UN CAFÉ STOCHASTIQUE DANS UNE VOITURE EN MÉTAL

Jeremy Askhenas, créateur du langage de programmation CoffeeScript, est connu dans le monde des programmeurs en Ruby[1] pour avoir porté en JavaScript les meilleures fonctionnalités de ce langage ; le résultat, appelé underscore.js, est toujours en cours de développement, et on peut l'utiliser dans CoffeeScript comme tout fichier JavaScript. Il sera donc fait un grand usage de cet outil trop peu connu dans cet article, et même il sera rajouté un utilitaire analogue (mais bien plus modeste) pour faciliter encore la tâche du programmeur qui veut calculer des probabilités :

I/ Simulation

1) Sans underscore.js

a) Lancer de dé

L'algorithme est très classique : Math.random() crée un nombre pseudo-aléatoire uniforme sur [0;1[, donc son produit par 6 est uniforme sur [0;6[et l'entier qui lui est immédiatement supérieur[2] est égal à 1, 2, 3, 4, 5 ou 6 avec équiprobabilité. Avec la calculatrice Ti82 stats fr, cela s'écrit *ent(6*NbrAléat)* [3] et en JavaScript, *Math.ceil(6*Math.random())* ; comme c'est long, autant créer une fonction *dé* en CoffeeScript :

```
dé = (nombreFaces) ->
    Math.ceil nombreFaces * Math.random()
```

Après, pour lancer un dé à 6 faces, il suffira d'écrire *dé 6* au besoin dans un test ou une boucle. Dans les figures CaRMetal ci-jointes, la fonction dé ci-dessus a été compilée, ce qui a produit une fonction dé en JavaScript:

```
var dé = function(nombreFaces){
    return(Mat.ceil(nombreFaces*Math.random())) ;
}
```

Cette fonction *dé* a été, avec d'autres fonctions JavaScript créées de façon analogue[4], ajoutée dans les figures CaRMetal[5], à l'instar de coffeescript-js et underscore.js.

Bien entendu, le paramètre « nombre de faces » a été laissé pour permettre à l'utilisateur de lancer

---

1 Populaire notamment par ses instructions *select*, *reject* ou *length* qui facilitent le comptage qui est fondamental en probabilités comme on le verra dans cet article.
2 En anglais, « ceiling », ou « plafond ».
3 Avec la Ti 82 stats fr, cet algorithme est inutile, en effet il suffit de faire *EntAléat(1,6)* pour simuler un dé à 6 faces
4 Par exemple une fonction *racineDe* qui remplace avantageusement la fonction *Math.sqrt*, ou, mieux, une fonction *cosinus* en degrés...
5 Dans un objet de type « texte », caché pour ne pas encombrer exagérément la figure.

des dés ayant plus de six faces (ou moins). Par exemple, un dé à 20 faces comme celui ci-dessous se simule avec *dé 20*...

Cela élargit le champ des activités que l'on peut traiter avec CoffeeScript, que ce soit en lançant des dés avec autant de faces que 20 (qui montrent l'intérêt de parcourir dans une boucle les 20 possibilités, plutôt que les énumérer à la main), ou des dés ayant un nombre de faces incompatible avec la construction classique des polyèdres de Platon. On s'en sert pour simuler un tirage dans une urne :

b) Tirage dans une urne

Pour CoffeeScript, une urne est un tableau, noté entre crochets. Par exemple, au loto-quine, on tire un jeton portant un nombre entre 1 et 90, dans un sac que l'on peut modéliser par un tableau contenant les 90 nombres entiers. Comme cela prend un temps rédhibitoire d'écrire ces 90 nombres entiers séparés par des virgules, on utilise un intervalle d'entiers noté [1..90]. Pour tirer une boule (soit, un numéro) au hasard dans l'urne, on fait juste

```
urne = [1..90]
Alert urne[dé 90]
```

Si on veut tester ce script en ligne, on doit mettre « alert » avec un « a » minuscule, l'orthographe « Alert » avec une initiale en majuscule étant spécifique à CaRMetal.

2) Avec underscore.js

a) Lancer de dé

Parmi les nombreux utilitaires fournis avec underscore.js, se trouve un dé similaire au randint de Python. On fournit les bornes de l'intervalle en paramètres, et pour lancer un dé à 20 faces comme celui ci-dessus, on peut faire

```
Alert _.random 1, 20
```

b) Permutations aléatoires

L'un des algorithmes les plus impressionnants de la boîte à outils qu'est underscore.js est celui de

Fisher[6] et Yates pour permuter aléatoirement[7] les éléments d'un tableau : Battre un jeu de cartes, mélanger les boules de loto dans l'urne, etc. En anglais, « battre un jeu de cartes » se dit « shuffle the cards » donc pour mélanger par exemple les 90 jetons d'un jeu de loto-quine, on fait simplement

```
urne = [1..90]
urne = _.shuffle urne
```

On peut se servir de ceci pour simuler des jeux de cartes, dès lors qu'on sait comment construire le tableau *jeu* contenant les noms des cartes. Par exemple, le « jeu de treize » analysé par Euler en 1739 consistait à mélanger deux jeux de 13 cartes (par exemple, les carreaux et les piques d'un jeu de 52 cartes) et les abattre l'une après l'autre en vis-à-vis, l'un des deux joueurs ayant parié sur une « rencontre » (deux cartes identiques au même moment) et l'autre sur son contraire. Euler savait que l'usage était de parier à 12 contre 7 ce qui revient à estimer la probabilité d'une rencontre à 12/19, et il a calculé la valeur approchée de cette probabilité : (e-1)/e, où e désigne l'unique nombre dont le logarithme népérien vaut 1. On simule le jeu de treize en mélangeant les entiers de 1 à 13 comme ci-dessus[8], et en cherchant si l'un d'entre eux n'a pas bougé.

Aussi, on peut construire un jeu de poker (52 cartes) en juxtaposant des valeurs (nombres de 1 à 10, réunis avec les initiales V, D, et R pour, respectivement, le valet, la dame et le roi) et des couleurs (carreau, cœur, pique et trèfle). Avec underscore.js cela donne quelque chose comme ceci :

`valeurs = _.union [1..10],["V","D","R"]`	On crée la liste des 10 nombres, on la réunit avec la liste des trois figures : On a la liste des 13 valeurs de cartes
`couleurs = ["\u2666","\u2665","\u2660","\u2663"]`	On crée la liste des 4 couleurs, abrégées par l'unicode les représentant (2666 pour ♦, etc) : On a la liste des couleurs de cartes
`jeu = (v+c for v in valeurs for c in couleurs)`	On concatène les 13 valeurs possibles avec les 13 couleurs possibles ; on obtient alors un tableau de tableaux (une matrice 13*4)
`jeu = _.flatten jeu`	On aplatit cette matrice pour la transformer en un tableau de 52 cartes

Ensuite on peut, comme précédemment, tirer une carte au hasard, mélanger les cartes etc. On s'en servira plus bas pour aborder ludiquement la notion d'échantillonnage. Les cartes s'affichent ainsi :

3♥,3♠,D♠,D♣,9♦

6 Il s'agit de l'inventeur de la statistique inférentielle

7 Comme le groupe des permutations sur n éléments est engendré par les transpositions (échanges de deux éléments), on fabrique une permutation aléatoire en appliquant successivement des transpositions aléatoirement choisies, en nombre judicieusement choisi pour que les n éléments soient bien mélangés sans y passer trop de temps. C'est un peu ce qu'on fait lorsqu'on mélange les cartes au poker, en alternant celles-ci après avoir coupé le jeu.

8 Suivant en cela Euler qui a fait la remarque que rien ne changeait dans les données du problème si un seul des deux jeux était mélangé.

II/ Calcul de probabilités

Pour calculer la probabilité d'un événement, on doit compter deux fois[9] :

- Le numérateur de la probabilité est le nombre d'éventualités favorables à la réalisation de l'événement ;
- le dénominateur est le nombre d'éventualités possibles dans l'univers de probabilité entier.

L'algorithmique peut être d'une aide précieuse, le comptage pouvant être fait dans une boucle. Underscore.js simplifie considérablement la tâche, en permettant de compter mais également en calculant des événements, vus comme des ensembles.

1) Ensembles et événements

a) Ensembles

Dans CoffeeScript, un ensemble est représenté par un tableau ; mais alors qu'un même élément peut apparaître plusieurs fois dans un tableau, ce n'est pas le cas pour un ensemble[10]. Alors pour transformer un tableau en ensemble, on fait *_.unique tableau*. Mais cela n'est pas nécessaire si les ensembles sont construits avec une seule occurrence de chacun de leurs éléments.

On reprend ici l'exemple du dé icosaédrique avec des faces numérotées de 1 à 20. On s'intéresse à deux événements (sous-entendu, après le lancer du dé) :

- A : « le numéro obtenu est impair »
- B : « le numéro obtenu est divisible par 3 »

Cette description des événements, habituelle au collège, est dite « en compréhension ». Au lycée, il est utile de la compléter par une description « en extension » : A={1,3,5,7,9,11,13,15,17,19} et B={3,6,9,12,15,18}. Bien entendu, l'univers est ici Ω={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}...

Avec CoffeeScript, on peut construire les événements en extension ; pour cela, on utilise la notation « % » pour « modulo » (reste dans la division euclidienne), et on redéfinit

- un nombre *x* comme impair si *x%2* vaut 1 (sinon x serait divisible par 2 et donc pair)
- un nombre *x* comme divisible par 3 si *x%3* vaut 0

---

9 Comme Chuck Norris, mais lui, c'est jusqu'à l'infini qu'il a compté deux fois...

10 Dans les westerns, le cow-boy typique, appelons-le Jack, essaye de gagner au poker avec un jeu de cartes qui n'est pas un ensemble, puisque l'as de trèfle y apparaît en 2 exemplaires. Lorsque ses adversaires découvrent la supercherie, Jack se voit ~~proposer~~ imposer de changer de jeu de hasard :

- passage du poker à la roulette russe, consistant à tirer au hasard une balle de calibre 35, d'une urne à 6 emplacements
- passage dans une urne contenant une quantité aléatoire de goudron, avec tirage au sort d'un nombre indéterminé de plumes ; la sélection des plumes se faisant avec l'outil Jack+goudron
- jeu des petits chevaux, avec un seul cheval qui sort tout de suite de l'écurie, en galopant, mais avec Jack attaché dessus, tourné vers l'arrière...

Les événements A et B précédents, ainsi que l'univers, peuvent être construits « en compréhension » à partir de leur description :

```
Omega=[1..20]
A=(x for x in Omega when x%2 is 1)
B=(x for x in Omega when x%3 is 0)
```

b) Intersection

L'événement « le résultat est à la fois impair et multiple de 3 » s'écrit en extension {3,9,15} et en compréhension *_.intersection A, B* avec underscore.js. Il s'agit évidemment de $A \cap B$.

c) Réunion

L'événement « le résultat est, ou bien impair, ou bien multiple de 3 » s'écrit

- en extension {1,3,5,7,9,11,13,15,17,19,6,12,18}
- ou en compréhension *_.union A, B* avec underscore. Il s'agit de $A \cup B$.

d) Contraire

Le contraire de A peut s'obtenir en compréhension avec *_.reject Omega, (x)-> _.contains A, x* (avec underscore.js) ou, plus simplement, *_.difference Omega, A* (le contraire de A s'obtient en soustrayant A à l'univers)

2) Probabilités

Le nombre d'éventualités que contient un événement s'appelle son cardinal ; mais pour CoffeeScript, c'est sa longueur (length). Pour calculer la probabilité d'un événement E, on effectue alors juste le quotient *E.length/Omega.length*. Le tout est de compter les éventualités de E et de l'univers, ce qui, avec un tableau ou un arbre, peut être long.

À titre d'exemple, on va comparer différentes manières de calculer la probabilité d'avoir au moins un 6 en lançant 4 dés[11] ; et pour aborder ce problème, on va commencer par une version simplifiée :

On lance simultanément[12] deux dés « normaux »[13] ; quelle est la probabilité d'avoir au moins un 6 ?

Première méthode : Avec un arbre

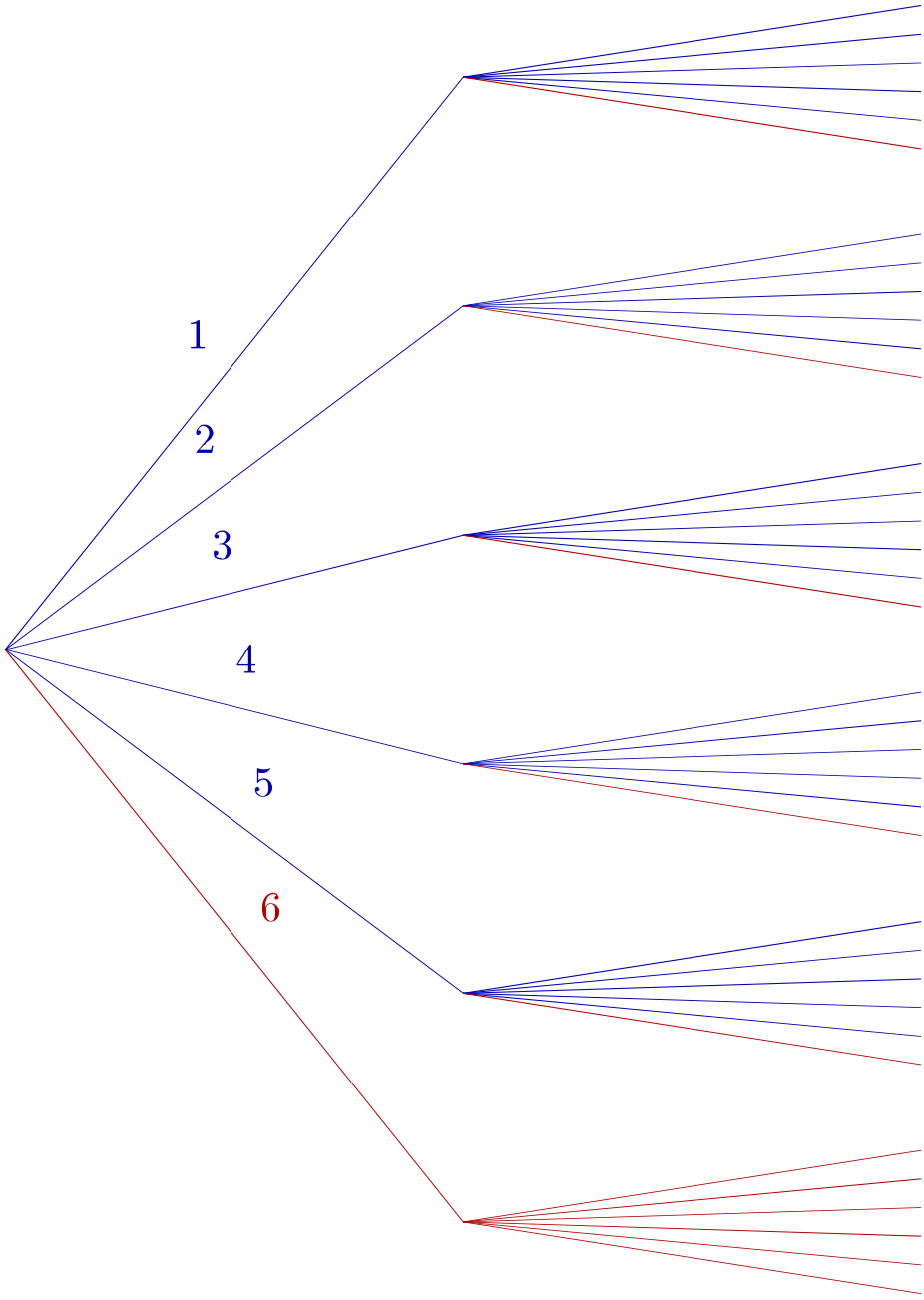
Si on essaye de calculer cette probabilité avec un arbre, l'arbre est long à dessiner :

---

11 problème posé par le chevalier de Méré à Blaise Pascal dans une lettre dont on ne connaît l'existence que parce que Blaise Pascal l'a citée dans une lettre à Pierre de Fermat

12 Et indépendamment, pour être en mesure de calculer par exemple la probabilité d'avoir deux 6, qui dans ce cas s'obtient par une multiplication.

13 Cubiques et non pipés



Les branches principales représentent le premier dé, et sont en rouge si celui-ci donne 6, puisque dans ce cas toute la suite de l'arbre est favorable à l'événement « il y a au moins un 6 ». Les branches secondaires de l'arbre représentent le second dé, et sont en rouge, soit lorsque celui-ci a donné 6, soit lorsque le premier dé a donné 6 (dernière branche principale en bas). Pour calculer la probabilité voulue, on compte les feuilles rouges (il y en a 5+6=11) et le nombre total de feuilles (36) puis leur quotient : On a 11 chances sur 36 d'avoir un 6 en lançant deux dés.

Pour le problème à 4 dés du chevalier, on a un problème : L'arbre[14] comprend 1296 feuilles, et devient très difficile à construire, sans parler de sa coloration...

14 D'où l'intérêt de la méthode algorithmique présentée ci-dessous : Il a de toute façon fallu écrire un programme (en JavaScript ci-dessus) pour construire l'arbre, alors autant modifier le programme pour compter les éventualités...

Seconde méthode : Avec un tableau

On peut alors essayer un tableau, avec un tableur ; on peut se simplifier la vie en remarquant que l'événement « au moins un dé donne 6 » est équivalent à l'événement « le maximum des deux dés est 6 ».

dés	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	2	3	4	5	6
3	3	3	3	4	5	6
4	4	4	4	4	5	6
5	5	5	5	5	5	6
6	6	6	6	6	6	6

La formule « *=MAX(B$1;$A2)* », entrée dans la cellule B2 puis copiée dans le reste du tableau, permet de remplir automatiquement celui-ci. Ensuite, on compte le nombre total de cellules (36) et le nombre de cellules comportant un 6 (11 : On ne compte pas les bords du tableau), et on a là encore la probabilité voulue : 11 chances sur 36.

Par rapport à la méthode précédente, celle-ci est plus visuelle (on voit non seulement mieux qu'il y a 11 fois un 6, on voit même pourquoi : 36-25, en soustrayant les aires de deux carrés). Mais outre le fait que, comme la méthode précédente, celle-ci s'étend difficilement au cas de 4 dés (problème historique du chevalier de Méré) à cause du nombre élevé de cas favorables (et possibles), mais en plus, un tableau à 4 dimensions est nettement moins visuel qu'un tableau à deux dimensions...

Troisième méthode : Algorithmique

On peut parcourir les 36 résultats possibles des deux dés, avec une boucle sur 6 valeurs, elle-même imbriquée dans une boucle sur 6 valeurs, et dans cette boucle, incrémenter un compteur chaque fois qu'il y a au moins un 6. En CoffeeScript cela donne quelque chose comme

```
compteur = 0
for dé1 in [1..6]
    for dé2 in [1..6]
        if dé1 is 6 or dé2 is 6
            compteur++
Alert compteur/36
```

Mais underscore.js peut compter lui-même les cas favorables :

```
tableau = (_.max [a,b] for a in [1..6] for b in [1..6])
tableau = _.flatten tableau
effectifs = _.countBy tableau, (x)-> x
Alert effectifs[6]/36
```

On peut également créer un événement « ensemble des cas favorables » avec *_.groupBy*, puis compter les 6...

Contrairement à l'arbre et au tableau, il est assez facile de modifier l'algorithme ci-dessus pour l'adapter au cas de 4 dés (problème du chevalier de Méré). On serait donc tenté de crier « vive l'algorithmique », mais Blaise Pascal a trouvé une méthode encore plus rapide pour calculer cette probabilité :

Quatrième méthode : Avec une suite géométrique

Le raisonnement de Pascal est le suivant : En lançant un dé, on a une chance sur 6 d'avoir un 6, donc 5 chances sur 6 de ne ***pas*** avoir de 6. En supposant les deux dés indépendants entre eux, en lançant deux dés, on a donc 25 chances sur 36 (produit des deux probabilités) de ne pas avoir de 6. La probabilité cherchée est celle de son contraire, soit 11 chances sur 36.

Pour généraliser cela au problème du chevalier de Méré, on cherche le quatrième terme d'une suite géométrique de raison 5/6. Ce qui avec CoffeeScript peut s'écrire

```
termeGénéral = 1
(termeGénéral *= 5/6 for indice in [1..4])
Alert 1-termeGénéral
```

III/ Échantillonnage

1) Sans remise

Pour aider Jack le cow-boy à s'entraîner au poker, on peut, une fois qu'on a construit avec CoffeeScript un jeu de 52 cartes comme ci-dessus, simuler une main de poker, qui s'obtient en tirant 5 cartes au hasard dans le jeu. Mais ce tirage doit être fait sans remise, sinon Jack court le risque d'avoir deux fois la même carte, et ne le souhaite pas, ayant passé des heures à enlever le goudron et les plumes...

CoffeeScript ne possède pas d'algorithme de tirage sans remise, ni underscore. Mais un algorithme peut aisément être mis en œuvre avec l'outil *_.shuffle*, qui se charge de mélanger au hasard les cartes. L'algorithme peut se décrire ainsi :

- on mélange les 52 cartes (en leur applicant une permutation aléatoire avec _.shuffle jeu)
- on donne à Jack les 5 premières cartes du jeu (qui s'appellent jeu[0], jeu[1], jeu[2], jeu[3] et jeu[4])

Avec CoffeeScript cela donne

```
valeurs = _.union [1..10],["V","D","R"]
couleurs = ["\u2666","\u2665","\u2660","\u2663"]
jeu = (v+c for v in valeurs for c in couleurs)
jeu = _.shuffle _.flatten jeu
main = (jeu[n] for n in _.range 5)
Alert main
```

C'est ce script par exemple, qui a produit l'exemple de la page 3[15].

Pour le loto (49 boules numérotées de 1 à 49), on effectue un tirage de 5 nombres dans un tableau de 49 nombres. Cela donne

```
urne = [1..49]
urne = _.shuffle urne
alert urne[0..4]
```

Pour simuler un sondage d'opinion, on effectue un tirage sans remise dans une population de « pour » et de « contre », le nombre d'éléments tirés au sort étant la taille de l'échantillon. Par exemple, un échantillon de taille 100 prélevé dans le village d'Hénon-Beaumin (10 000 électeurs dont 3 000 sont « pour » le maire sortant), se fait avec

```
population = []
_.times 3000, ()->population.push "pour"
_.times 7000, ()->population.push "contre"
population = _.shuffle population
échantillon = _.countBy population[1..100], (x)->x
Alert échantillon["pour"]
```

Les deux dernières lignes transforment l'échantillon en un tableau d'effectifs, avec affichage de l'effectif des « pour », c'est-à-dire du nombre de succès dans l'échantillon. Ce nombre suit une loi

15 Dans le fichier CaRMetal (cartes.zirs), il a été créé une classe « Carte », et la programmation objet a permis de rajouter des méthodes comme « estUnPique », ou « estRouge » à partir desquelles on peut exprimer presque en Français, certains événements en compréhension.

hypergéométrique de paramètres 100, 0,3 et 10000, qui est assez bien approchée par une loi binomiale de paramètres 100 et 0,3, laquelle est à son tour approchée par une loi normale de paramètres 30 et 4,58...

2) Avec remise

La loi hypergéométrique n'étant pas au programme de lycée, on cherche également à simuler un tirage avec remise. Ça tombe bien, c'est plus facile, avec une boucle dans laquelle on répète un tirage d'un élément (carte, boule etc) au hasard. En plus, lorsqu'on lance plusieurs dés en même temps, on effectue en fait un tirage avec remise.

La simulation de sondage d'opinion avec remise donne ceci :

```
population = []
_.times 3000, ()->population.push "pour"
_.times 7000, ()->population.push "contre"
échantillon = (population[dé 10000] for n in [1..100])
effectifs = _.countBy échantillon, (x)->x
Alert effectifs["pour"]
```

Remarque sur la Ti82 stats fr : L'instruction *NbrAléat(100)* créée une liste de 100 nombres (pseudo)aléatoires uniformes sur [0;1[. On peut s'en servir pour simuler des lancers de dés, on des sondages. Par exemple, dans le village d'Hénon-Beaumin, le nombre de 1[16] dans la liste *ent(NbrAléat(100)+0.3)* est le nombre de succès dans une épreuve de Bernoulli de paramètres 100 et 0,3, donc le nombre d'électeurs restés fidèles au maire parmi les 100 personnes sondées.

3) Problème des anniversaires

Lorsqu'on choisit 30 nombres au hasard (avec remise) dans une liste de 365 nombres, on est étonné si deux d'entre eux sont égaux. C'est le problème des anniversaires : Lorsque deux élèves d'une même classe ont leur anniversaire le même jour, ils s'en étonnent. En fait, il s'agit d'une confusion entre tirages avec et sans remise, on s'attend naturellement à ce que le tirage des dates d'anniversaires dans l'année soit fait sans remise parce que « ça a l'air plus aléatoire »[17]. Pour calculer la probabilité que, parmi 30 dates choisies au hasard (et avec remise!) dans l'année, il y en ait deux identiques, on peut utiliser cet algorithme :

16 Comme la liste ne comporte que des 0 et des 1, le nombre de 1 est juste la somme des éléments de la liste : *somme(ent(NbrAléat(100)+0.3))*

17 En fait c'est le contraire, un tirage sans remise n'étant pas indépendant, est donc ***moins*** aléatoire qu'un tirage avec remise. Ce paradoxe est similaire à celui qui fait qu'on s'étonne de voir autant de séries de « pile » successifs en lançant une pièce plusieurs fois.

```
u = 1
(u *= (365-n)/365 for n in [1...30])
Alert 1-u
```

Cet algorithme annonce que la probabilité que deux élèves d'une classe de 30 aient leur anniversaire le même jour est 0,7 qui est suffisamment élevé pour ne pas s'en étonner.

4) Loi géométrique

Encore un problème inspiré par le chevalier de Méré : ***On lance une paire de dés jusqu'à ce qu'on ait un double 6. On veut simuler la variable aléatoire « nombre de lancers nécessaires »***. Comme la probabilité d'avoir un double 6 en lançant une fois la paire de dés est 1 chance sur 36, la variable aléatoire considérée est géométrique de paramètre 1/36. On la simule ainsi :

```
n = 0
(n++ until (dé 6 is 6) and (dé 6 is 6))
Alert n
```

Les instructions entre parenthèses se traduisent par « on incrémente n jusqu'à ce que le premier dé donne 6 et le second dé donne 6 ».

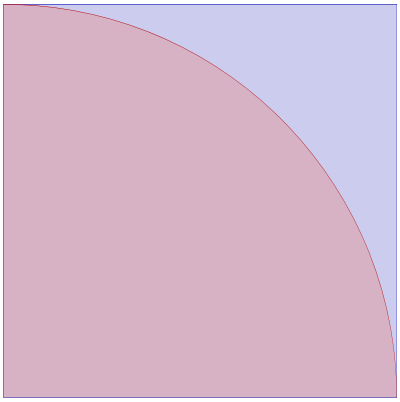
Pour en faire une variable géométrique tronquée, on doit rajouter dans le test (tout ce qui est après « until ») un *or n >= 20* qui a pour effet de tronquer n à 20.

IV/ CaRMetal

CaRMetal est un logiciel de géométrie. Programmer en CoffeeScript sous CaRMetal permet donc de parler de géométrie. Ou d'utiliser les possibilités graphiques de CaRMetal. Quelques exemples d'application sont brièvement exposés ci-dessous.

1) Méthode de Monte-Carlo

Pour estimer l'aire du quart de disque de rayon 1, on crée un nuage de points de coordonnées aléatoires (uniformes sur [0;1[) et on compte les points qui sont dans le disque de rayon 1, c'est-à-dire ceux dont la distance à 0 est inférieure à 1. Leur fréquence est un estimateur de l'aire du quart de disque. La définition de la probabilité mise en jeu ici est celle du quotient de deux aires, typiquement mise en œuvre au jeu de fléchettes : La probabilité que la fléchette soit à une distance inférieure à 1 de l'origine, est le quotient de l'aire du quart de disque (en rouge) par celle du carré (qui vaut 1), soit $\frac{\pi}{4}$:



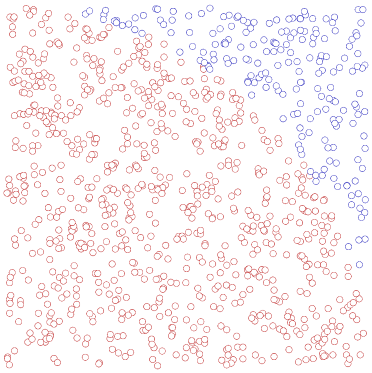
underscore permet de construire le nuage de points avant de le colorer :

- on construit le nuage avec nuage = *(Point Math.random(), Math.random() for n in [1..100])* ; on a alors un tableau de noms de points (il y en a 100, allant de P1 à P100) ;
- on sépare ce tableau en les points qui sont proches de l'origine (indexés par « true ») et les autres (indexés par « false ») ; cela construit un tableau « couleurs » de deux tableaux avec *couleurs = _.groupBy nuage, (p)->X(p)*X(p)+Y(p)*Y(p)<1*. En fait on définit une fonction qui à un (nom de) point p, associe « true » ou « false » selon sa distance à l'origine, et les deux tableaux sont les images réciproques de {true, false} par cette fonction.
- alors *couleurs[true]* est la liste des points à colorier en rouge et *couleurs[false]* est la liste des points à colorier en bleu. Le coloriage en bleu se fait avec *(SetColor p, "blue" for p in couleurs[false])* (on peut traduire cette ligne par « colorier en bleu tous les points qui sont dans couleurs[false] ; c'est une sorte d'apartheid inversé) et le coloriage en rouge se fait de façon analogue.

Le script suivant dessine et colorie le nuage de points :

```
nuage = (Point Math.random(), Math.random() for n in [1..100])
couleurs = _.groupBy nuage, (p)->X(p)*X(p)+Y(p)*Y(p)<1
(SetColor p, "blue" for p in couleurs[false])
(SetColor p, "red" for p in couleurs[true])
```

Le résultat ressemble à ceci (avec 1000 fléchettes plutôt que 100) :



Pour avoir une valeur approchée de π, il suffit de remplacer groupBy par countBy et d'utiliser length comme précédemment pour connaître le nombre de points rouges, puis diviser ce nombre par 100 et le multiplier par 4. Mais le dessin prend du temps, et on va plus vite en faisant seulement des calculs (ici, un million de points comptés en quelques secondes) :

```
rouges = 0
for indice in [1..1000000]
    [x,y]=[Math.random(),Math.random()]
    (rouges++ if x*x+y*y<1)
Alert rouges/1000000*4
```

2) Diagrammes en bâtons

Sur les 100 points du nuage précédent, le nombre de ceux qui sont coloriés en rouge suit donc une loi binomiale de paramètres 100 et $\frac{\pi}{4}$. On peut représenter cette loi par un diagramme en bâtons ; ces bâtons sont des segments joignant deux sommets, l'un de coordonnées (k;0) et l'autre de coordonnées (k;f) où f est la fréquence des expériences ayant donné k succès. Alors si ces points sont cachés, il suffit de faire un *Move « B#{n} », n, f* pour mettre à jour le diagramme en bâtons, et ceci sous forme d'une animation.

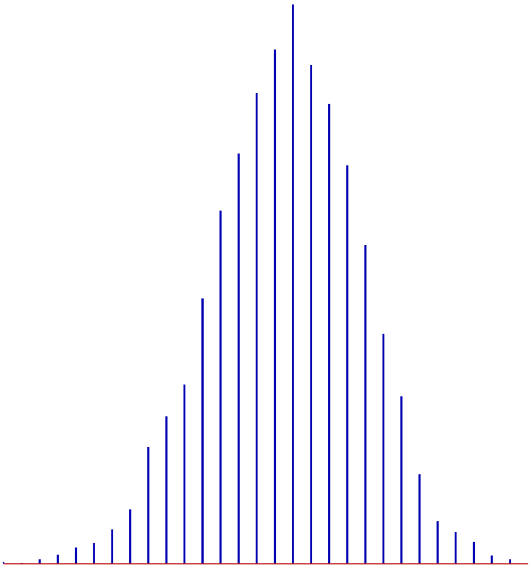
On commence par créer les bâtons par un script :

```
for n in [0..100]
    a = Point n, 0
    SetHide a, true
    b = Point "B#{n}", n, 0
    SetHide b, true
    s=Segment a, b
```

Puis on réalise 600 fois l'expérience de créer un nuage de 100 points, et compter les points rouges. Le diagramme en bâtons s'anime alors avec Move b, n, f et on voit progressivement apparaître une courbe en cloche qui esquisse l'intervalle de fluctuation :

```
for exper in [1..600]
    nrouges = 0
    for n in [1..100]
        [x,y]=[Math.random(), Math.random()]
        (nrouges++ if x*x+y*y<1)
    Move "B#{nrouges}", X("B#{nrouges}"), Y("B#{nrouges}")+1
```

Le diagramme sur 6000 expériences ressemble à ceci :



3) Paradoxe de Bertrand

Le problème étudié par Joseph Bertrand est le suivant : ***Quelle est la probabilité qu'une corde choisie au hasard dans le cercle unité, mesure moins que le côté du triangle équilatéral inscrit dans ce cercle ?***

Le script suivant est destiné à être lancé dans une figure CaRMetal qui comprend déjà

- le cercle unité (centré sur l'origine, et de rayon 1)
- le point A(1;0) qui sera la première extrémité de la corde aléatoire[18]
- une expression « E1 » qui permet au script de communiquer avec CaRMetal et de profiter de sa puissance de calcul.

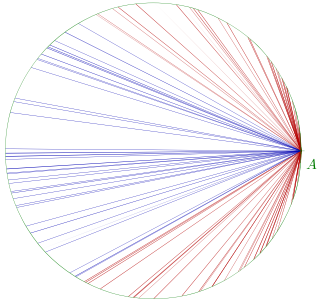
18 Ça ne change rien à la probabilité à calculer, que A soit mobile ou, comme ici, fixe. On peut aisément le vérifier expérimentalement en modifiant le script suivant.

La fonction *alea* calcule un angle aléatoire entre 0 et t. Son cosinus et son sinus seront les coordonnées de l'extrémité B de la corde s. Les cordes, une fois construites, sont nommées automatiquement s1, s2, etc. Alors #{s}<sqrt(3) devient successivement égal à *s1<sqrt(3)*, *s2<sqrt(3)* etc. Ces expressions valent « vrai » ou « faux » selon que la corde mesure moins ou plus que $\sqrt{3}$. Alors,

- si la corde mesure moins que $\sqrt{3}$, elle est colorée en rouge et le nombre *nrouges* de cordes rouges est incrémenté
- sinon, la corde est colorée en bleu.

```
alea = (t) -> Math.random()*t/360*2*pi
nrouges = 0
for n in [1..100]
    a=alea 360
    b=Point cos(a), sin(a)
    s=Segment "A", b
    SetExpressionValue "E1", "#{s}<sqrt(3)"
    if GetExpressionValue "E1"
        SetColor s, "red"
        nrouges++
    else
        SetColor s, "blue"
Alert nrouges/100
```

Après avoir exécuté le script, la proportion de segments rouges est affichée[19], et le dessin suivant apparaît :



19 La variable *nrouges* contient alors le nombre de cordes rouges, d'après l'algorithme

On voit alors pourquoi la proportion de cordes coloriées en rouge avoisine les deux tiers : Les angles correspondant vont entre -120° et +120°, soit les deux tiers du cercle.

La probabilité cherchée est deux chances sur 3. Sauf que d'autres protocoles donnent des résultats différents... Par exemple, un protocole inspiré du troisième protocole de Bertrand[20] donne une figure peu lisible[21] mais affiche une probabilité d'une chance sur deux :

```
alea = (t) -> Math.random()*t/360*2*pi
nrouges = 0
for n in [1..100]
    theta = alea 360
    r = Math.random()
    [x,y]=[r*cos(theta),r*sin(theta)]
    milieu = Point x, y
    dd = Ray "O", milieu
    droite = Perpendicular dd, milieu
    [a,b] = Intersections("c1",droite).split ","
    s = Segment a, b
    SetHide "#{milieu},#{dd},#{droite},#{a},#{b}", true
    SetExpressionValue "E1", "#{s}<sqrt(3)"
    if GetExpressionValue "E1"
        SetColor s, "red"
        nrouges++
    else
        SetColor s, "blue"
Alert nrouges/100
```

Alain Busser
Lycée Roland-Garros
Le Tampon

20 La différence, c'est que le nuage de points choisis dans le disque n'est pas uniformément réparti. On trouve donc l'image de l'anneau par une mesure autre que celle de Lebesgue.

21 On y devine toutefois le fait que pour que la corde mesure moins que la côté du triangle équilatéral, son milieu doit être choisi à l'extérieur d'un disque.