Algorithmique et tableur

- N°23 - Janvier 2011 -

Date de mise en ligne : dimanche 12 dÃ©cembre 2010

Description :

Algorithmique ou tableur ? Telle est la question. Voire...

Les nouvelles technologies pour l'enseignement des mathématiques

L'usage de l'algorithmique risque de supplanter celui du tableur, bien que souvent (et notamment quand on travaille sur des suites), l'activité sous tableur soit nettement plus facile que celle avec algorithmique. Mais il est possible de faire de l'algorithmique au sein même du tableur !

Le tableur choisi ici est Open Office, pour les raisons suivantes :

1. Il possède 4 langages de programmation, plus un cinquième avec *CMath* ;
2. Il est libre, ce qui en cas de problème permet de mettre les mains dans le cambouis pour arranger lesdits problèmes ;
3. Seul OOo ("OpenOffice.org") permet de faire tourner la barre d'outils *CMath* qui lui offre à la fois le calcul formel [1] et un langage de programmation multiforme (celui d'Xcas) ;
4. Ce tableur est multiplateforme et libre, ce qui n'exclut personne de son utilisation.

Fibonacci

Pour illustrer la différence entre la démarche algorithmique et la manipulation du tableur, on va considérer l'exercice suivant :

> Calculer les 20 premiers nombres de Fibonacci.

Un algorithme possible est le suivant :

Variables a,b (entiers)

Initialisation :

$a\leftarrow 1$

$b\leftarrow 1$

Pour $i\leftarrow 2$ jusque 20 :

$a\leftarrow b$ et simultanément $b\leftarrow a+b$

Afficher b

Fin pour

Seulement cet algorithme qui utilise deux affectations simultanées, est basé sur du calcul parallèle. Si on essaye de faire les deux affectations l'une après l'autre, la boucle calcule les puissances de 2 au lieu des nombres de Fibonacci !

Il est donc nécessaire de créer une troisième variable, dite "tampon" et notée *t*, où on stocke temporairement *b* afin de pouvoir écraser celui-ci par *a+b* pour ensuite récupérer son ancienne valeur, et la placer dans *a*. Écrit en JavaScript, l'algorithme devient alors

```
var a=1;
var b=1;
for(n=2;n<=20;n++){
  t=b;
  b=a+b;
  a=t;
  Println(b);
}
```

C'est pareil avec l'algorithme d'Euclide

La boucle de l'algorithme d'Euclide nécessite aussi une affectation simultanée (d'une variable par l'autre, et de l'autre variable par le reste dans la division euclidienne). Là encore, l'intérêt d'une troisième variable *t* pour stocker temporairement l'une des deux autres, se fait sentir. Voir un exemple ici. Exception : Un algorithme n'utilisant que deux variables est visible dans le sujet du Rallye mathématique 2010 de la Réunion. Mais, faisant appel à des tests et des soustractions, il est plus long que les implémentations classiques.

Et l'exemple de la page 18 du livre de Guillaume Connan montre que, même pour échanger les contenus de deux variables, ce n'est pas si simple si on ne veut pas en utiliser une troisième !

En fait on peut faire en même temps

Certes, si on utilise un langage de programmation tel que Lua, qui permet l'affectation simultanée de deux variables, on peut économiser la troisième variable, comme le montre l'exemple en Python (langage) :

```
>>> a,b=1,1
>>> for n in range(2,20):
        a,b=b,a+b
        print(b)
```

Et comme rien ne vaut l'expérimentation par soi-même, essayer à l'occasion d'aller sur l'interpréteur en ligne de Lua et d'y copier-coller le script suivant :

```
 a=1
b=1
for n=2 , 20 do
    a, b=b, a+b
    print(b)
end
```

Après un clic sur "Run", on devrait voir les nombres de Fibonacci !

Scoop : Si, comme ici, il n'y a que deux variables à affecter en même temps, CaRMetal le permet, à condition de stocker ces deux variables comme coordonnées d'un point. Au sein de ce paradigme, l'affectation simultanée des deux variables s'illustre par le mouvement du point :

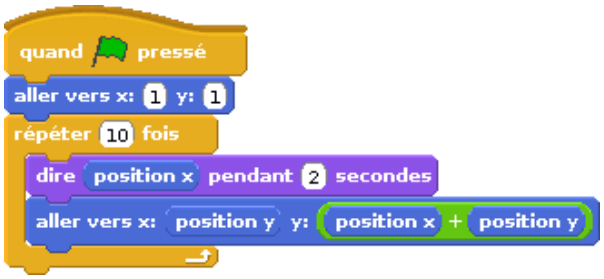
```
var p=Point(1,1);
for(n=2;n<=20;n++){
  Move(p,Y(p),X(p)+Y(p));
  Println(Y(p));
}
```

La même technique s'applique aussi à l'échange de deux variables et à l'algorithme d'Euclide puisque ces problèmes utilisent deux variables. Ceci dit, ce qu'on gagne en place mémoire, on le perd largement en rapidité, les mouvements du point étant assez lents. Pour mesurer le temps mis à calculer les 20 nombres de Fibonacci, on peut ajouter 3 lignes au CaRScript ci-dessus :

```
var debut=new Date();
var p=Point(1,1);
for(n=2;n<=20;n++){
  Move(p,Y(p),X(p)+Y(p));
  Println(Y(p));
}
var fin=new Date();
Println(fin-debut);
```

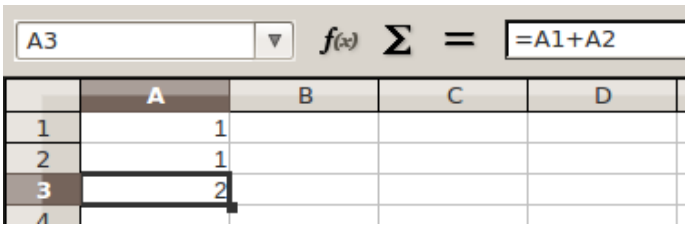
Ce qui, avec un affichage de *1857* final, affirme qu'il a fallu 1,857 seconde pour faire ce calcul, alors qu'avec les autres méthodes, la réponse est pratiquement instantanée...

À titre de comparaison, voici la version *Scratch* :

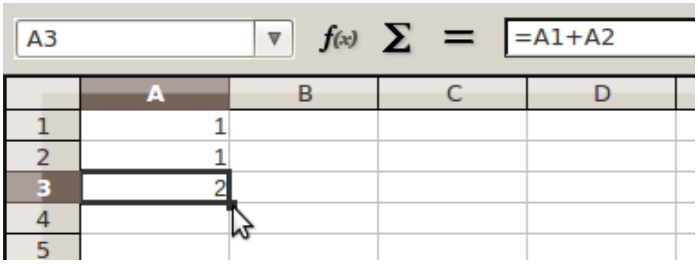


Avec un tableur, l'itération est tellement bien cachée que la manipulation est beaucoup plus facile :

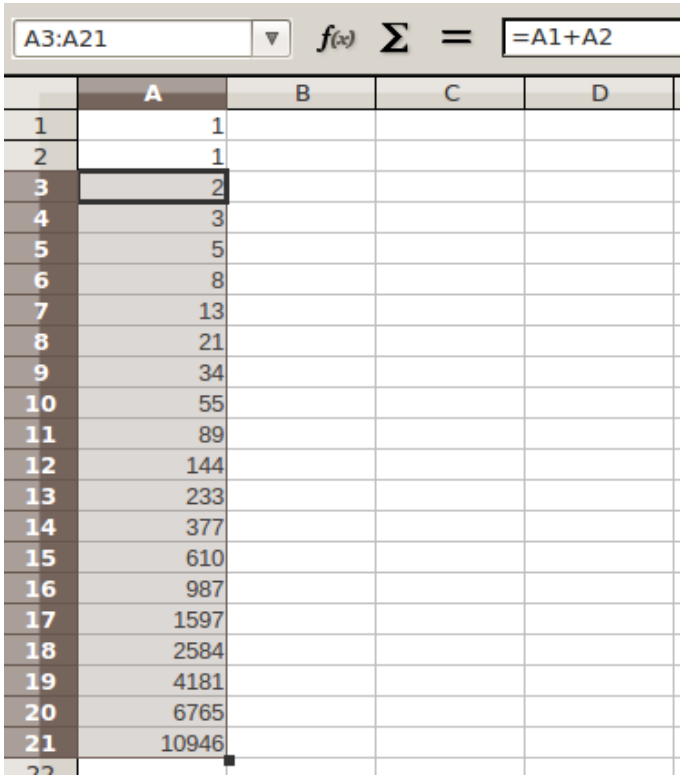
On commence (initialisation !) par placer un "1" dans les cellules A1 et A2 ; puis, en A3, on écrit la formule *=A1+A2* (presque sans utiliser le clavier, il suffit de cliquer sur une cellule pour faire apparaître son nom dans la formule) :



Puis, une fois la cellule A3 sélectionnée en cliquant dessus, il suffit de tirer avec la souris, la petite poignée qui est en bas à droite de la cellule :



En relâchant celle-ci, on a instantanément les nombres de Fibonacci :



Difficile d'imaginer plus simple !

On peut faire des tableaux sans tableur.

La plupart des langages de programmation possèdent des tableaux, Lua est même spécialisé dans les tableaux (on peut citer AlgoBox, Scratch, Python, Euler Math Toolbox, Octave, SciLab, Xcas, ...) !

Avec l'objet *Array* de JavaScript, on peut faire ainsi, avec affichage de tout le tableau d'un coup :

```
var Fibo=new Array(1,1);
for(n=2;n<=20;n++){
  Fibo[n]=Fibo[n-1]+Fibo[n-2];
}
Print(Fibo);
```

Les tableaux ne sont pas au programme d'algorithmique du lycée (sauf en Terminale L), mais ils sont utiles notamment en statistiques, et ne sont pas difficiles à manipuler, surtout avec des langages de programmation qui possèdent des fonctions agissant sur les tableaux (tri, quantiles, somme, moyenne, comptage, bref ce qui se trouve dans un tableur...). Et les élèves de Seconde semblent vite à l'aise avec la notion de tableaux, ne serait-ce qu'avec les tableaux de valeurs du cours.

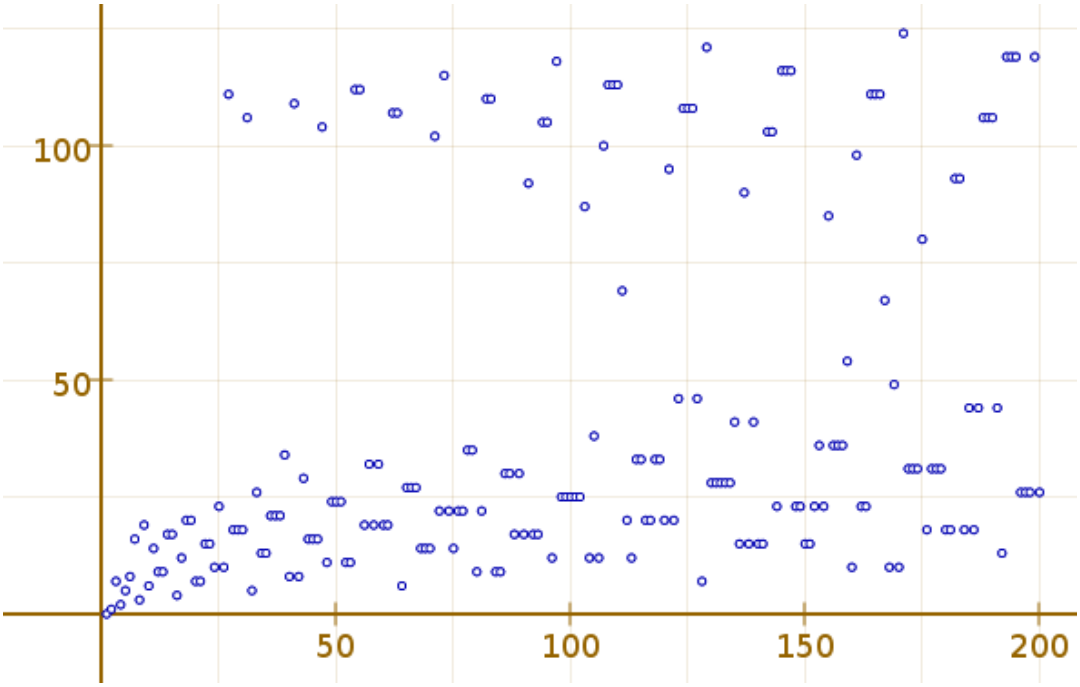
++++ BASIC

Dans la suite, on va s'intéresser à un problème suffisamment spécifique pour que le tableur *OOoCalc* ne risque pas trop de se voir doté des fonctionnalités permettant de l'étudier : Le temps de vol de la conjecture de Syracuse.

On définit donc la suite de Collatz de façon récurrente par

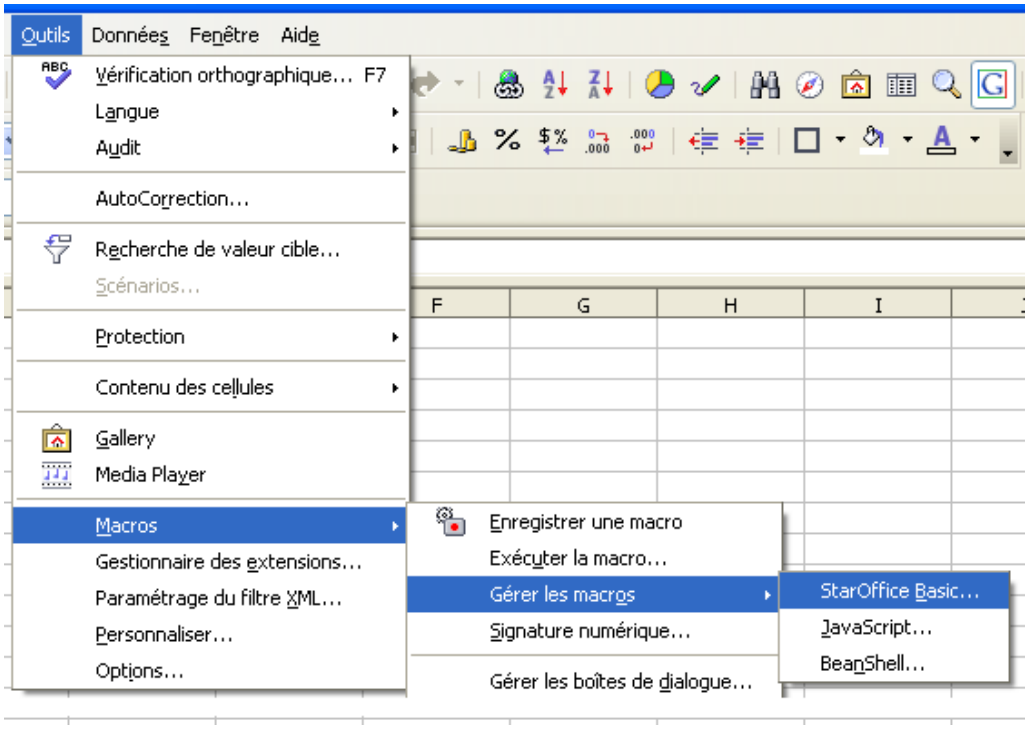
$$u_{n+1}=\left\{\begin{array}{l} \frac{u_n}{2} \mbox{ si } \frac{u_n}{2}\in N \\ 3u_n+1 \mbox{ sinon} \end{array} \right.$$

Alors le *temps de vol* d'un entier u_0 est le plus petit entier n tel que $u_n=1$. L'énoncé de la conjecture de Syracuse devient alors : Pour tout $u_0 \in N$, le temps de vol de u_0 est fini. Pour montrer combien ce temps de vol est chaotique, voici sa représentation graphique :

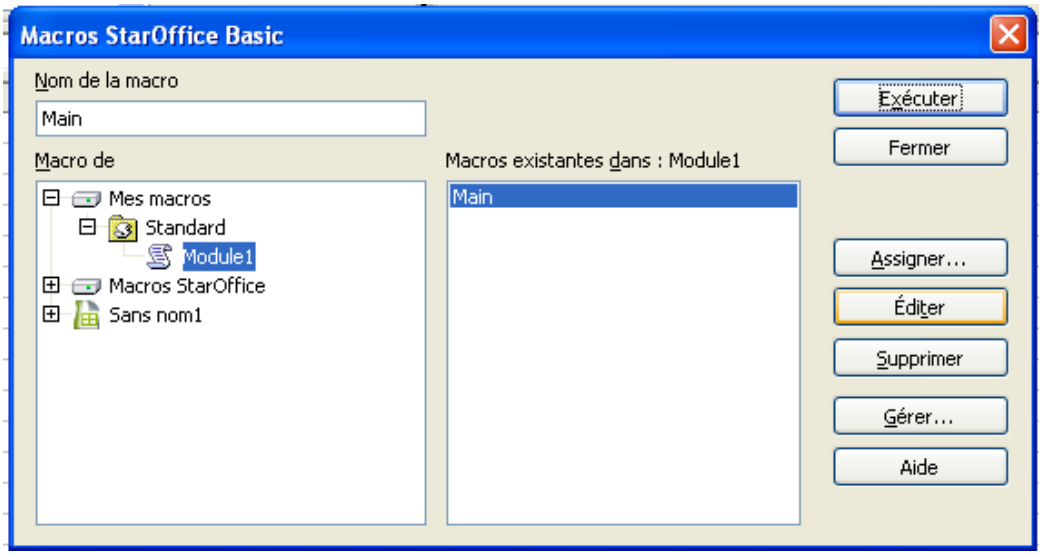


OOCalc ne possède pas de fonction "temps de vol" alors on va en quelque sorte enrichir le tableur avec deux nouvelles fonctions : *Collatz* qui calcule la suite, et *Vol* qui calcule le temps de vol.

Dans cet onglet, on va utiliser le langage BASIC qu'Open Office a hérité de *Star Office*. En cliquant sur "Outils" puis en choisissant parmi les "Macros", "Gérer les macros", on devrait voir au moins la catégorie "Star Office Basic" :



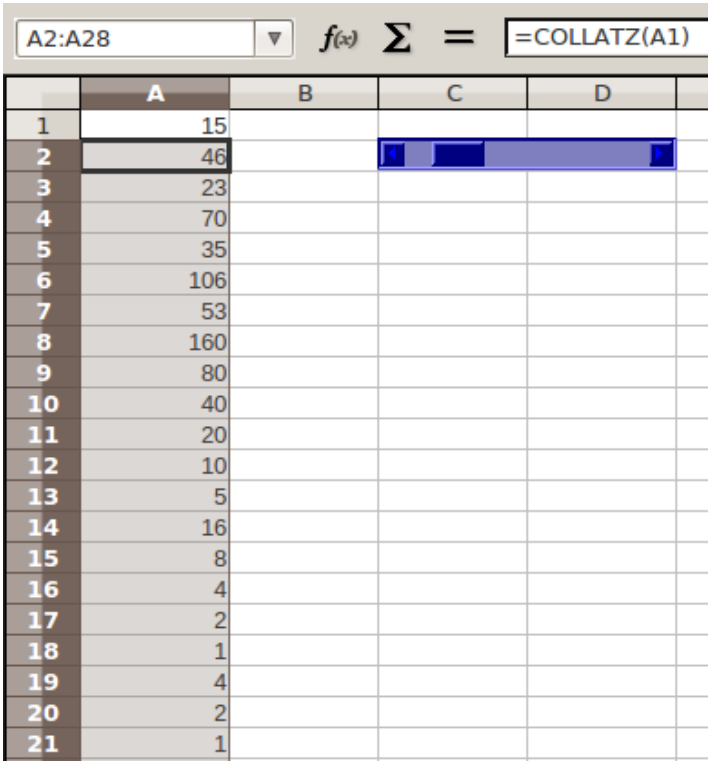
On peut créer des macros par enregistrement d'une série d'actions, ou en programmant en Basic. C'est ce qu'on va faire ici. Parmi les macros Basic présentes, on a au moins un "Module 1" dans "Standard". Au début, celui-ci contient une méthode "Main" qu'on peut modifier en cliquant sur "éditer" :



C'est à partir de là qu'on peut commencer à programmer. La fonction *Collatz* peut porter sur de grands entiers ("Long") et se programmer ainsi :

```
Function Collatz(Un as Long) as Long
    If Un Mod 2 Then
        Collatz=3*Un+1
    Else
        Collatz=Un/2
    Endif
End Function
```

C'est magique : Sitôt le programme entré, OOCalc possède une fonction "Collatz" (qui s'écrit en majuscules dans le tableur) et qu'on peut utiliser comme n'importe quelle autre fonction du tableur ! Ainsi la formule *=Collatz(A1)*, entrée dans A2 puis recopiée vers le bas, donne la suite de Collatz dans la colonne A :

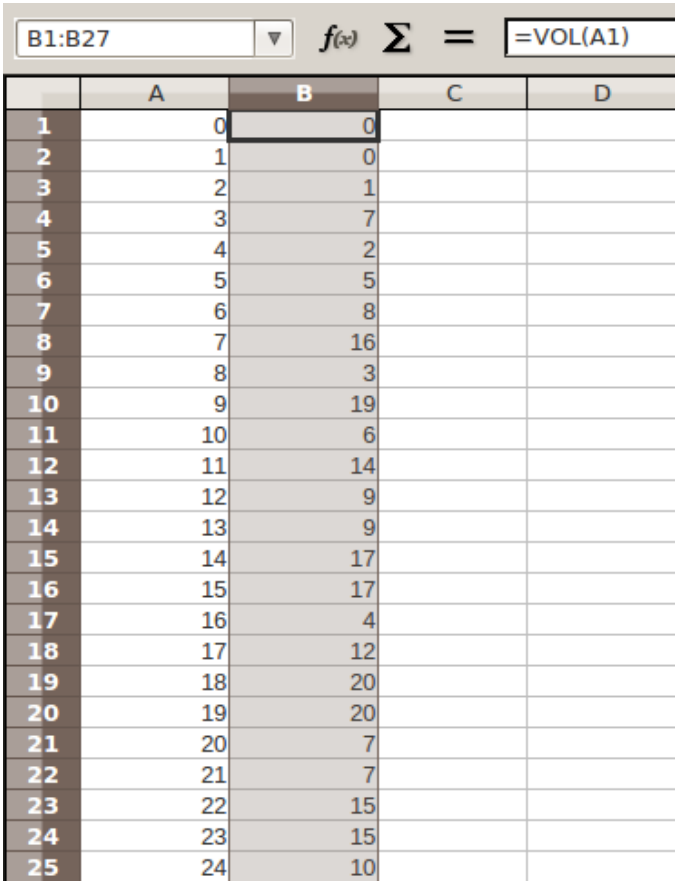


(On remarque ci-dessus que le contenu de A1 n' a pas été entré numériquement mais piloté par un curseur, en utilisant la technique exposée dans cet article). On peut ensuite aisément représenter graphiquement la suite, et ce de façon dynamique puisque l'action sur le curseur modifie presque instantanément la représentation de la suite. On comparera avec la version CaRMetal.

Puis on peut utiliser la fonction *Collatz* (maintenant qu'elle est là) pour créer avec une boucle "tant que", le temps de vol :

```
Function Vol(U0 as Long) as Long
    u=U0
    t=0
    While(u>1)
        t=t+1
        u=Collatz(u)
    Wend
    Vol=t
End Function
```

Dorénavant, on dispose du temps de vol, comme une fonction du tableur. Alors pour finir l'activité, il suffit d'entrer les entiers successifs dans la colonne A, puis en B1, de mettre la formule *=Vol(A1)*, qui, une fois copiée vers le bas, donne le tableau suivant :



	A	B	C	D
1	0	0		
2	1	0		
3	2	1		
4	3	7		
5	4	2		
6	5	5		
7	6	8		
8	7	16		
9	8	3		
10	9	19		
11	10	6		
12	11	14		
13	12	9		
14	13	9		
15	14	17		
16	15	17		
17	16	4		
18	17	12		
19	18	20		
20	19	20		
21	20	7		
22	21	7		
23	22	15		
24	23	15		
25	24	10		

Après ça on peut sélectionner les deux colonnes A et B puis insérer un graphique dans le tableur, pour avoir quelque chose d'analogue à la figure qui ouvre cet onglet.

Cette suite de manipulations est suffisamment simple pour qu'on puisse envisager d'utiliser le langage Basic pour enseigner l'algorithmique en Seconde. Un avantage est la facilité des entrées de données (l'algorithme classique consistant à entrer les coordonnées de 3 points pour faire une série de diagnostics sur la nature du triangle est pénible à évaluer parce qu'il faut à chaque essai, entrer 6 nombres. Avec le tableur, surtout avec des curseurs, c'est tout de suite beaucoup moins pénible...). La coloration syntaxique et le fait qu'il s'agisse d'un Basic structuré jouent également en faveur de cet outil. Et ce langage Basic est aussi multiplateforme que la suite OOo qui le contient. Et libre, surtout depuis l'apparition de la page suivante Libre Office.

Pour voir ce qu'on peut faire avec le langage Basic d'OOCalc, voir cet article sur la statistique. Bien d'autres choses peuvent être envisagées en maths, avec le langage Basic, par exemple en arithmétique, sur les nombres complexes etc. On peut même faire du calcul formel, comme l'outil présenté dans le dernier onglet.

++++ JavaScript

Dans l'onglet précédent, au moment de choisir le langage *Basic* parmi les outils, on a vu qu'il y avait aussi *JavaScript* parmi les choix. Ainsi, Open Office permet de programmer en JavaScript, et pas seulement dans le tableur ! En opérant comme dans l'onglet précédent, mais au sein de la rubrique *JavaScript*, on peut créer, puis modifier, un fichier JavaScript. Dans le présent exemple, il s'appelle *gauss.js* parce qu'on va essayer, en JavaScript, de remplir la colonne A avec des échantillons pseudo-aléatoires gaussiens centrés et reduits.

La programmation en JavaScript est plus compliquée qu'en Basic parce qu'elle fait appel à de la programmation objet,

le tableur lui-même étant considéré comme un objet.

On a donc besoin des "classes" qui permettent de créer et gérer les objets tableur, classeur, feuille et cellule, ce qui se fait avec ces déclarations préliminaires (qu'on peut fournir aux élèves avant le TP) :

```
 importClass(Packages.com.sun.star.uno.UnoRuntime);
importClass(Packages.com.sun.star.sheet.XSpreadsheetDocument);
importClass(Packages.com.sun.star.sheet.XSpreadsheets);
importClass(Packages.com.sun.star.table.XCellRange);
importClass(Packages.com.sun.star.container.XIndexAccess);
```

Après cela on peut implémenter en JavaScript une simulation de variable aléatoire gaussienne avec la méthode de Box-Muller :

```
 function normal(){
var u=Math.random();
var v=Math.random()*2*Math.PI;
return Math.sqrt(-2*Math.log(u))*Math.cos(v);
}
```

(il s'agit ici d'une fonction sans argument, d'où les parenthèses vides, qui retourne un réel simulant un tirage aléatoire quivant une loi normale centrée et réduite).

Contrairement au langage Basic de l'onglet précédent, le tableur n'est pas doté de fonction "normal" après avoir enregistré le script : On perd l'engagement direct avec JavaScript. Pour que le script puisse agir sur le tableau, il faut

1. passer le document courant au script, ici en l'appelant *oDoc* (parce que c'est un document générique Open Office) ;
2. créer un objet *Tableur* de type tableur, contenant le fichier tableur qu'Uno arrive à trouver dans *oDoc* ;
3. créer un objet *Feuilles*, contenant les feuilles de calcul du tableur ;
4. créer un tableau *pages* contenant les numéros des feuilles actives dans l'objet précédent ;
5. créer un objet *Feuille* (au singulier) affecté avec la première des pages du tableau *pages*. Cet objet contient la feuille de calcul dans laquelle JavaScript va écrire. Toutes ces opérations se résument à ceci :

```
 oDoc = XSCRIPTCONTEXT.getDocument();
Tableur = UnoRuntime.queryInterface(XSpreadsheetDocument, oDoc);
Feuilles = Tableur.getSheets();
pages = UnoRuntime.queryInterface(XIndexAccess, Feuilles);
Feuille = pages.getByIndex(0);
```

Pour remplir la première colonne avec des nombres calculés par la fonction *normal()*, il reste à récupérer les cellules actives de la feuille courante (dans l'objet *Cellules*), puis dans une boucle, récupérer les cellules de la colonne A (avec

la méthode *getCellByPosition* de l'objet *Cellules*) et y écrire les nombres pseudoaléatoires avec la méthode *setValue* :

```
 Cellules = UnoRuntime.queryInterface(XCellRange, Feuille);
for(i=1;i<=20;i++){
xCell0 = Cellules.getCellByPosition(0, i);
xCell0.setValue(normal());
}
```

Après ça, pour remplir la colonne A de nombres gaussiens, il suffit de cliquer sur "Run", mais en appuyant sur le bouton *F9*, rien ne change : Si on veut un nouveau tirage, on doit cliquer à nouveau sur "Run". Ceci dit l'effet poduit est immédiat :

	A
1	
2	-1,915196175
3	-0,149317040
4	0,028279148
5	1,397824540
6	-0,797878208
7	1,451205794
8	0,729611302
9	-1,976661188
10	0,545174738
11	0,834287841
12	0,425635400
13	0,613277497
14	0,911150234
15	-0,500641962
16	-3,002479842
17	-0,180675685
18	-0,734547333
19	0,715536993
20	0,389542571
21	0,290294701
22	

Grâce à la méthode *GetValue*, on peut faire communiquer JavaScript dans les deux sens avec le tableur, mais sans engagement direct puisque là encore, une modification ultérieure du contenu d'une cellule n'aura pas d'effet sur les contenus des cellules qui dépendent d'elle (à moins de relancer le script). De ce point de vue, l'utilisation de JavaScript dans un tableur est didactiquement parlant, moins intéressante que celle du Basic (à moins de vouloir montrer ce que représente le changement de cadre).

Mais le logiciel que l'on fait tourner ici est Rhino (moteur JavaScript), qui permet de déboguer du Javascript, avec

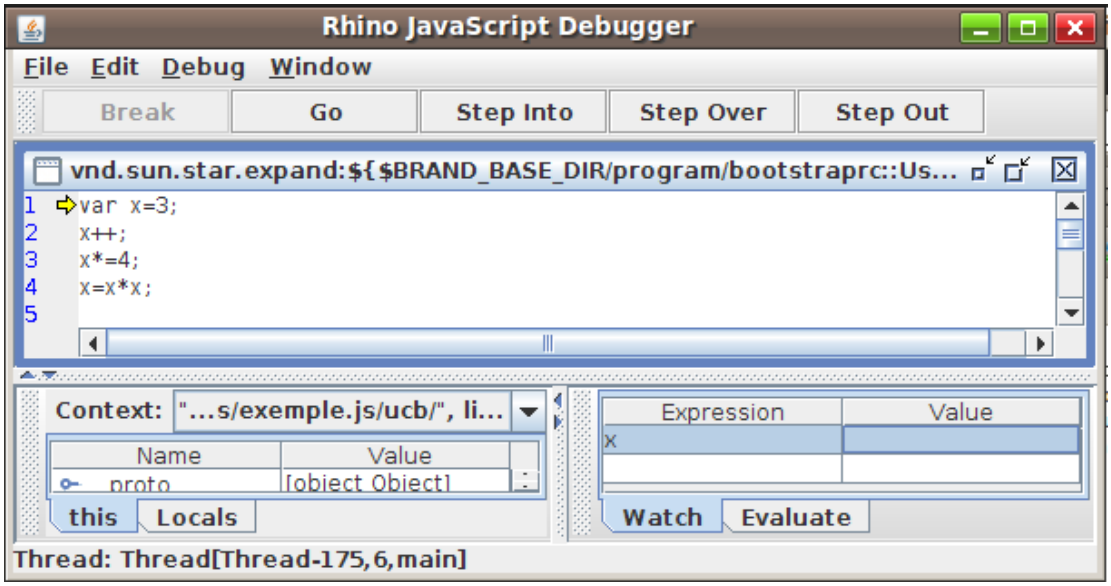
- l'affichage du contenu de variables ;
- le mode pas-à-pas.

Ceci permet alors de faire des TP d'algorithmique en JavaScript sans le tableur, tout simplement en vidant la console JavaScript et en tapant un petit bout de code à tester, puis en cliquant sur "step into" à chaque pas. Par exemple, pour le programme de calcul :

```
 var x=3;
x++;
x*=4;
x=x*x;
```

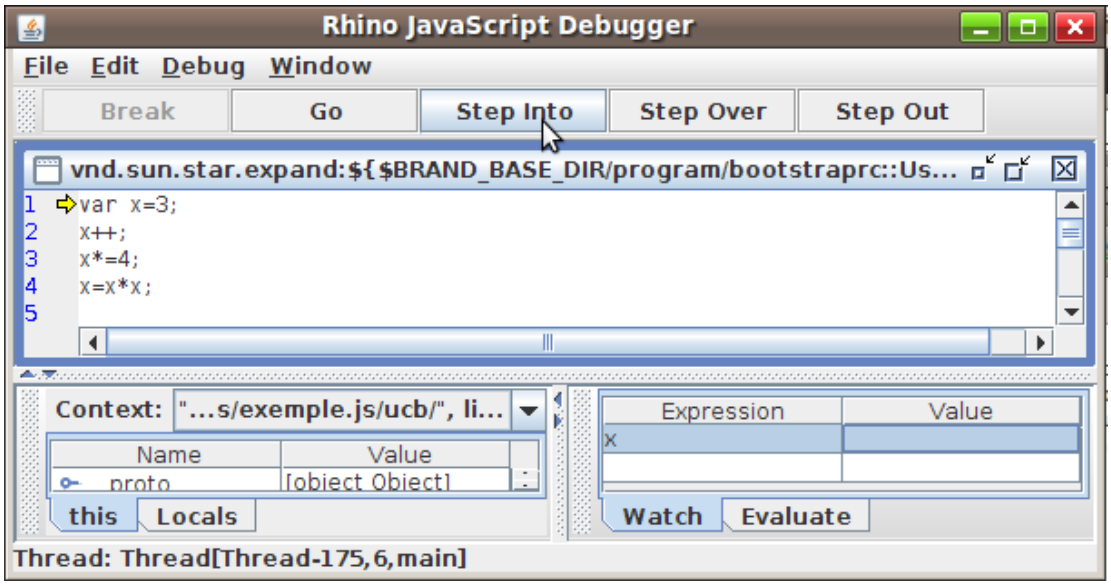
la question la plus importante (au moins en début de Seconde) n'est pas tant "que contient la variable *x* à la fin du script ?" mais bien "pourquoi *x* vaut-il 16 à la fin ?" ce qui oblige les élèves à une reconstitution chronologique des affectations, difficile pour eux. Or voir ce qui se passe (en prenant le temps de le faire), c'est moins abstrait qu'imaginer ce qui s'est passé !

Après avoir entré le script et cliqué sur *File>Run*, on voit en bas de la fenêtre *rhino* un espace où on peut entrer des noms de variables à surveiller (ici *x*) :



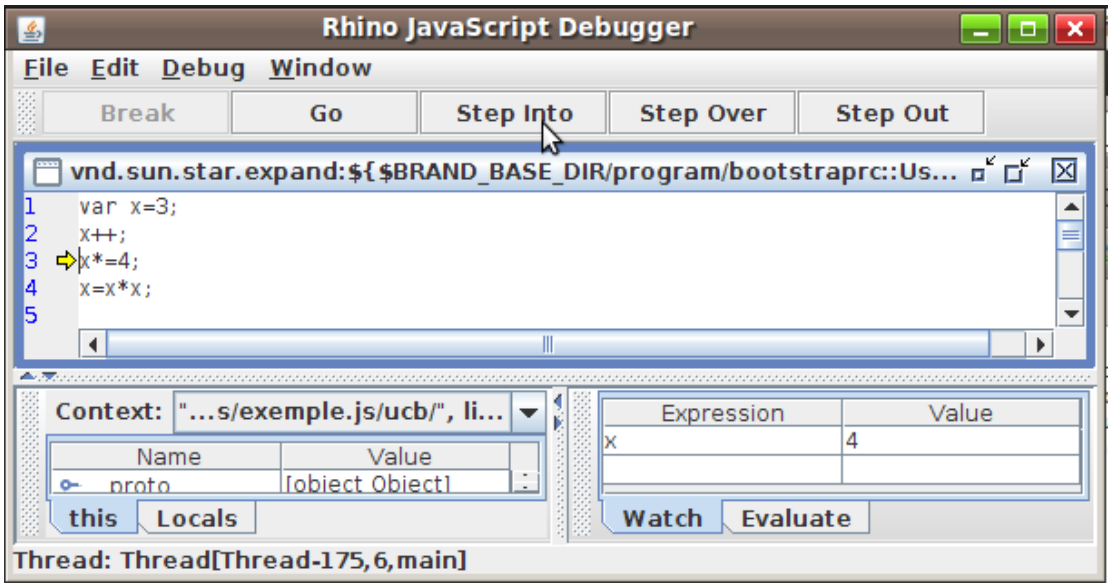
Ci-dessus la case "value" est vide parce que, la ligne 1 n'ayant pas encore été exécutée (la flèche jaune se trouve au début de la prochaine ligne qui sera exécutée), la variable *x* n'a pas encore été créée et ne peut donc rien contenir !

Pour avancer d'un seul pas (n'exécuter que la ligne 1), on clique sur "Step Into" (ou on appuie sur la touche *F11* :



ce qui a pour effet d'aller à la ligne (la flèche jaune se trouve sur les starting blocks de la ligne 2) et d'afficher le contenu actuel de *x*, soit 3. L'intérêt didactique de ce genre d'exercice est que l'élève peut suivre à son rythme

l'évolution de la situation. Ainsi, pour voir ce qui s'est passé lors de l'incrémentation de *x* (ligne 2), on clique une fois de plus sur "Step Into" :

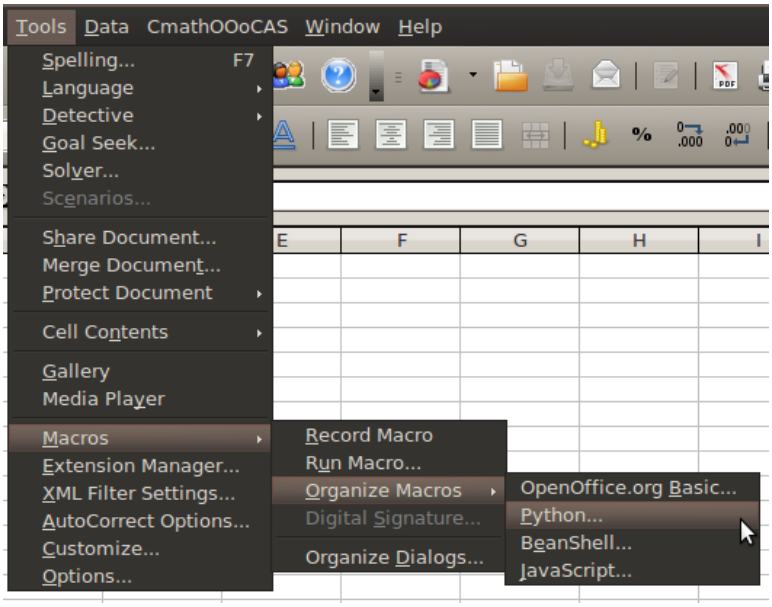


Ce mode pas-à-pas est à comparer avec ceux d'Xcas (même principe mais il faut créer une fonction où mettre tout ça, la compiler et la transmettre comme argument à *debug*) ou Algobox, qui affiche toutes les variables (ce qui dans le cas présent ne change pas grand'chose, vu qu'il n'y en a qu'une !). Et le langage *Basic* d'*Open Office*, décrit dans l'onglet précédent, possède lui aussi un outil de débogage qui permet d'afficher le contenu de certaines variables, afin d'observer leur évolution dans le temps.

Un TP basé sur le mode pas-à-pas est visible dans le quatrième onglet de cet article (le sujet est téléchargeable dans l'onglet, intitulé "nombres triangulaires")

++++ Python

Si on regarde la liste des macros sous Linux, on voit une quatrième entrée, intitulée *Python* :



Contrairement à JavaScript vu dans l'onglet précédent, Python n'est pas doté d'un éditeur de texte avec mode pas-à-pas. Il est donc nécessaire d'utiliser un éditeur de texte pour y entrer le code suivant

```
 from fractions import *

def pgcd( ):
    """Remplit le tableau de plus grands communs diviseurs"""

    model = XSCRIPTCONTEXT.getDocument()
    feuille = model.CurrentController.getActiveSheet()
    for i in range(1,20):
        cellule=feuille.getCellByPosition(i,0)
        cellule.setValue(i)
        cellule=feuille.getCellByPosition(0,i)
        cellule.setValue(i)
    for i in range(1,20):
        for j in range(1,20):
            cellule=feuille.getCellByPosition(i,j)
            cellule.setValue(gcd(i,j))
```

et l'enregistrer sous le nom *pgcd.py*. Quelques explications sur le code :

- La première ligne a pour but d'importer la fonction *gcd* de Python (l'algorithme d'Euclide) ;
- La fonction pgcd, sans argument, va remplir le tableau ;
- *model* est le tableur ;
- *feuille*, comme dans l'onglet précédent, est la feuille de calcul actuellement ouverte ;
- la première boucle sur *i* sert à remplir les premières ligne et colonne, avec les entiers successifs (en bleu ci-dessous) ;
- La double boucle (sur *i* et *j*) calcule les pgcd des nombres bleus (en rouge ci-dessous).

Une fois le fichier *pgcd.py* tapé (en respectant les indentations), il suffit de le placer dans le dossier *Python* des *Scripts* d'*Open Office*. Et en lançant le tableur, on trouve dans la liste des *macros* en *Python*, *pgcd*, qu'il suffit de lancer (*Run*) pour obtenir cette feuille :

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1
4	3	1	1	3	1	1	3	1	1	3	1	1	3	1	1	3	1	1	3	1
5	4	1	2	1	4	1	2	1	4	1	2	1	4	1	2	1	4	1	2	1
6	5	1	1	1	1	5	1	1	1	1	5	1	1	1	1	5	1	1	1	1
7	6	1	2	3	2	1	6	1	2	3	2	1	6	1	2	3	2	1	6	1
8	7	1	1	1	1	1	1	7	1	1	1	1	1	1	7	1	1	1	1	1
9	8	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	8	1	2	1
10	9	1	1	3	1	1	3	1	1	9	1	1	3	1	1	3	1	1	9	1
11	10	1	2	1	2	5	2	1	2	1	10	1	2	1	2	5	2	1	2	1
12	11	1	1	1	1	1	1	1	1	1	1	11	1	1	1	1	1	1	1	1
13	12	1	2	3	4	1	6	1	4	3	2	1	12	1	2	3	4	1	6	1
14	13	1	1	1	1	1	1	1	1	1	1	1	1	13	1	1	1	1	1	1
15	14	1	2	1	2	1	2	7	2	1	2	1	2	1	14	1	2	1	2	1
16	15	1	1	3	1	5	3	1	1	3	5	1	3	1	1	15	1	1	3	1
17	16	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16	1	2	1
18	17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	17	1	1
19	18	1	2	3	2	1	6	1	2	9	2	1	6	1	2	3	2	1	18	1
20	19	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
21																				

(cette version numérique est à comparer avec la version graphique obtenue avec le Gimp)

Comme pour JavaScript, on perd l'interactivité (si on remplace le 2 de la cellule C1 par un 3, cela ne changera pas les pgcd en rouge, qui deviennent alors faux). Et contrairement à JavaScript, on n'a pas d'éditeur Python intégré à Open Office. Par contre, la quantité de code à entrer pour se référer aux objets *tableur*, *feuille* et *cellule* est beaucoup plus réduite qu'avec JavaScript, ce qui permet d'envisager des activités en classe, où le tableur servirait d'interface d'entrée-sortie pour des programmes en Python.

++++ Java

Dans l'onglet consacré au langage Basic, on a vu qu'en cherchant celui-ci dans le menu des langages de programmation d'Open Office, il était suivi d'un mystérieux BeanShell. Ce *BeanShell* est une petite merveille technologique, qui permet d'écrire du code en Java (langage) et de le tester sans avoir à le compiler puisqu'il est interprété. Et Open Office contient ce bijou !

La ressemblance entre Java (langage) et JavaScript incite à faire la comparaison entre la version *JavaScript* des variables normales, déjà vue, et la version *Java*. Donc, comme avec JavaScript, on doit rendre Open Office accessible à *BeanShell* :

```
 import com.sun.star.uno.UnoRuntime;
import com.sun.star.sheet.XSpreadsheetView;
import com.sun.star.text.XText;
import com.sun.star.sheet.XCellRangeData;
```

La fonction *normal()* doit, contrairement à la version *JavaScript*, être déclarée comme un *double* (le langage Java est fortement typé) :

```
 double normal(){
double u,v;
u=Math.random();
v=2*Math.PI*Math.random();
return Math.sqrt(-2*Math.log(u))*Math.cos(v);
}
```

Et comme en JavaScript, on a besoin d'accéder au tableur, à la feuille courante et à ses cellules :

```
 oDoc=XSCRIPTCONTEXT.getDocument();
control=oDoc.getCurrentController();
classeur=UnoRuntime.queryInterface(XSpreadsheetView.class, control);
feuille=classeur.getActiveSheet();
```

Enfin, la boucle remplissant la colonne A ressemble à la version JavaScript, à ceci près que son indice *i* doit être déclaré entier :

```
 for(int i=1;i<=20;i++){
cellule=feuille.getCellByPosition(0,i);
cellule.setValue(normal());
}
```

Une fois tout ce code entré, il suffit de cliquer sur *Save* puis sur *Run* pour avoir une feuille du même style que celle de l'onglet *JavaScript*.

Ceci dit, cet outil n'apporte rien de vraiment nouveau pour l'algorithmique au lycée :

- À l'instar (l'inStarOffice ?) de JavaScript, il faut entrer beaucoup de lignes peu compréhensibles avant même de pouvoir "entrer l'algorithme".
- Contrairement à Python, BeanShell a un éditeur dans Open Office, mais l'éditeur en question est assez basique (mais il est extrêmement léger : Moins de 300 kilooctets !). En particulier pas de mode pas-à-pas...
- Tout comme JavaScript et Python, l'interactivité n'est pas optimale, une fois le tableur rempli, il n'est plus dynamique. On peut y remédier en remplaçant les *cellule.setValue()* par des *cellule.setFormula()* mais ça complique encore la manip, la construction de formules
  - se faisant par concaténation de chaînes de caractères, et
  - nécessitant une connaissance du tableur que des élèves en cours d'apprentissage de l'algorithmique n'ont pas vraiment le temps d'acquérir au Lycée...

En résumé, Open Office possède plusieurs langages de programmation, ce qui permet d'envisager son utilisation pour l'enseignement de l'algorithmique, mais outre Basic, ces langages sont peu interactifs, et on peut souhaiter l'existence d'un langage qui interagit réellement avec le tableur, et soit proche de langages *web 2.0* tels *Java* et *JavaScript*. Pour un tel outil, voir l'onglet suivant :

++++ CMath

CmathOOoCAS, décrite par son auteur dans ce magistral article, permet de faire non seulement du calcul formel dans le tableur, mais également de l'algorithmique puisqu'il est lui-même muni de 5 langages de programmation :

1. Le langage de la Ti89 ;
2. Le langage MuPAD ;
3. Le langage Maple ;
4. Le langage traditionnel d'Xcas ;
5. Et une version française de ce dernier, proche du pseudocode.

Pour éviter une redite avec le mode d'emploi de CmathOOoCAS, on ne va pas reparler du temps de vol de la suite de Collatz, mais plutôt reprendre un problème classique à l'envers :

On *définit* une fonction *Heron* de la façon suivante : Pour tout *x* non nul, *Heron(x)* est la limite de la suite u_n definie par $u_0=1$ et $u_{n+1}=\frac{u_n+\frac{x}{u_n}}{2}$ [2].

Comme la limite d'une suite est bien approchée par un terme de rang suffisamment élevé, on va donc calculer *Heron(x)* par une boucle itérative, avec l'algorithme suivant [3] :

```
Entrer x
 Mettre 1 dans u
 Mettre x dans v
 Tant que u est suffisamment éloigné de v faire
 calculer la moyenne arithmétique de u et v
 la stocker dans m
 mettre m dans u et le quotient de x par m dans v
 fin tant que
 sortir u
```

Pour bénéficier de la fonction *Heron* dans le tableur, on peut utiliser le langage *Basic* mais aussi l'un au choix des langages de CmathOOoCAS. Pour cela on clique sur l'icône *CAS* et on entre le programme :

version algorithmique

Pour avoir accès à ce langage proche du pseudocode, il faut choisir le langage *Xcas* dans les préférences de CmathOOoCAS.

```
fonction Heron(x)
    local u,v,m;
    u:=1;
    v:=x;
    tantque abs(u-v)>1e-8 faire
        m:=(u+v)/2;
        u:=m;
        v:=x/u;
    ftantque;
    retourne u;
ffonction
```

version Xcas

Ce langage est accessible lui aussi en sélectionnant *Xcas* dans les préférences de CmathOOoCAS :

```
Heron(x):={
  local u,v,m;
  u:=1;
  v:=x;
  while (abs(u-v)>1e-8){
   m:=(u+v)/2;
   u:=m;
   v:=x/u;
  }
  return(m);
};
```

version Calculatrice TI

Pour avoir accès au langage des calculatrices Texas Instruments, il faut choisir *Ti 89/92* dans le préférences de CmathOOoCAS, et taper un double-point au début de chaque ligne :

```
:Heron(x)
:Func
:u:=1
:v:=x
:While(abs(u-v)>1e-8)
:m:=(u+v)/2
:u:=m
:v:=x/u
:EndWhile
:return u
:EndFunc
```

version Maple

On peut écrire dans le langage de programmation de Maple en sélectionnant *Maple* parmi les langages dans les préférences de CmathOOoCAS.

```
Heron:=proc(x)
  u:=1;
  v:=x;
  while(abs(u-v)>1e-8) do
    m:=(u+v)/2;
    u:=m;
    v:=x/u;
  od
  RETURN(u);
end
```

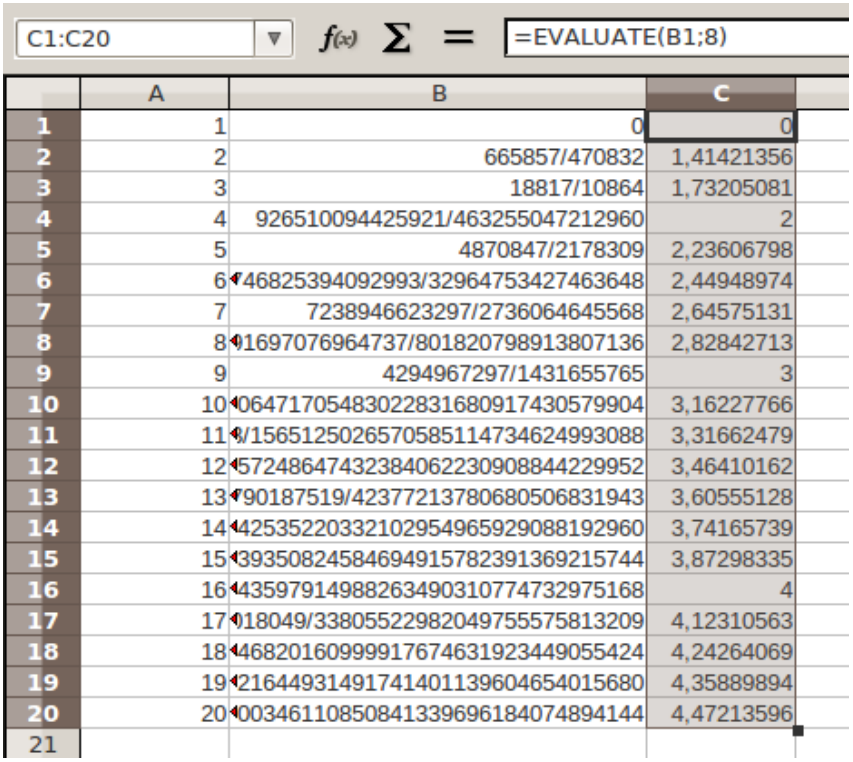
version MuPad

Une fois qu'on a sélectionné MuPad dans le menu des préférences de CmathOOoCAS, on peut programmer dans ce langage proche de *Pascal* :

```
Heron:=proc(x)
local u, v, m;
begin
  u:=1;
  v:=x;
  while abs(u-v)>1e-8 do
    m:=(u+v)/2;
    u:=m;
    v:=x/u;
  end_while;
  u;
end_proc:
```

Une fois qu'on a entré le programme *Heron* dans la fenêtre de programmation, il suffit de le "compiler" en cliquant sur le bouton de compilation, puis de cliquer sur "j'ai terminé" pour avoir une fonction *Heron* qui n'existait pas auparavant dans le tableur.

Alors, si on entre dans la colonne *A* les nombres entiers naturels non nuls, et qu'on entre dans la cellule *B1* la formule `=Heron(A1)`, il suffit de recopier celle-ci vers le bas pour avoir les approximations des racines carrées des entiers successifs :



	A	B	C
1	1	0	0
2	2	665857/470832	1,41421356
3	3	18817/10864	1,73205081
4	4	926510094425921/463255047212960	2
5	5	4870847/2178309	2,23606798
6	6	46825394092993/32964753427463648	2,44948974
7	7	7238946623297/2736064645568	2,64575131
8	8	1697076964737/801820798913807136	2,82842713
9	9	4294967297/1431655765	3
10	10	0647170548302283168091743057990 4	3,16227766
11	11	1565125026570585114734624993088	3,31662479
12	12	5724864743238406223090884422995 2	3,46410162
13	13	90187519/42377213780680506831943	3,60555128
14	14	4253522033210295496592908819296 0	3,74165739
15	15	3935082458469491578239136921574 4	3,87298335
16	16	4359791498826349031077473297516 8	4
17	17	18049/3380552298204975557581320 9	4,12310563
18	18	4682016099991767463192344905542 4	4,24264069
19	19	2164493149174140113960465401568 0	4,35889894
20	20	0034611085084133969618407489414 4	4,47213596
21			

Seulement on est dans du calcul formel, et les résultats donnés sont des fractions, données sous forme exacte ! Pour les racines de 4 et 9, on voit des intéressantes approximations d'entiers par des fractions ! Pour retrouver les racines carrées que l'on attendait, on est obligé de calculer des valeurs approchées de ces fractions, ce qu'on a fait ci-dessus par `=evaluer(B1;8)` dans *C1*, recopié vers le bas.

L'utilisation d'algorithmes dans le calcul formel s'illustre très bien avec l'algorithme de Fibonacci, décrit dans l'article de Wikipedia intitulé "Fraction égyptienne", et dont la réalisation par tableur (sans boucle) est décrite dans l'article suscité .

Le document est téléchargeable ci-dessous, mais pour l'ouvrir il faut qu'Open Office soit installé (on n'en meurt pas !) et que dans le tableur d'Open Office, soit installée la barre d'outils CmathOOoCAS, par exemple en ouvrant le gestionnaire d'extensions, et en y choisissant "ajouter une extension". Normalement il suffit d'accepter la licence GPL pour que l'installation se fasse. Et on peut alors ouvrir le fichier ci-dessous et en créer d'autres, qui ne sauront être que géniaux et innovants !

Fractions Egyptiennes

Conclusion :

Dans cet article, on a appris que

$$\frac{926510094425921}{463255047212960}\simeq 2$$

ainsi, espérons-le, que quelques autres informations...

---

[1] CASenPoche offre aussi du calcul formel, mais je n'ai pas eu l'occasion de l'essayer, et l'outil ne semble plus actuellement développé ; CmathOOoCAS est assez orienté vers l'algorithmique, ce qui ne semble pas avoir été le cas avec CASenPoche. Ce qui n'enlève rien à l'intérêt que peut présenter un tableur en ligne comme CASenPoche, même si dans ce cas, j'ai personnellement fait le choix de GeoGebra...

[2] Ce n'est pas la fonction *racine carrée*, les deux fonctions n'ayant pas le même ensemble de définition...

[3] L'usage de la variable *m* est superflu mais rend la description de l'algorithme un peu plus claire.

