Traitement du sujet de bac S Polynésie 2013 avec CoffeeScript

Dans ce document, je vais essayer d'expliquer pourquoi, en voyant le sujet du bac 3013 Polynésie (exercice 1, avec un algorithme), le choix du langage de programmation CoffeeScript m'est paru tout-à-fait naturel : Quoique aussi universel que les langages de programmation les plus connus[1], CoffeeScript[2] a quelques particularités qui le prédestinent à ce genre d'exercice. Ces particularités viennent d'un héritage multiple, l'auteur du langage[3] s'étant visiblement inspiré de Python et Haskell pour créer CoffeeScript.

1. Le sujet Polynésie 2013 porte sur une fonctionnelle, c'est-à-dire une fonction qui porte sur des fonctions (comme antécédents, à la place des nombres) ; et justement, sous l'influence d'Haskell, CoffeeScript est un langage fonctionnel, c'est-à-dire dont l'objet de base est la fonction, qui ici peut très bien être une fonction de fonctions...
2. L'algorithme est court, et la concision de CoffeeScript (en grande partie due à l'influence de Python, notamment avec l'usage de l'indentation pour représenter la structure du programme) permet de conserver cette brièveté dans la traduction de l'algorithme en CoffeeScript
3. On verra plus bas les avantages de CoffeeScript en notation d'intervalles et en conditions de sortie de boucles, donnant au programme un aspect « naturel » (relativement proche de la langue française). Ce sera accentué par les noms donnés aux variables, qui, avec la possibilité de ne pas écrire de parenthèses (héritage de Ruby) rend la proximité avec la langue française encore plus remarquable.

Pour simplifier encore la programmation, deux choix supplémentaires ont été fait :

- Le fichier underscore.js[4] permet de plus facilement implémenter une somme de liste[5]
- Comme le compilateur de CoffeeScript est écrit en JavaScript, celui-ci a été (ainsi que underscore.js) intégré dans une figure CaRMetal. Cela permet de tester les scripts sous CaRMetal et même de les faire interagir avec la figure.

I/ Avec N rectangles

Comme la valeur approchée de l'intégrale est obtenue par addition des aires de N rectangles, et que CoffeeScript, encore très jeune pour un langage, n'a pas de « module » permettant d'additionner les éléments d'une liste (ou tableau) de longueur quelconque, on va utiliser une fonction de underscore.js appelée « reduce [6]» (on réduit la liste à un seul nombre, la somme de ses éléments). Pour appliquer cette fonction à une liste, on doit dire comment on réduit, et comme CoffeeScript est

---

1 En termes d'informatique théorique, on dit « Turing complet » pour « universel ». Cela veut juste dire que ces langages de programmation permettent de calculer tout ce qui est calculable, en termes un peu pompeux. Mais ça fait toujours un bel effet dans les dîners de famille, pour décourager celui qui veut qu'on lui répare son ordinateur en commençant sa question par « au fait, toi qui bosses là-dedans... »

2 Sur www.coffeescript.org, on peut non seulement le télécharger mais aussi le tester en ligne, en cliquant sur « try CoffeeScript »

3 Jeremy Askhenas est célèbre dans la communauté Ruby, et a lancé un projet appelé underscore.js visant à amener à JavaScript tout ce qui fait le succès de Ruby

4 Téléchargeable ici : www.underscorejs.org

5 C'est-à-dire la fonction qui, à une liste de nombres (tableau en JavaScript), associe la somme de ses éléments

6 Encore un héritage de Python, d'ailleurs remplacé par « sum » dans Python 3

un langage fonctionnel, la réduction se fait par une fonction à fournir. Il s'agit de la fonction qui, à 2 nombres, associe leur somme :

```
sommeDe = (liste) -> _.reduce liste, (a,b) -> a+b
```

Alors, pour additionner tous les nombres qui sont dans la liste maListe, il suffit d'écrire sommeDe maListe... C'est ce qu'on va faire pour calculer l'intégrale.

Ensuite, puisqu'on veut estimer l'intégrale d'une fonction f, il faut évidemment programmer cette fonction en CoffeeScript. Pour améliorer la lisibilité de la suite, plutôt qu'appeler cette fonction f, je préfère l'appeler laFonction :

```
laFonction = (x) -> (x+2)*Math.exp -x
```

La syntaxe est assez naturelle si on pense que les deux signes « moins-supérieur » représentent une flèche : à x, la fonction associe (x+2) fois l'exponentielle de -x...

Reste maintenant à écrire la fameuse fonctionnelle, qui va s'appeler aireApprochéeDe puisqu'elle fournit une aire approchée. Comme c'est une fonctionnelle, elle se note comme les deux fonctions précédentes, avec une fonction f comme antécédent (notée entre parenthèses, avant la flèche), et renvoie la somme des f(x/N) divisée par N, pour des x entiers allant de 0 (compris) à N (non compris). Le fait que N est non compris se note par les trois points (au lieu de 2) :

```
aireApprochéeDe = (f) ->
    sommeDe (laFonction(x/N)/N for x in [0...N])
```

L'expression entre parenthèses est une liste de valeurs de f. Donc sommeDe calcule effectivement la somme des termes de cette liste. Pour compléter l'écriture de l'algorithme en CoffeeScript, il suffit donc d'ajouter le fait que N=4 et d'afficher le résultat, lequel, grâce à tout le travail de préparation, s'écrit simplement aireApprochéeDe laFonction. Essayez d'imaginer quelqu'un dictant au téléphone un programme en CoffeeScript : À part la nécessité de veiller à l'indentation, on comprend assez facilement ce qui se passe[7]. Le programme complet sous CaRMetal est donc celui-ci :

```
sommeDe = (liste) -> _.reduce liste, (a,b) -> a+b
laFonction = (x) -> (x+2)*Math.exp -x
N = 4
aireApprochéeDe = (f) ->
    sommeDe (laFonction(x/N)/N for x in [0...N])
Println aireApprochéeDe laFonction
```

Ceux qui n'ont pas la chance inouïe d'avoir une copie de CaRMetal[8] sous la main, devront remplacer Println par alert...

7 Cette simplicité est réductrice, puisqu'en se contentant de dire que la valeur approchée de l'intégrale est la somme des f(x/N)/N, on omet de dire comment on effectue l'addition. Je propose dans la suite de parler d' « algorithmique superficielle » si on ne décrit un algorithme que dans ses grandes lignes sans aller en profondeur. Les algorithmes de tri s'y prêtent particulièrement bien, les activités de statistique répondant à des questions comme « qu'est-ce qu'on fait avec une liste triée » plutôt que « comment on trie la liste ».

8 On peut y remédier en téléchargeant celui-ci ici : http://db-maths.nuxit.net/CaRMetal/telechargement.html

Pour lancer un script Coffee dans CaRMetal, on utilise quelques lignes de JavaScript qui compilent le fichier CoffeeScript (un objet de type « texte » dans la figure CaRMetal) après avoir chargé CoffeeScript et underscore. Pour voir le détail, voir les scripts dans le fichier ci-joint (et si on veut voir les fichiers source, les rendre visibles avec la baguette magique).

II/ Faire varier le nombre de rectangles

Pour rendre le script plus interactif[9], on peut créer une variable numérique dans CaRMetal, appelée E2, et dans laquelle se trouve f1(E1) (E1 étant une expression numérique, et f1 la fonction de l'énoncé, représentée graphiquement). Alors la fonction devient, en CoffeeScript :

```
laFonction = (x) ->
    SetExpressionValue "E1", x
    GetExpressionValue "E2"
```

En bref, on met x dans E1 et on lit son image dans E2. Cela permet de facilement changer de fonction[10]. Ensuite, on modifie la fonctionnelle pour qu'elle admette également N en antécédent :

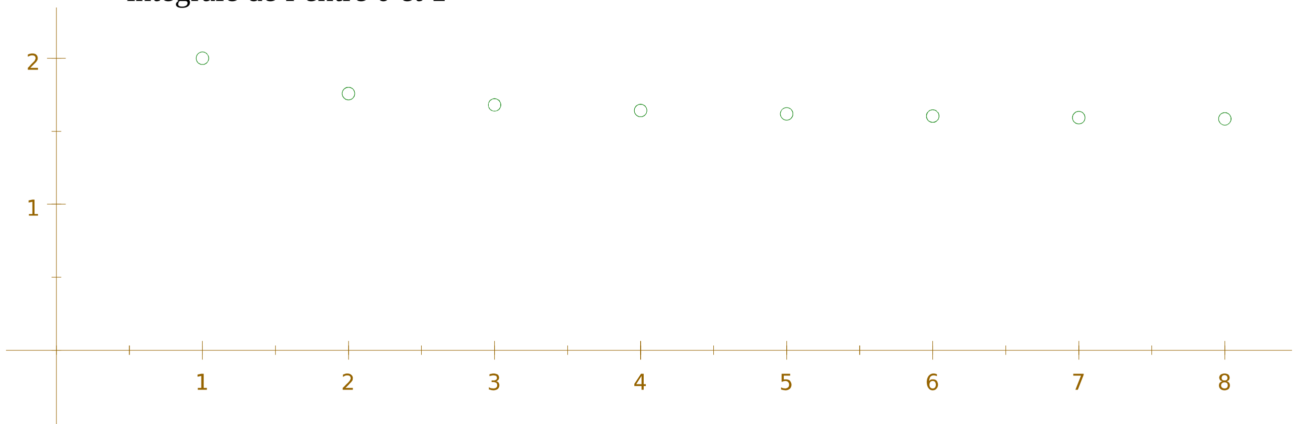
```
aireApprochéeDe = (f,N) ->
    sommeDe (laFonction(x/N)/N for x in [0...N])
```

Du coup, il devient facile de représenter graphiquement la suite des aires approchées indexées par N, en dessinant des points dans une boucle :

```
(Point N, aireApprochéeDe laFonction, N for N in [1..8])
```

On y gagne plusieurs choses :

1. On rappelle que l'aire approchée dépendant de N, on obtient non pas un nombre, mais une suite
2. on rappelle comment on fait pour représenter graphiquement une suite (ça ne fait pas de mal, surtout en Terminale)
3. on constate que cette suite converge, vers une limite qu'il est assez naturel d'appeler « intégrale de f entre 0 et 1 »



Du coup, on peut se poser des questions sur la vitesse de convergence, et chercher N tel que

9 Entrer une valeur numérique pour N chaque fois qu'on veut tester l'algorithme, on fatigue vite, alors un peu d'engagement direct est vite préférable à la gestion des entrées de données comme celle vue dans le programme.

10 Par exemple pour voir si l'intégrale est aussi bien approchée avec d'autres types de fonctions...

l'intégrale est approchée à epsilon près...

III/ Chercher un seuil

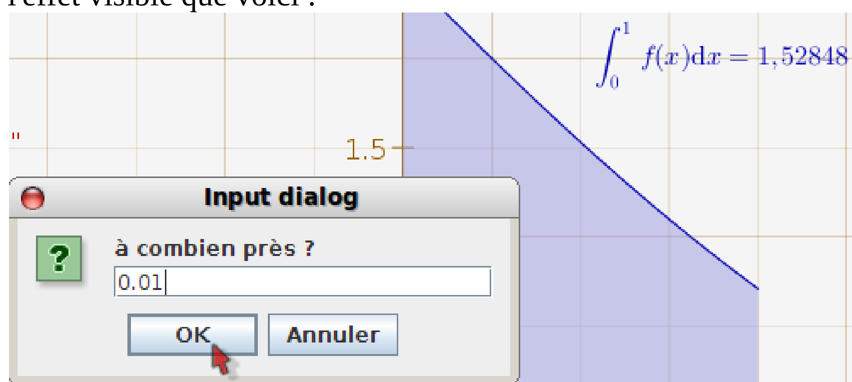
La puissance de CoffeeScript atteint ses limites (elle aussi...) avec cette boucle à condition de sortie. Mais alors que l'expression « jusqu'à ce que » vient spontanément chez les élèves qui veulent décrire un algorithme à condition d'arrêt, son contraire « tant que » semble beaucoup moins naturel chez eux. Or la plupart des outils d'algorithmique ne possèdent que cette boucle « while » et pas la boucle « until » tant désirée. Sauf, évidemment, CoffeeScript[11] !

Comme CaRMetal sait assez bien calculer numériquement des intégrales avec une précision satisfaisante[12], on peut obtenir cette intégrale précise en créant dans la figure une expression appelée E3 et contenant integrate(f1;0;1) ; puis lire cette expression dans le script Coffee, avec GetExpressionValue, pour obtenir la condition d'arrêt souhaitée.

Pour rester dans l'esprit du programme avec son entrée de données obligatoire, on peut demander à l'utilisateur d'entrer eps (le nom choisi pour la précision), avec :

```
eps = Input "à combien près ?"
```

Son exécution a l'effet visible que voici :



et pour effet invisible, l'affectation de la variable eps (ci-dessus, par 0,01).

Il y a deux raisons pour lesquelles il pourrait ne pas être prudent de boucler :

- eps peut être trop grand (on ne souhaite pas calculer l'intégrale à 100 unités près) ou négatif (aucun sens)
- eps peut être trop petit (la boucle est trop longue à effectuer car, dans la boucle, beaucoup de calculs sont faits ; notamment la constitution d'une liste et les additions pour sommer ses éléments). Concrètement on va éviter de calculer des intégrales à moins d'un millième près (eps>0,001)

Encore un avantage de CoffeeScript : Ces deux tests sur eps peuvent être faits en même temps :

```
if 0.001 < eps < 1
    n = 1
```

11 La boucle « until » existait dans le langage Pascal ; depuis elle a disparu, pour une raison que je ne m'explique pas ; alors que « while » oblige souvent à manipuler une négation, transformation linguistique avec laquelle les élèves sont rarement à l'aise...

12 Avec la méthode des trapèzes, l'algorithme de Romberg étant appelé à la rescousse pour accélérer la convergence ; algorithme implémenté en Java par René Grothmann

```
vraie = GetExpressionValue "E3"
until Math.abs(appr-vraie) < eps
    appr = aireApprochéeDe laFonction, n
    n++
    Println "#{n} rectangles : #{appr}"
```

L'écriture formatée donne l'affichage suivant avec eps=0,01 :

```
2 rectangles : 2
3 rectangles : 1.7581633246407917
4 rectangles : 1.6803395695863623
5 rectangles : 1.6419091078075088
6 rectangles : 1.619001424125957
7 rectangles : 1.603791781012287
8 rectangles : 1.5929580368315515
9 rectangles : 1.584849242848615
10 rectangles : 1.5785521707795798
11 rectangles : 1.5735206588839377
12 rectangles : 1.5694080269433002
13 rectangles : 1.565983622769119
14 rectangles : 1.5630880294322862
15 rectangles : 1.5606075358895213
16 rectangles : 1.5584588523726368
17 rectangles : 1.5565795748815288
18 rectangles : 1.554922024754417
19 rectangles : 1.5534491472509284
20 rectangles : 1.552131708628677
21 rectangles : 1.5509463361386033
22 rectangles : 1.5498741193170897
23 rectangles : 1.5488995935864744
24 rectangles : 1.548009989563672
25 rectangles : 1.5471946704218797
26 rectangles : 1.546444704548214
27 rectangles : 1.5457525370033687
28 rectangles : 1.5451117341218457
29 rectangles : 1.5445167829358606
30 rectangles : 1.5439629321674384
31 rectangles : 1.5434460650737205
32 rectangles : 1.5429625969415346
33 rectangles : 1.5425093918310024
34 rectangles : 1.5420836944790322
35 rectangles : 1.5416830742370675
36 rectangles : 1.541305378632904
37 rectangles : 1.540948694682679
38 rectangles : 1.5406113164848743
39 rectangles : 1.540291717937613
40 rectangles : 1.539988529658556
41 rectangles : 1.5397005193710291
42 rectangles : 1.5394265751639007
43 rectangles : 1.5391656911456737
44 rectangles : 1.5389169551026112
45 rectangles : 1.5386795378417126
46 rectangles : 1.5384526839561659
```

On y voit également la convergence, et il est maintenant facile de modifier le script pour le transformer en recherche de la plus petite valeur de N telle que la somme des N aires vaut l'intégrale à epsilon près. Ce qui est fait dans le dernier onglet du fichier CaRMetal ci-joint, avec en prime le dessin des rectangles, parce que CaRMetal reste avant tout un logiciel de géométrie dynamique...