Tom le robot prend un café

Dans ce document, on va voir comment le langage de programmation CoffeeScript permet de rédiger ou améliorer les algorithmes du sujet Antilles-Guyane 2013. On se concentrera sur l'outil joint à ce document, appelé ***alcoffeethmique***[1].

Partie A : Modélisation et simulation

La traduction directe de l'algorithme donné dans l'énoncé, avec les raccourcis permis par CoffeeScript, donne ceci :

```
n = 0
[x,y] = [0,0]
while -1<=y<=1 and x<=9
   n = dé(3) - 2
   y += n
   x++
affiche "La position de Tom est (#{x};#{y})"
```

La modification de la question 2 devient alors naturellement

```
n = 0
[x,y] = [0,0]
while -1<=y<=1 and x<=9
   n = dé(3) - 2
   y += n
   x++
if x is 10 and -1<=y<=1
   affiche "Tom a réussi la traversée"
else
   affiche "Tom est tombé"
```

On peut simplifier, dans l'idée d'itérer le procédé pour faire des statistiques :

```
[x,y] = [0,0]
[x,y] = [x+1, y + dé(3)-2] while -1<=y<=1 and x<=9
affiche (x is 10 and -1<=y<=1)
```

Alors le résultat est affiché sous forme booléenne (« true » ou « false » selon que Tom a réussi ou échoué). On le transforme en un tableau booléen sur lequel on pourra ensuite boucler, avec

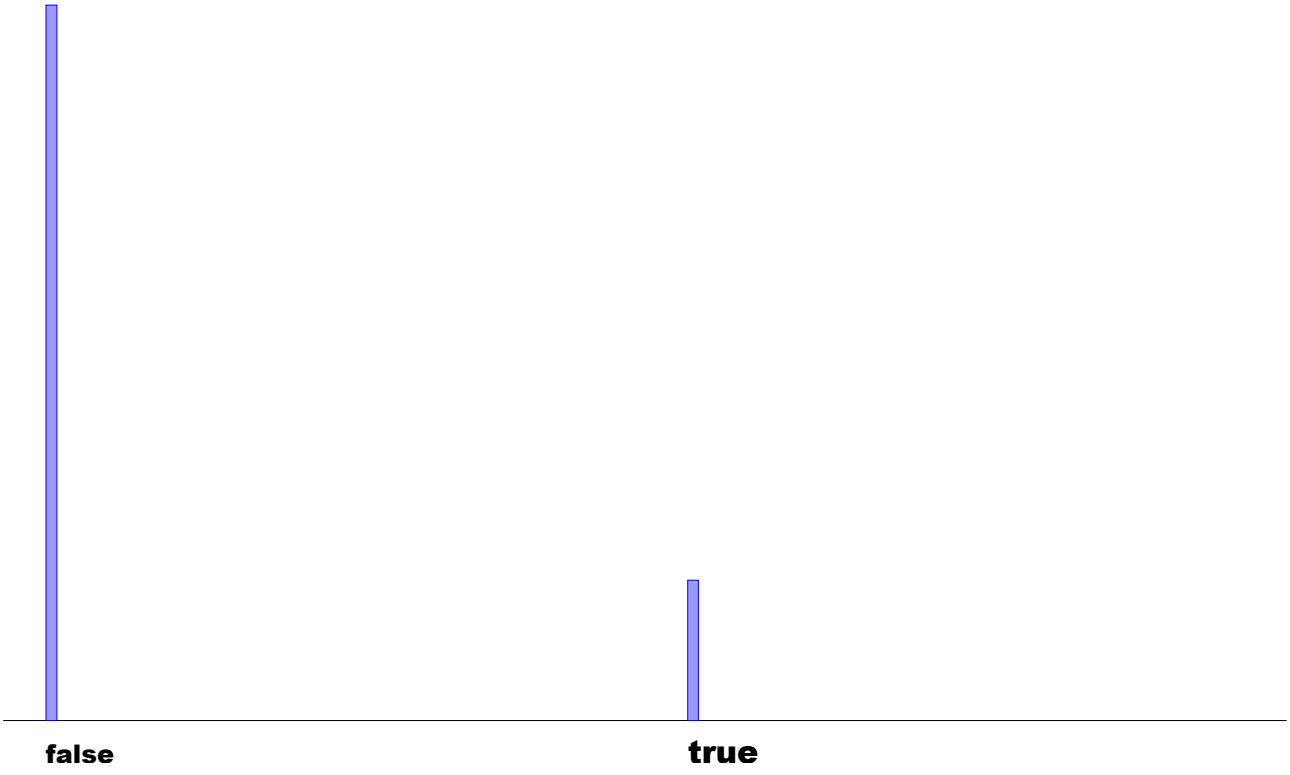
1 Après avoir dézippé le zip, on obtient un dossier, dans lequel se trouve un fichier ***alcoffeethmique.html*** ; un double-clic sur celui-ci permet de lancer l'outil

```
stats = []
[x,y] = [0,0]
[x,y] = [x+1, y + dé(3)-2] while -1<=y<=1 and x<=9
stats.push (x is 10 and -1<=y<=1)
affiche stats
```

Puis avec la boucle :

```
stats = new Sac []
for n in [1..1000]
    [x,y] = [0,0]
    [x,y] = [x+1, y + dé(3)-2] while -1<=y<=1 and x<=9
    stats.ajoute (x is 10 and -1<=y<=1)
diagrammeBatons stats.effectifs, 1000
```

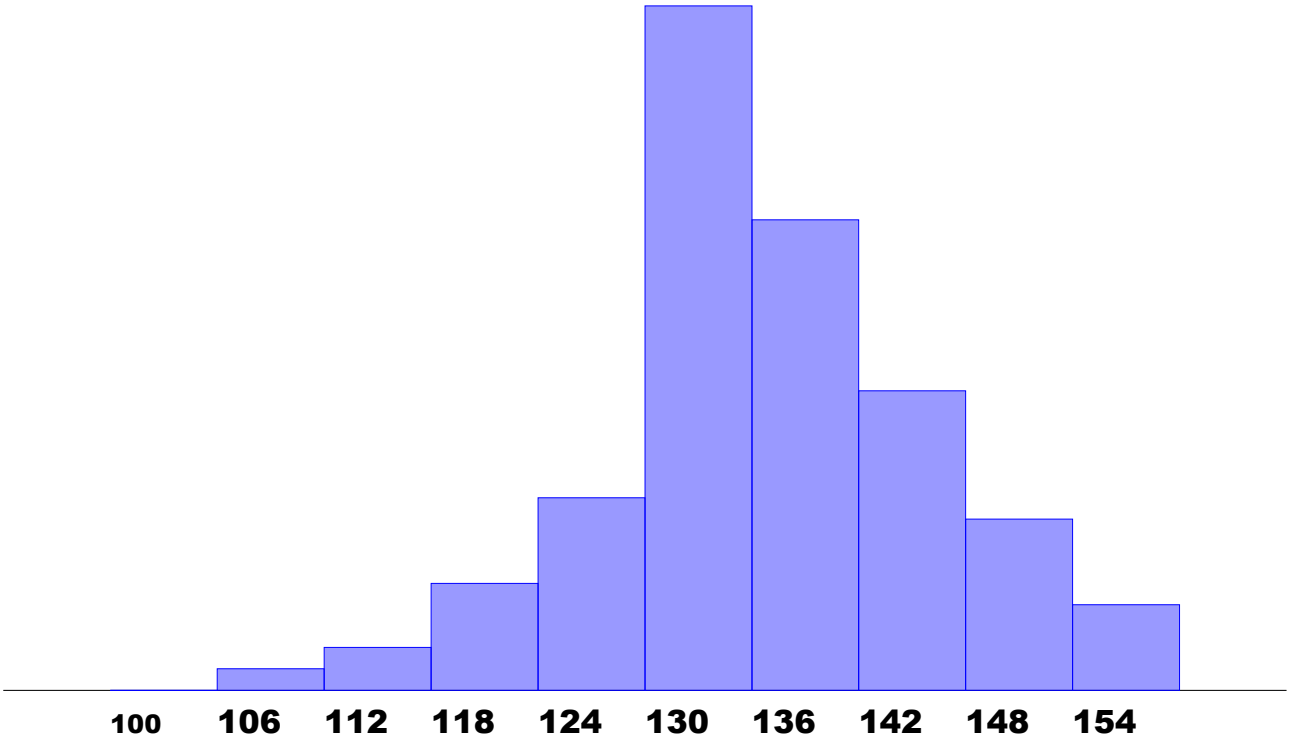
Le diagramme produit sur 1000 tentatives montre que le malheureux Tom boit la tasse plus souvent qu'à son tour :



L'étape suivante, c'est de mettre tout ça dans une boucle, où on recommence 100 fois l'expérience de 1000 tentatives, et où on stocke les nombres de succès dans un tableau :

```
succès = []
for exper in [1..100]
   stats = new Sac []
   for n in [1..1000]
      [x,y] = [0,0]
      [x,y] = [x+1, y + dé(3)-2] while -1<=y<=1 and x<=9
      stats.ajoute (x is 10 and -1<=y<=1)
   succès.push stats.effectifs["true"]

histogramme succès, 100, 160, 10, 40
```

En quelques secondes, on a cet histogramme :



En fait, le nombre de succès suit une loi binomiale de paramètres 1000 et $\frac{8119}{59049} \approx 0{,}1375$; on peut donc simuler les expériences ci-dessus en simulant cette loi binomiale, et avoir un intervalle de fluctuation à 95% en faisant

```
affiche IntFluctBinom 1000, 8119/59049
```

On trouve alors que 95% des expériences donnent entre 117 succès et 159 succès ; cet intervalle est centré sur 158 qui est une bonne estimation de la probabilité cherchée.

Partie B

Pour avoir le fichier tableur, on peut constituer le tableau avec CoffeeScript et enregistrer ledit tableau au format csv. Alors un simple double-clic[2] suffit à l'ouvrir dans un tableur. On constitue ce fichier avec ce script :

```
[a,b,c] = [0,1,0]
for n in [1..10]
    [a,b,c] = [(a+b)/3,(a+b+c)/3,(b+c)/3]
    affiche "#{n}, #{a}, #{b}, #{c}"
```

2 Ce qui revient exactement au même qu'un double simple-clic...

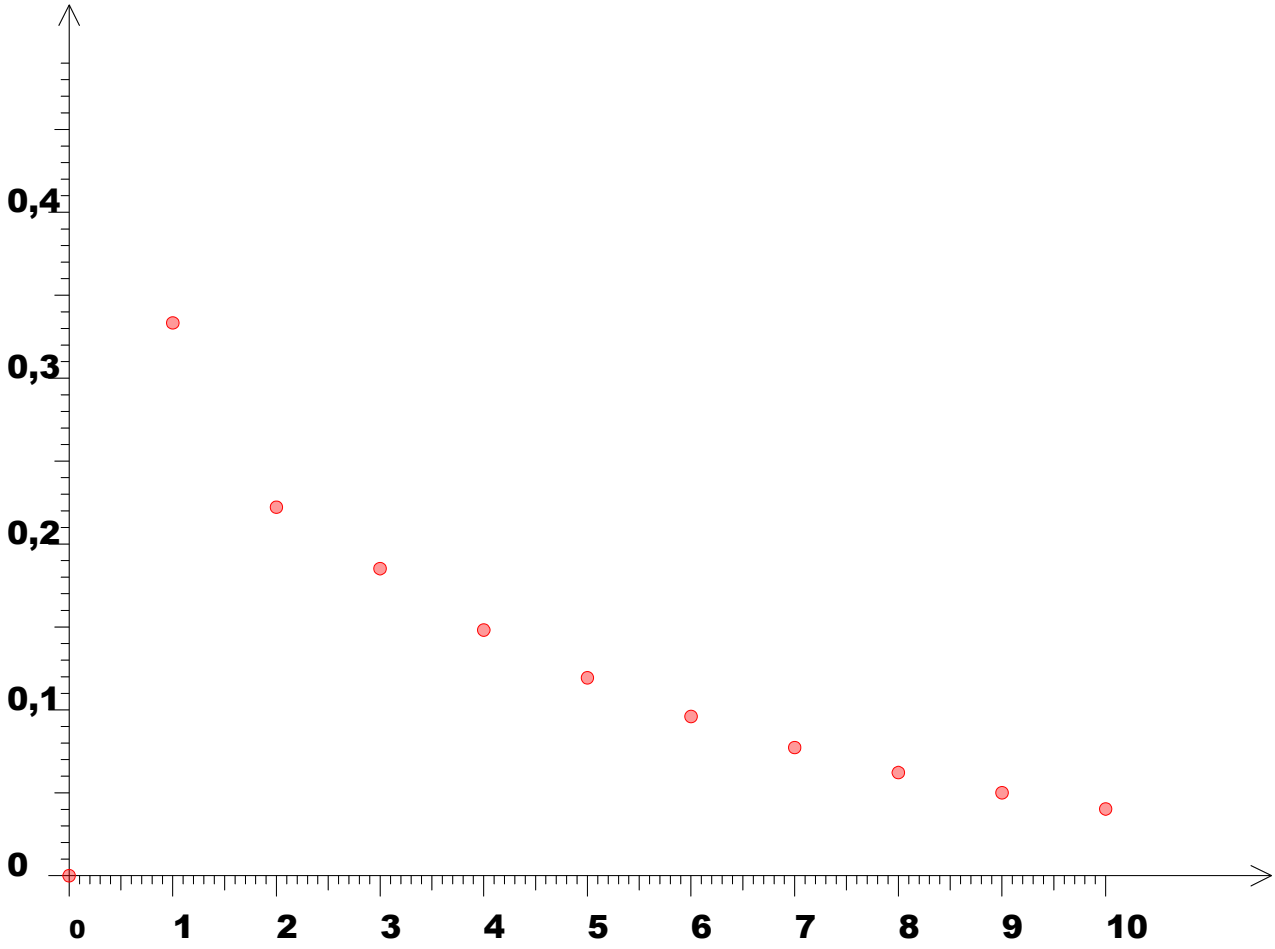
Voici l'affichage obtenu :

```
Algorithme lancé
1, 0.3333333333333333, 0.3333333333333333, 0.3333333333333333
2, 0.2222222222222222, 0.3333333333333333, 0.2222222222222222
3, 0.1851851851851852, 0.25925925925925924, 0.1851851851851852
4, 0.14814814814814814, 0.20987654320987656, 0.14814814814814814
5, 0.11934156378600824, 0.16872427983539096, 0.11934156378600824
6, 0.09602194787379974, 0.1358024691358025, 0.09602194787379974
7, 0.0772748056698674, 0.10928212162780065, 0.0772748056698674
8, 0.06218564243255601, 0.08794391098917848, 0.06218564243255601
9, 0.05004318447391149, 0.07077173195143016, 0.05004318447391149
10, 0.04027163880844722, 0.056952700299751045, 0.04027163880844722

Algorithme exécuté en 55 millisecondes
```

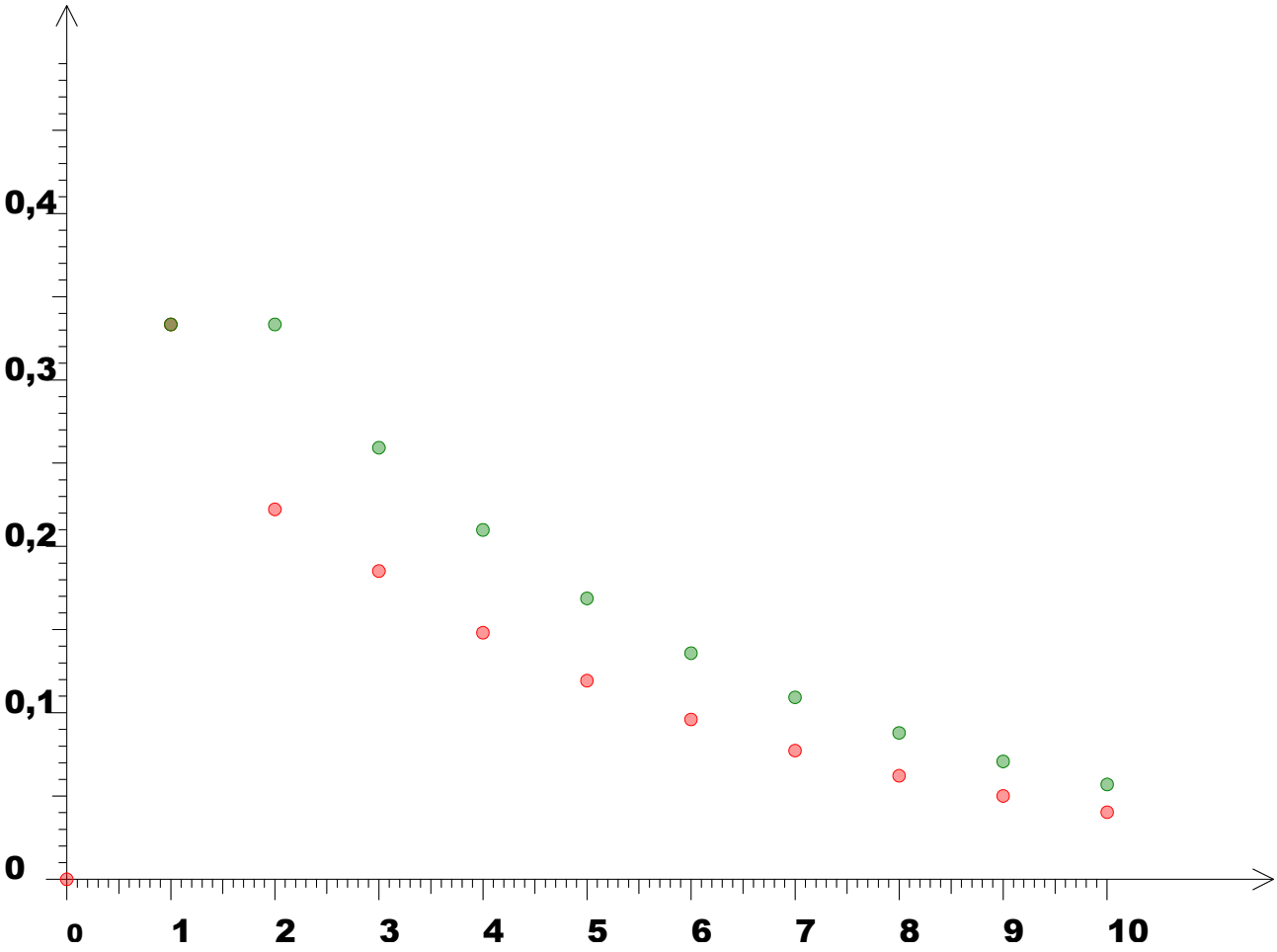
Pour représenter graphiquement la suite a_n , le script précédent subit une légère modification : Créer un objet supplémentaire appelé *liste*, et y pousser les valeurs de a au fur et à mesure qu'elles sont calculées :

```
[a,b,c] = [0,1,0]
liste = [a]

for n in [1..10]
    [a,b,c] = [(a+b)/3,(a+b+c)/3,(b+c)/3]
    liste.push a

dessineSuite liste, 10, 0, 0.5, 3, "red"
```

Ce qui produit la représentation graphique suivante :



En ajoutant la représentation graphique de la suite b_n en vert[3], on peut d'ailleurs comparer les suites :



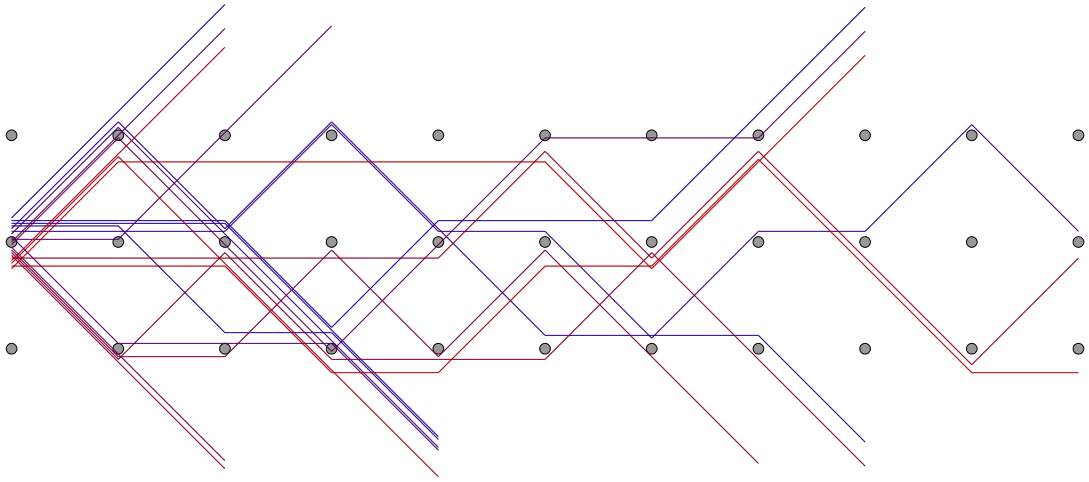
Partie C : Annexes au sujet de bac

On peut aussi tracer les trajectoires de Tom avec ce script :

3 La représentation graphique de c_n est inutile parce que $c_n = a_n$

```
effaceDessin()
for n in [0..10]
   dessineCercle 40+40*n, 40, 2, "black"
   dessineCercle 40+40*n, 80, 2, "black"
   dessineCercle 40+40*n, 120, 2, "black"
for exp in [1..20]
   [x1,y1] = [0,0]
   while -1<=y1<=1 and x1<=9
      y2 = y1 + dé(3)-2
      x2 = x1 + 1
      dessineSegment 40+40*x1, 70-40*y1+exp, 40+40*x2, 70-
40*y2+exp, "rgb(#{10*exp},0,#{10*(20-exp)})"
      [x1, y1] = [x2, y2]


$("#sortieSVG").text $("#graphique").html()
```

Le graphique ressemble à ceci (ici 3 trajets au sec sur 20) :



Pour finir en géométrie, rappelons qu'il est possible de programmer en CoffeeScript au sein de CaRMetal, et que c'est même plutôt rapide si la figure est préparée (munie de quelques objets géométriques) ; on peut la préparer ainsi :

- Créer 4 points de coordonnées respectives (-0,25;1,25), (9,25;1,25), (9,25;-1,25), (-0,25;-1,25)
- Les joindre par un rectangle appelé « pont »

- Créer un point appelé « Tom »

Créer une expression appelée « plouf » contenant ***!inside(Tom;pont)***

Créer deux objets texte, l'un caché appelé « compilateur » contenant le compilateur, l'autre appelé « script » où l'on écrira le CoffeeScript

Le texte « compilateur » sera rempli par un copié-coller depuis le compilateur « minifié » appelé ***coffee-script-min.js***[4]. Quand au script lui-même, on a tout fait pour qu'il soit court :

- On remet Tom à l'origine en faisant Move « Tom », 0, 0 ;
- Tant qu'il n'a pas fait « plouf », on le bouge en ajoutant 1 à son abscisse et le nombre aléatoire à son ordonnée, toujours avec un « Move » ; et on marque une petite pause avec Pause 800 ;
- Si on est arrivé ici, c'est donc que Tom n'est plus sur le pont ; alors on effectue un test sur son abscisse et on affiche le résultat comme précédemment.

Le script donne ceci :

```
Move "Tom", 0, 0
until GetExpressionValue("plouf")
   Move "Tom", X("Tom")+1, Y("Tom")+Math.floor(Math.random()*3)-1
   Pause 800
if X("Tom") is 10 and -1 <= Y("Tom") <= 1
   Println "Tom a réussi la traversée"
else
   Println "Tom est tombé"
```

Pour lancer l'animation, il faut du JavaScript, qui va « évaluer » le contenu (en JavaScript) du texte « compilateur », ce qui va alors créer un nouvel objet dans la figure, lequel s'appelle « CoffeeScript ». Il suffit ensuite d'invoquer sa méthode « run » en passant comme argument, le contenu du « script » :

```
eval(GetText("compilateur"));
CoffeeScript.run(GetText("script"));
```

Il suffit ensuite de lancer ce CaRScript pour martyriser ce pauvre Tom à volonté...

Après tout, les robots sont souvent faits de métal, Tom est carrément fait de … CaRMetal !!!

Alain Busser
Lycée Roland-Garros
Le Tampon

4 On le trouve par exemple sur ce site ; sinon on peut le récupérer dans le dossier zippé d'***alcoffeethmique***