puisque tout se passe dans le plan d'équation $x+y+z=1$, on peut traiter le problème en dimension 2, dans le plan d'équation $x+y+z=1$ ou dans son projeté sur le plan xOy : C'est ce qu'on va faire dans la suite de l'exercice, en utilisant la question (b) de la partie 1[9] :

III/ Question 2

1. Remarque géométrique

En dimension 2, on n'a plus qu'une matrice d'ordre 2 au lieu de 3, mais le problème est devenu affine puisqu'il y a des constantes additionnées à x_{n+1} et y_{n+1} . On résume les résultats précédents à $\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix}=\begin{pmatrix} 0,4 & 0,4 \\ 0,2 & 0,1 \end{pmatrix}\begin{pmatrix} x_n \\ y_n \end{pmatrix}+\begin{pmatrix} 0,1 \\ 0,2 \end{pmatrix}$. On itère donc une affinité du plan affine et plus une application affine de l'espace vectoriel de dimension 3. Ce qui soulève maintes questions d'ordre géométrique comme

8 Le dénominateur de ces nombres est 46, on peut donc les multiplier par 46 pour vérifier que les trois produits sont entiers, et donc obtenir les valeurs exactes de ceux-ci. Ce sera fait dans la troisième partie.

9 Remplacer z_n par $1-x_n-y_n$ dans x_{n+1} et y_{n+1} permet en effet d'obtenir $x_{n+1}=0,4\,x_n+0,4\,y_n+0,1$ et $y_{n+1}=0,2\,x_n+0,1\,y_n+0,2$

- Quels sont les points fixes de cette transformation ?
- Quelle est l'image d'une droite ?
- Quelle est l'image d'un cercle ?
- Quelles sont les droites fixes ?
- Etc.

Mais comme le déterminant de la matrice est -0,04 on sait déjà que l'affinité n'est pas une isométrie.

2. Remarque algébrique

En posant $A=\begin{pmatrix}0,4 & 0,4\\ 0,2 & 0,1\end{pmatrix}$ et $B=\begin{pmatrix}0,1\\ 0,2\end{pmatrix}$ comme dans l'énoncé, on a $U_{n+1}=AU_n+B$ et on étudie donc une suite arithmético-géométrique, à ceci près que la raison de la suite est une matrice. La suite géométrique du II montrait déjà une analogie avec ses sœurs dont la raison est numérique, et on retrouve un résultat similaire qui sera utilisé ci-dessous : Comme les valeurs propres de A sont les nombres r et s vus dans la partie II, A^n tend vers la matrice nulle lorsque n tend vers l'infini. Et donc U_n tend vers une limite qu'on calculera dans la partie IV, mais qu'on conjecturera très bientôt numériquement :

3. Traduction de l'algorithme en CoffeeScript

Avec les notations de l'énoncé et comme les opérations portent sur des vecteurs et matrices, on a rapidement ceci :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
for n in [1..3]
    U = (A.x U).add B
    affiche U.inspect()
```

Les réponses à la question 2 apparaissent donc dans l'affichage obtenu :

```
Algorithme lancé
[0.42, 0.33]
[0.4, 0.317]
[0.3868, 0.3117]

Algorithme exécuté en 52 millisecondes
```

En fait il a fallu affiner un peu l'affichage[10], à cause des erreurs d'arrondi de JavaScript :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
for n in [1..3]
    U = (A.x U).add B
    affiche U.map((x)->arrOdg(x,4)).inspect()
```

En bouclant sur plus de valeurs de n, on peut constater la convergence rapide de la suite U_n vers une limite qui sera déterminée algébriquement dans la partie IV.

4. Résultats obtenus

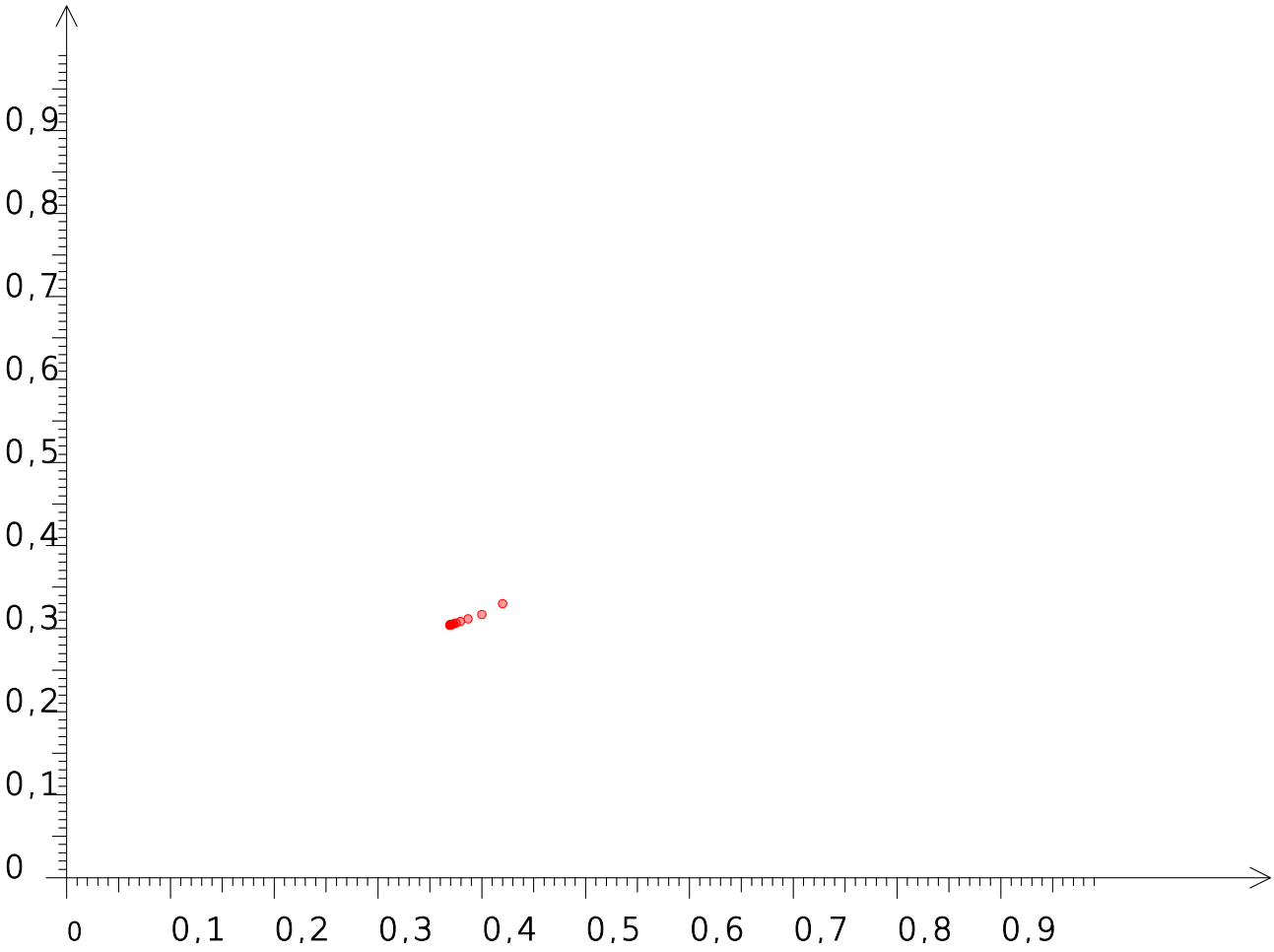
Puisque le script pour calculer U_n est très court, on peut l'agrémenter de calculs graphiques pour représenter graphiquement U_n comme nuage de points :

- L'instruction `effaceDessin()` a pour effet de vider la zone graphique d'***alcoffeethmique*** ;
- L'instruction `dessineAxes` crée deux axes de coordonnées sur le dessin, allant de 0 à 1 pour x, et aussi de 0 à 1 pour y. Il se trouve que l'échelle des x est 500 pixels et l'échelle des y est 400 pixels. De plus, l'origine (intersection des deux axes) a pour coordonnées (40 ; 440) en pixels, et que l'axe des y est orienté vers le bas. Les deux axes sont dessinés en noir (« black ») et sont gradués automatiquement.
- Ensuite on fait comme précédemment, en bouclant sur l'indice n, et en calculant U dans la boucle. Mais pour créer le point à tracer, on multiplie U par 400 (comme ça on a déjà la bonne échelle sur l'axe des y) puis on le transforme en tableau en cherchant ses `elements`.
- Enfin on dessine le point sous forme d'un cercle, l'abscisse du centre étant obtenue en multipliant celle du point par 1,25 pour avoir la bonne échelle sur les x, puis en additionnant 40 pour avoir la bonne origine du repère, et l'ordonnée du centre étant obtenue en soustrayant (car l'axe des ordonnées est orienté vers le bas) à 440 (car c'est l'ordonnée de l'origine du repère) celle du point. Le cercle a pour rayon 2 pixels (compromis entre la précision et la visibilité) et est en rouge pour qu'on aie des chances d'y voir quelque chose...

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
effaceDessin()
dessineAxes 0, 1, 0, 1, "black"
for n in [1..20]
    U = (A.x U).add B
    pt = (U.x 400).elements
    dessineCercle 40+pt[0]*1.25, 440-pt[1], 2, "red"
```

10 On a appliqué (« map ») au vecteur, c'est-à-dire à ses coordonnées, la fonction qui, à x, associe son arrondi à 4 décimales, cette fonction se notant en CoffeeScript `(x) → arrOdg x, 4`

L'empilement des points rouges vers la gauche montre graphiquement la vitesse de convergence :



IV/ Question 3

1. Deux nouvelles matrices

La matrice notée I tire son nom du fait que c'est l'identité. Il se trouve que dans sylvester la même lettre est utilisée pour noter la matrice identité, mais il faut lui fournir un entier (2) pour rappeler en quelle dimension on regarde l'identité :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
```

En fait, `I` n'est pas une matrice mais une fonction qui a pour effet de créer une matrice[11]. Cette fonction n'est pas disponible dans ***CoffeeScript*** de base, mais elle appartient à la classe `Matrix`. Elle ne s'appelle donc pas `I` mais `Matrix.I` pour rappeler de quelle fonction on parle.

11 Une matrice de matrices en quelque sorte...

La matrice N est définie par soustraction, on la calcule donc avec ***sylvester*** en utilisant sa fonction de soustraction de matrices :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
N = I.subtract A
affiche N.inspect()
```

La matrice N est donc celle-ci :

```
Algorithme lancé
[0.6, -0.4]
[-0.2, 0.9]

Algorithme exécuté en 71
millisecondes
```

L'état C d'équilibre est solution de NC=B. On l'obtient donc directement en multipliant l'inverse de N par B :

2. Inversion de N

Le point fort de ***sylvester*** est la rapidité à laquelle il inverse les matrices. Cependant, pour une matrice d'ordre 2, le calcul est rapide quel que soit l'algorithme utilisé[12]. La syntaxe est logique :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
N = I.subtract A
affiche N.inverse().inspect()
```

La problème est que l'inverse est donné sous forme d'approximation décimale et non de fractions :

```
Algorithme lancé
[1.956521739130435, 0.8695652173913043]
[0.4347826086956523, 1.3043478260869565]

Algorithme exécuté en 103 millisecondes
```

12 Celui utilisé par sylvester est l'algorithme du pivot de **Gauß**

Mais comme l'énoncé indique que le dénominateur est 23, on peut toujours multiplier l'inverse par 23 pour voir si les coefficients sont bien entiers (à la précision de la machine près) :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
N = I.subtract A
affiche
N.inverse().x(23).inspect()
```

On peut vérifier directement (puisque l'inverse est donné dans l'énoncé) que les produits de N par son inverse (dans les deux sens) donnent la matrice identité :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
N = I.subtract A
N1 = $M [[45,20],[10,30]]
N1 = N1.x (1/23)
affiche (N.x N1).snapTo(1).inspect()
affiche (N1.x N).snapTo(1).inspect()
```

3. Résolution du système

Il ne reste plus, maintenant qu'on a l'inverse de N, qu'à le multiplier par B pour avoir C :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
N = I.subtract A
C = N.inverse().multiply B
affiche C.inspect()
```

Une fois de plus, la distribution d'équilibre est donnée avec une grande précision mais sous forme approchée :

```
Algorithme lancé
[0.3695652173913044, 0.30434782608695654]

Algorithme exécuté en 29 millisecondes
```

Comme précédemment, on peut multiplier C par le dénominateur commun 46 pour vérifier que le résultat est formé d'entiers :

```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
N = I.subtract A
C = N.inverse().multiply B
affiche C.x(46).inspect()
```

On peut aussi vérifier que C est solution des équations données au début de la question 3 :

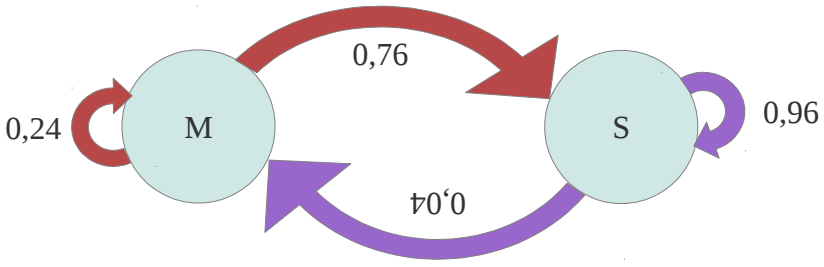
```
A = $M [[0.4,0.4],[0.2,0.1]]
B = $V [0.1,0.2]
U = $V [0.5,0.3]
I = Matrix.I 2
N = I.subtract A
C = $V [17/46,7/23]
affiche (N.x C).inspect()
affiche ((A.x C).add B).inspect()
affiche C.inspect()
```

Le fait que les deux derniers affichages renvoient le même résultat montre que C est bien la distribution d'équilibre.

V/ Partie 4

1. Remarque sur les suites de matrices

Si on regarde l'exercice 4 du sujet de Pondichery 2013, on y trouvait déjà une chaîne de Markov (mais à 2 états)[13], dont voici le graphe (M veut dire « malade », donc « le salarié est malade la semaine n » sera noté M_n et on utilise la lettre S comme « santé » pour désigner le contraire, afin d'éviter l'usage de la barre horizontale $S=\bar{M}$) :



Alors puisque $m_{n+1}=0{,}24\,m_n+0{,}04\,s_n$, $s_{n+1}=0{,}76\,m_n+0{,}96\,s_n$ et $m_n+s_n=1$ (en notant $m_n=P(M_n)=P(E_n)$ avec les notations de l'énoncé, et de même $s_n=P(S_n)=P(\bar{E}_n)$), en

13 Sujet récurrent d'ailleurs puisqu'il est réapparu (mais avec un terme constant) en 2014 en Amérique du Nord.

remplaçant s_n par $1-m_n$, on obtient $s_{n+1}=0{,}2\,s_n+0{,}04$ et s_n (p_n dans l'énoncé) est une suite arithmético-géométrique. On voit les mêmes phénomènes à l'œuvre qu'ici :

- baisse d'une dimension en utilisant le fait que la somme des coordonnées vaut 1
- passage d'une suite géométrique en grande dimension à une suite arithmético-géométrique en dimension plus petite
- perte de la valeur propre 1 qui transforme une suite ayant une limite matricielle en la somme d'une constante (la limite) et d'une suite géométrique tendant vers 0.
- La limite se trouve en résolvant une équation (ou un système) linéaire.

Seulement, en 2013, la suite arithmético-géométrique était numérique, alors qu'en 2014 la raison est une matrice d'ordre 2. Le principe reste le même. En particulier, pour déterminer la forme explicite d'une suite arithmético-géométrique, on la met sous forme de la somme de sa limite et d'une suite géométrique, et c'est ce qu'on fait ici, à ceci près que la limite est un vecteur de dimension 2 (C dans l'énoncé), et la raison de la suite, une matrice (A dans l'énoncé) dont les puissances tendent vers la matrice nulle. On démontre par récurrence que U_n-C est géométrique de raison A.

2. Théorie spectrale et calcul formel

Un petit tour par Xcas pour avoir les valeurs propres et vecteurs propres de A (notamment pour vérifier que les puissances de A tendent vers la matrice nulle). Pour commencer, on définit A dans Xcas presque comme dans ***sylvester*** :

```
A := [[4/10,4/10],[2/10,1/10]]
```

Les valeurs propres sont presque les mêmes que dans la partie I :

```
D := diag(eigenvals(A))
```

La matrice de passage est formée par les vecteurs propres de A :

```
P := eigenvectors(A)
```

En notant P1 l'inverse de P, on calcule sa valeur exacte par

```
P1 := simplify(P^(-1))
```

L'expression de P et surtout de P1, est compliquée, mais en demandant à Xcas de calculer le produit PDP1, on trouve bien A. Alors on montre par récurrence que

$A^n=P\,D^n\,P^{-1}=P\,D^n\,P1=P\begin{pmatrix} r^n & 0 \\ 0 & s^n \end{pmatrix}P1$ qui permet d'avoir la forme explicite de A^n et, partant de là, de x_n et y_n .

Sous Xcas, on définit la puissance n de A comme matrice notée Dn :

```
r := D[0][0]
s := D[1][1]
Dn := diag([r^n,s^n])
```

En notant x0 et y0 les valeurs initiales de x et y, on obtient l'expression explicite de x_n et de y_n en entrant

```
P*Dn*P1*[x0,y0]
```

Alors on apprend que x_n vaut

```
y0*(1/1640*(-5*sqrt(41)+15)*(3*sqrt(41)+41)*((-(sqrt(41))
+5)/20)^n+1/1640*(-
3*sqrt(41)+41)*(5*sqrt(41)+15)*((sqrt(41)+5)/20)^n)
+x0*((sqrt(41)*(5*sqrt(41)+15)*((sqrt(41)+5)/20)^n)/410-
((sqrt(41)*(-5*sqrt(41)+15)*((-(sqrt(41))+5)/20)^n)/410))
```

Alors que y_n vaut

```
y0*(1/82*(3*sqrt(41)+41)*((-(sqrt(41))+5)/20)^n+1/82*(-
3*sqrt(41)+41)*((sqrt(41)+5)/20)^n)+x0*(-((20*sqrt(41)*((-
(sqrt(41))+5)/20)^n)/410)+(20*sqrt(41)*((sqrt(41)+5)/20)^n)/410)
```

Ce qui permet effectivement de calculer les probabilités directement en fonction de n, puisque $z_n=1-x_n-y_n$ [14]. Du point de vue algorithmique, on n'est pas sûr d'y gagner puisque les puissances se calculent normalement par des boucles...

3. Réponse à la question du sujet

Au mois de mai, n est égal à 4, donc la réponse attendue est $A^4(U_0-C)+C$. Le calcul de la puissance quatrième se fait comme précédemment :

---

14 Attention : Ici x0 et y0 ne sont pas les valeurs initiales de x et y, mais les coordonnées de U_0-C . Il faudrait donc d'abord soustraire les coordonnées de C, puis remplacer x0 et y0 par ce qu'on obtient, enfin ajouter aux résultats les coordonnées de C à nouveau, comme le suggère la formule donnée dans la partie 4b. Les résultats obtenus, on s'en doute, ne sont guère plus simples que précédemment, puisqu'on obtient `(y0+(-7)/23)*(1/1640*(-5*sqrt(41)+15)*(3*sqrt(41)+41)*((-(sqrt(41))+5)/20)^n+1/1640*(-3*sqrt(41)+41)*(5*sqrt(41)+15)*((sqrt(41)+5)/20)^n)+(x0+(-17)/46)*((sqrt(41)*(5*sqrt(41)+15)*((sqrt(41)+5)/20)^n)/410-((sqrt(41)*(-5*sqrt(41)+15)*((-(sqrt(41))+5)/20)^n)/410))+17/46` pour x_n et `(y0+(-7)/23)*(1/82*(3*sqrt(41)+41)*((-(sqrt(41))+5)/20)^n+1/82*(-3*sqrt(41)+41)*((sqrt(41)+5)/20)^n)+(x0+(-17)/46)*(-((20*sqrt(41)*((-(sqrt(41))+5)/20)^n)/410)+(20*sqrt(41)*((sqrt(41)+5)/20)^n)/410)+7/23` pour y_n ...

```
A = $M [[0.4,0.4],[0.2,0.1]]
P = A
for n in [1..4]
    P = P.x A
affiche P.map((x)->arrOdg(x,6)).inspect()
```

On apprend alors que la puissance quatrième est

```
Algorithme lancé
[0.04424, 0.03764]
[0.01882, 0.01601]

Algorithme exécuté en 89
millisecondes
```

Pour calculer les coordonnées de $U_0 - C$ on peut utiliser les fractions d'***alcoffeethmique*** :

```
affiche differenceFractions (new Fraction 5, 10), (new Fraction 17, 46)
affiche differenceFractions (new Fraction 3, 10), (new Fraction 7, 23)
```

On connaît maintenant la valeur de $U_0 - C$:

```
Algorithme lancé
3/23
(-1/230)

Algorithme exécuté en 85
millisecondes
```

Alors $U_4 = \begin{pmatrix} 0{,}04424 & 0{,}03764 \\ 0{,}01882 & 0{,}01601 \end{pmatrix} \begin{pmatrix} \frac{3}{23} \\ \frac{-1}{230} \end{pmatrix} + \begin{pmatrix} \frac{17}{46} \\ \frac{7}{23} \end{pmatrix}$ qui fournit les valeurs exactes de x_4 et y_4 .

On conclut avec $z_4 = 1 - x_4 - y_4$ [15].

VI/ Simulation

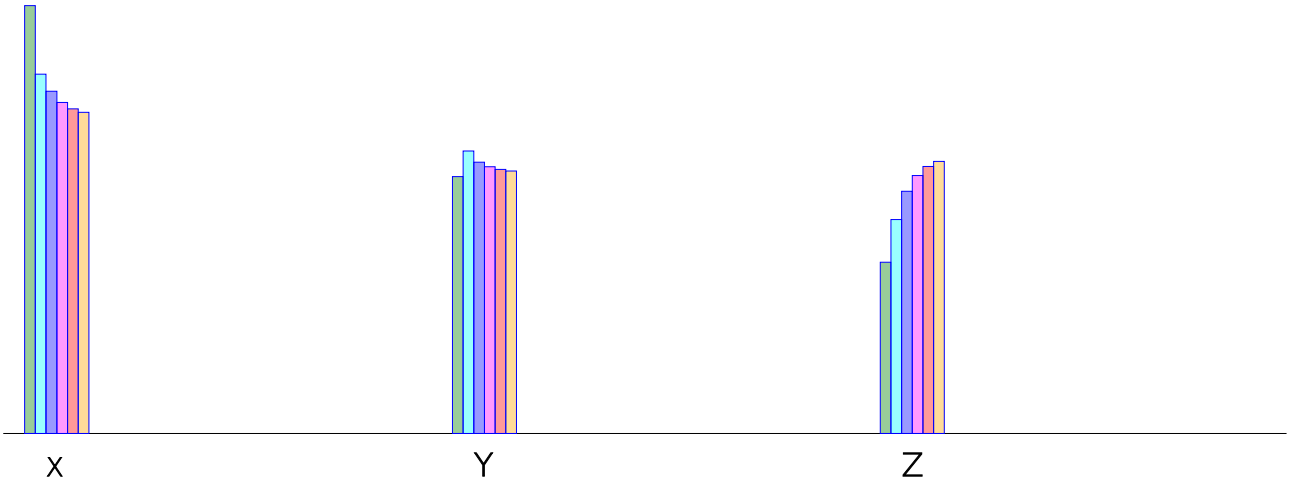
1. Convergence en loi

La loi d'équilibre est rapidement atteinte (visuellement, en moins de 10 itérations), ce qui peut s'illustrer par des diagrammes en bâtons, représentant les distributions selon n. Par exemple, pour la distribution au bout d'une itération :

15 En faisant tourner l'algorithme de la partie 2, on trouve (0,3794 ; 0.30853 ; 0.31207) qui sont déjà de bonnes approximations décimales de la distribution limite.

```
M = $M [[0.5,0.5,0.1],[0.4,0.3,0.2],[0.1,0.2,0.7]]
V = $V [0.5,0.3,0.2]
for n in [1..1]
    V = M.x V
freq = { 'X': V.elements[0], 'Y': V.elements[1], 'Z': V.elements[2] }
diagrammeBatons freq, 1
```

En superposant (de gauche à droite) les diagrammes en bâtons pour n allant de 0 à 5, on voit vers la droite (en rouge et orange) la stabilisation de la loi à long terme :



En fait il ne s'agit que de représenter graphiquement les suites x_n , y_n et z_n d'une autre manière que précédemment ...

2. Simulation d'une chaîne de Markov

Il y a deux choses à simuler pour faire des statistiques sur la chaîne de Markov : La valeur initiale et la valeur mise à jour à chaque étape n, selon la valeur précédente. Les deux simulations utilisent le même genre d'algorithme. Pour la valeur initiale, on choisit X, Y ou Z dans une urne qui contient 5 « X », 3 « Y » et 2 « Z » puisque la distribution initiale est (0,5 ; 0,3 ; 0,2) :

```
initial = new Sac ['X','X','X','X','X','Y','Y','Y','Z','Z']
affiche initial.extraireAuHasard()
```

Pour la suite, il suffit de constituer des urnes similaires, mais différentes selon que l'état actuel soit X, Y ou Z :

```
pas = []
pas.X = new Sac ['X','X','X','X','X','Y','Y','Y','Y','Z']
pas.Y = new Sac ['X','X','X','X','X','Y','Y','Y','Z','Z']
pas.Z = new Sac ['X','Y','Y','Z','Z','Z','Z','Z','Z','Z']
initial = new Sac ['X','X','X','X','X','Y','Y','Y','Z','Z']
trajet = initial.extraireAuHasard()
for n in [1..100]
    trajet += pas[trajet[trajet.length-1]].extraireAuHasard()
affiche trajet
```

On voit que les états changent peu souvent :

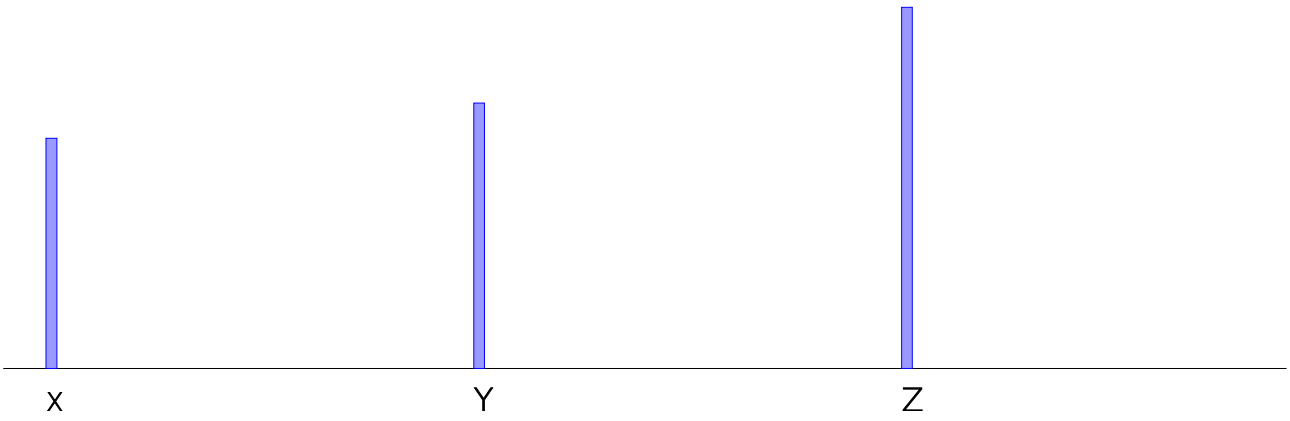
```
Algorithme lancé
XXYXXXXYYXXZZZZZZZZYXXZYXXYXXYYXYY
YXYXXXYXYYXYZYXYYYZZZYYYXZYXXXXYXY
XXYYZZZZYXXXXXYXXXZYYXXYZZYXXXXXX

Algorithme exécuté en 137
millisecondes
```

3. Diagramme en bâtons

L'algorithme précédent nécessite peu de modifications pour donner un diagramme en bâtons ; essentiellement remplacer la chaîne de caractères par un tableau :

```
pas = []
pas.X = new Sac ['X','X','X','X','X','Y','Y','Y','Y','Z']
pas.Y = new Sac ['X','X','X','X','X','Y','Y','Y','Z','Z']
pas.Z = new Sac ['X','Y','Y','Z','Z','Z','Z','Z','Z','Z']
initial = new Sac ['X','X','X','X','X','Y','Y','Y','Z','Z']
trajet = [initial.extraireAuHasard()]
for n in [1..1000]
    trajet.push pas[trajet[trajet.length-1]].extraireAuHasard()
tabEff = new Sac()
for x in trajet
    tabEff.ajoute x
diagrammeBatons tabEff.effectifs, 1000
```

Cependant, l'effet des premières valeurs se fait sentir même sur le long terme :



Alain Busser,
LPO Roland-Garros
juin 2014