Tutoriel SuperCollider

Chapitre 3

Par Celeste Hutchins

2005

www.celesteh.com

Licence Creative Commons : Attribution uniquement

Tableaux

Vous vous souvenez peut-être qu'un **tableau** est une liste indexée. Nous pouvons créer un tableau en plaçant zéro ou plusieurs expressions délimitées par une virgule entre crochets. Par exemple, voici quelques tableaux :

```
[1]
[x, y, z]
["dans", "le", "la",
"la"] []
```

Il y a souvent plus d'une façon de faire quelque chose dans SuperCollider, comme dans la vie. Nous pouvons également créer un tableau en utilisant un constructeur. Le constructeur prend un argument, la taille du tableau.

```
a = Array.new(size) ;
```

Les tableaux peuvent contenir n'importe quel type d'objet, y compris des nombres, des variables, des chaînes de caractères ou rien du tout. Ils peuvent même mélanger les types :

```
[3, "poules françaises", 2 "tourterelles", 1 "perdrix"].
```

Les tableaux peuvent même contenir d'autres tableaux : `[[1, 2, 3], [4, 5, 6]]`

Vous pouvez également placer une **expression** dans un tableau. L'interprète l'évalue et stocke les résultats. Ainsi, `[ 3 - 1, 4 + 3, 2 * 6]` est également un tableau valide, stocké sous la forme `[2, 7, 12]`. `[x.foo(2), x.bar(3)]` est également un tableau. Il transmet ces messages aux objets et place le résultat dans le tableau. Les virgules ayant une **priorité** extrêmement faible**,** elles sont évaluées en dernier.

Une variable peut faire partie d'une expression qui est placée dans un tableau :

```
(
  var foo, bar ;
  ...
  [foo + 1 , bar] ;
)
```

Dans ce dernier cas, il conserve la valeur des expressions (y compris les variables) au moment où elles ont été intégrées au tableau. Par exemple :

```
(
  var foo, arr ;
  foo = 2 ;
  arr = [foo] ;
  arr.postln ;
  foo. postln ;

  " ".postln ;

  foo = foo + 1
  ; arr.postln ;
  foo. postln ;
 )
```

Sorties :

```
[ 2 ]
2

[ 2 ]
3
3
```

C'est *presque ce à* quoi nous nous attendions. D'où vient le deuxième 3 en bas ? L'interpréteur de SuperCollider imprime une valeur de retour pour chaque bloc de code qu'il exécute. `bar` était le dernier objet du bloc de code, donc `bar` est retourné. la valeur de `bar` est 3.

Les variables peuvent continuer à changer, mais le tableau conserve la valeur qui a été introduite, comme un instantané du moment où l'expression a été évaluée.

Que faire si nous déclarons un tableau à cinq éléments et que nous voulons y ajouter quelque chose ? Nous utilisons le message Array.add.

```
(
	var arr, new_arr ;
	arr = ["Mary", "had", "a", "little"]
	; new_arr = arr.add("lamb") ;
	arr.postln ;
	nouveau_arr.postln ;
)
```

Sorties :

```
[ Marie, avait, un, peu ]
[ Marie, avait, un, petit, agneau ]
```

Les tableaux ne peuvent pas augmenter en taille une fois qu'ils ont été créés. Ainsi, comme indiqué dans le fichier d'aide, "la méthode 'add' peut renvoyer ou non le même objet Array. Elle ajoute l'argument au récepteur s'il y a de la place, sinon elle renvoie un nouvel objet Array avec l'argument ajouté". Par conséquent, lorsque vous ajoutez quelque chose à un tableau, vous devez affecter le résultat à une variable. `arr ne` change pas dans l'exemple car il est déjà plein.

Il existe d'autres messages que vous pouvez envoyer aux tableaux, qui sont détaillés dans le fichier d'aide du tableau et dans le fichier d'aide de sa superclasse ArrayedCollection. Deux de mes messages préférés sont **scramble** et **choose.**

Le fichier d'aide indique que scramble "renvoie un nouveau tableau dont les éléments ont été brouillés. Le récepteur est inchangé."

Les résultats de scramble sont différents à chaque fois que vous l'exécutez, parce qu'il brouille dans un ordre aléatoire. Lorsqu'il est dit "le récepteur est inchangé", cela signifie que si nous voulons sauvegarder le tableau brouillé, nous devons assigner cette sortie à une nouvelle variable. Le **récepteur** est l'objet qui **reçoit** le message "scramble". Dans cet exemple, le récepteur est `arr`, qui contient `[1, 2, 3, 4, 5, 6]`.

```
(
  var arr, brouillé ;
  arr = [1, 2, 3, 4, 5, 6] ;
  scrambled = arr.scramble
  ; arr.postln ;
  scrambled.postln ;
)
```

Sorties :

```
[ 1, 2, 3, 4, 5, 6 ]
[ 4, 1, 2, 3, 6, 5 ]
```

Bien sûr, le deuxième tableau est différent à chaque fois. Et `arr` est inchangé.

Choose est similaire. Il choisit un élément aléatoire du tableau et le transmet. Le récepteur reste inchangé.

```
[1, 2, 3].choisir.postln ;
```

Sorties :

```
2
```

Ou 1 ou 3, ce qui peut changer à chaque fois que vous l'exécutez.

Les tableaux sont des listes, mais ce ne sont pas de simples listes. Ce sont des listes **indexées.** Vous pouvez demander quelle est la valeur d'un élément d'un tableau **à** une position particulière.

```
(
	var arr ;
	arr = ["Mary", "had", "a", "little"] ;

	arr.at(1).postln ;
	arr.at(3).postln ;
)
```

Sorties :

```
n'avai
t que
peu
```

Dans SuperCollider, les index des tableaux commencent par 0. Dans l'exemple ci-dessus, `arr.at(0)` est

`"Marie"`.

Les tableaux comprennent également le message **do**, mais le traitent un peu différemment des entiers.

```
(
	[3, 4, 5].do ({ arg item, index ;
```

```
            ("index : " ++ index ++ " item : " ++ item).postln ;
        }) ;
)
```

Sorties :

```
l'inde  0 articl 3
  x :       e :
l'inde  1 articl 4
  x :       e :
l'inde  2 articl 5
  x :       e :
```

Dans cet exemple, nous appelons nos arguments "`item`" et "`index`", mais ils peuvent porter n'importe quel nom. Le nom que nous leur donnons n'a pas d'importance. Le premier obtient toujours la valeur de l'élément du tableau sur lequel la boucle se trouve et le second obtient toujours l' index.

Le ++ signifie **concaténer**, soit dit en passant. Il sert à ajouter quelque chose à la fin d'une chaîne de caractères. Par exemple, le ++ est utilisé pour ajouter quelque chose à la fin d'une chaîne :

`"foo " ++ 3` renvoie la chaîne `"foo 3"`

Dans le dernier chapitre, j'ai parlé de Nicole, l'ancienne étudiante diplômée qui travaille dans une start-up de SuperCollider. Depuis, son patron lui a confié une nouvelle mission. Elle doit écrire une fonction qui prend comme arguments un tableau de rapports d'accord, une fréquence de base et une valeur de désaccord. Elle doit calculer les hauteurs finales en multipliant d'abord la fréquence de base par le rapport, puis en ajoutant la valeur de désaccord au résultat. Il doit ensuite les imprimer dans le format suivant :

rapport d'accord : <ratio>, hauteur : <pitch>

Après s'être familiarisée avec les tableaux, notre héroïne fait quelques recherches sur les ratios d'accord et trouve un tableau de ratios qu'elle utilisera

pour tester sa fonction. Il se présente comme suit

comme : `[ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5 ]`

C'est-à-dire 1 divisé par 1, 3 divisé par 2, quatre divisé par 3, etc. N'oubliez pas que la préséance signifie que l'interprète évalue les choses dans un ordre particulier. Il regarde / avant de regarder les virgules. Il voit donc un tas de / et commence à diviser. Ensuite, il regarde les virgules et les traite comme un tableau. L'interprète stocke ce tableau sous la forme suivante :

```
[ 1, 1.5, 1.3333333333333, 1.125, 1.7777777777778, 1.25, 1.6 ]
```

La fonction de Nicole fait défiler les rapports, en prenant chacun d'entre eux et en le multipliant par la fréquence de base et en ajoutant la quantité de désaccord. Rappelez-vous que SuperCollider est comme une calculatrice bon marché et que +, -, * et / ont tous la même priorité. Les mathématiques sont évaluées de gauche à droite. Ainsi, `ratio * baseFreq + désaccord` n'est pas équivalent à `désaccord + ratio * baseFreq`, comme ce serait le cas en algèbre. Cependant, heureusement, elle peut utiliser des parenthèses comme en algèbre.

Elle pourrait écrire son **expression** comme `(ratio * baseFreq) + detune` ou `(detune + (ratio * baseFreq))` ou de bien d'autres façons. Même si elle pouvait obtenir la bonne réponse sans utiliser de parenthèses, c'est une bonne pratique de programmation que de les utiliser.

Nicole a une formule et elle a une Array. Elle a juste besoin d'un moyen de le parcourir. Heureusement, elle sait que la méthode "do" existe aussi pour les tableaux.

Elle écrit son code comme suit :

```
(

    var func, arr ;

    func = { arg ratio_arr, baseFreq = 440, detune = 10 ;
```

```
        var pitch ;

        ratio_arr.do({arg ratio, index ;
            hauteur = (ratio * baseFreq) + detune ;
            ("Ratio : " ++ ratio ++ " Pitch : " ++ pitch).postln ;
        }) ;
    } ;

    arr = [ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5] ;

    func.value(arr, 440, 10) ;
)
```

Quelles sont les sorties :

```
Rapport : 1 Pas : 450
Rapport : 1,5 Pas : 670
Rapport : 1.3333333333333 Pas : 596.66666666667
Rapport : 1,125 Pas : 505
Rapport : 1.7777777777778 Pas : 792.22222222222
Rapport : 1,25 Pas : 560
Rapport : 1,6 Pas : 714
```

Enveloppes

La prochaine tâche logique de Nicole est de jouer ces hauteurs plutôt que de les imprimer. Dans le chapitre précédent, nous avons écrit une fonction pour jouer les harmoniques, mais elle les jouait toutes en même temps. Nous avons besoin d'un moyen pour indiquer aux notes de se terminer et de passer d'une note à l'autre.

> "Tout comme les êtres humains, les sons naissent, atteignent leur apogée et meurent.

la vie d'un son, de la création à l'évanescence, ne dure que quelques secondes" Yamaha GX-1 Guide

Nous contrôlons ce processus à l'aide d'un UGen appelé **enveloppe**.

```
(

    var syn, sound ;

    syn = SynthDef.new("example3", {arg out = 0, freq = 440, amp = 0.2,
                                dur = 1 ;
        var sin, env_gen, env ;

        env = Env.triangle(dur, amp) ;
        env_gen = EnvGen.kr(env) ;
        sin = SinOsc.ar(freq, mul : env_gen) ;
        Out.ar(out, sin) ;
    })
    ;

    syn.load(s) ;
)

Synth.new("exemple3") ;
```

Il y a plusieurs choses qui rendent ce SynthDef différent. La première est qu'il utilise des variables. Les variables sont un bon moyen de rendre les SynthDefs plus lisibles. Plus il est facile à lire, plus il est facile à comprendre, à corriger et à modifier.

L'autre changement évident est l'ajout d'une enveloppe. Les enveloppes sont composées de deux parties. La première est **Env**. La classe Env vous permet de décrire la forme de l'enveloppe. Dans ce cas, nous utilisons une enveloppe de **durée fixe** ayant la forme d'un

triangle. Lorsque nous utilisons une enveloppe à durée fixe, nous connaissons la longueur de l'enveloppe lorsque nous créons une instance du Synth. Il existe d'autres enveloppes dont nous ne connaîtrons pas l'enveloppe jusqu'à ce que nous décidions de terminer la note, par exemple, en réponse à une touche vers le haut sur un clavier.

`triangle` est un constructeur. Il crée une nouvelle instance d'un Env. Les arguments qu'il prend sont la durée et l'amplitude maximale.

L'autre partie de l'enveloppe est l'**EnvGen**. Comme l'EnvGen est une classe, `kr` doit être un constructeur. Les deux constructeurs UGen les plus courants sont `ar` et `kr`. Vous vous souvenez peut-être que ar signifie "taux audio" et kr "taux de contrôle". Comme l'EnvGen ne crée pas de son, mais contrôle l'amplitude d'un son, nous voulons le taux de contrôle. Les signaux de taux de contrôle changent moins souvent que les signaux de taux audio, et l'utilisation de kr réduit donc la charge de notre ordinateur et rend notre SynthDef plus efficace.

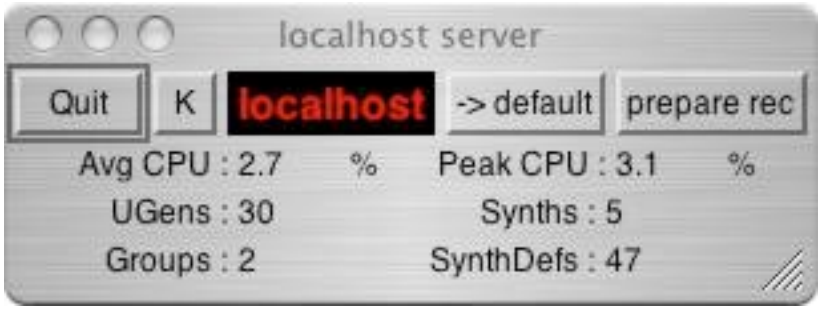
EnvGen.kr génère l'enveloppe sur la base des spécifications de son premier argument, une instance de Env. Ainsi, avec Env, nous définissons une enveloppe. Nous transmettons cette définition à un Envelop Generator, qui reproduit l'enveloppe que nous avons définie. Nous utilisons cette enveloppe pour contrôler l'amplitude d'un SinOsc. Nous envoyons la sortie du SinOsc vers un bus de sortie. Lorsque l'enveloppe est terminée, son amplitude est revenue à zéro, de sorte que le SinOsc n'est plus audible.

Dans nos exemples précédents, le Synth ne s'arrêtait jamais. La façon d'arrêter le son était d'arrêter l'exécution avec apple-. Avec notre nouveau SynthDef, le son continue pendant la durée exacte et se termine sans qu'il soit possible de le redémarrer. Après avoir créé un Synth et l'avoir joué, il n'y a rien qui puisse être fait avec lui.

Essayez de lancer `Synth.new("example3") ;` plusieurs fois. Si vous regardez la fenêtre du serveur local, vous verrez le nombre d'UGens et de Synths augmenter

à chaque fois.

vous exécutez Synth.new. Le CPU moyen et le CPU de pointe peuvent également augmenter.



Chaque fois que nous créons une nouvelle instance de Synth, nous utilisons davantage de ressources système. Nous finirons par manquer de mémoire ou le CPU dépassera les 100%. Nous pouvons effacer les Synths en appuyant sur pomme-, mais cela limiterait encore le nombre de sons que nous pourrions jouer d'affilée sans arrêter l'exécution.

Nous avons besoin d'un moyen automatique de retirer nos synthés du serveur et de **désallouer** leurs ressources. Cela signifie que d'autres synthés pourraient utiliser les ressources qu'ils occupaient.

Heureusement, les EnvGen peuvent à la fois faire taire un Synth pour toujours et le mettre au repos pour qu'il soit désalloué. EnvGen prend un argument appelé "doneAction". L'ajout d'une `doneAction` à votre EnvGen ressemblerait à ceci :

```
env_gen = EnvGen.kr(env, doneAction : 2) ;
```

Le fichier d'aide EnvGen décrit doneActions. Vous devriez consulter le fichier d'aide de EnvGen pour connaître la signification des différentes valeurs. 2 signifie "enlever le synthé et le désallouer".

Essayez de changer en ajoutant une doneAction à la SynthDef et en exécutant `Synth.new("example3") ;` plusieurs fois. Remarquez que maintenant les Synths ne s'accumulent plus. Maintenant, lorsqu'un Synth arrête de produire des sons, il a terminé sa vie utile et est donc supprimé du serveur à cause de l'action doneAction : 2. Cependant,

Notez que l'action doneAction ne supprime que les instances du Synth. Le SynthDef persiste et peut être utilisé autant de fois que nous le souhaitons.

Les enveloppes ne doivent pas seulement contrôler l'amplitude, mais peuvent contrôler (presque) n'importe quel aspect de n'importe quel UGen. Vous ne pouvez pas régler le bus sur Out.ar avec une enveloppe, mais vous pouvez contrôler presque n'importe quoi d'autre.

Routines

Maintenant que nous savons comment créer des synthés qui peuvent s'arrêter de jouer, nous avons juste besoin d'un moyen de chronométrer leur création. Une routine est un objet très polyvalent qui peut être utilisé de nombreuses façons. L'une d'entre elles consiste à ajouter du temps à vos programmes.

```
(

     r = Routine.new ({

          5.do({ arg index ;
               index.postln ;
               1.wait ;
          }) ;
     }) ;

     r.play ;
)
```

Cet exemple s'imprime :

```
0
1
2
```

```
3
4
```

Avec une pause d'une seconde entre chaque numéro. Essayez de l'exécuter.

La classe SimpleNumber, dont on se souvient qu'elle est la superclasse de Float et Integer, prend en charge un message appelé `wait`. Ce message ne fonctionne que dans le contexte d'une routine. Si nous essayons d'utiliser `wait` en dehors d'une routine :

```
(

     f = {

          5.do({ arg index ;
               index.postln ;
               1.wait ;
          }) ;
     } ;

     f.play ;
)
```

Nous obtenons une erreur :

```
• ERREUR : yield a été appelé en dehors d'une
routine. ERREUR : La primitive '_RoutineYield'
a échoué.
```

Les routines sont un moyen puissant d'ajouter du temps à votre programme. Routine.new prend une fonction en argument. Elle exécute cette fonction lorsque nous lui envoyons le message **play.** D'où `r.play ;` dans l'exemple de la routine. Lorsque nous jouons une routine, elle se met en pause pendant une durée de SimpleNumber.wait. Si nous codons `5.wait`, il se mettra en pause

pendant cinq secondes. Si nous avons `0.5.wait`, la pause durera une demi-seconde.

Essayons de prendre l'affectation d'accord de Nicole et de lui faire jouer des sons au lieu de simplement imprimer des données. Nous ne pouvons pas passer d'arguments à une routine, mais heureusement, à cause de la **portée** et des fonctions qui **renvoient des** choses, il existe un moyen astucieux de contourner le problème :

```
(

  var func, rout ;

  func = { arg ratio_arr, baseFreq = 440, detune = 10 ;

     Routine.new({

          ratio_arr.postln ;
          baseFreq.postln ;
          detune.postln ;
     }) ;

  } ;

  rout = func.value([1/1, 3/2, 2/1]) ;
  rout.play ;
)
```

Tout d'abord, nous déclarons une fonction qui prend trois arguments. Ensuite, à l'intérieur de la fonction, nous déclarons une routine. Comme la routine se trouve à l'intérieur du bloc de code de la fonction, elle se trouve dans la portée des arguments. Et parce que la routine est la dernière (et la seule, à part les arguments) c h o s e à l'intérieur de la fonction, elle est renvoyée lorsque la fonction est évaluée. Ainsi, `rout` récupère la routine renvoyée par la fonction.

Ensuite, nous appelons `rout.play ;` pour exécuter la routine.

N'oubliez pas que les fonctions **renvoient** leur dernière valeur. Et n'oubliez pas que la **portée** signifie que les blocs de code internes peuvent voir les variables des blocs de code externes, mais pas l'inverse. Ainsi, le Routine peut voir les args. Et il n'y a rien après Routine dans la fonction, c'est donc la dernière valeur, elle est donc renvoyée lorsque nous envoyons un message de valeur à la fonction. Ensuite, nous envoyons un message de lecture à la routine renvoyée. Nous pourrions abréger les deux dernières lignes de notre programme en ne conservant qu'une seule ligne :

```
func.value([1/1, 3/2, 2/1]).play ;
```

L'interprète, en lisant de droite à gauche, passera un message de valeur à la fonction, avec un argument de `[1/1, 3/2, 2/1]`, un tableau. Cela renvoie une Routine, à qui l'on passe ensuite un message de jeu.

Nous pourrions utiliser une routine pour modifier le programme de Nicole afin qu'au lieu d'imprimer les ratios, il les joue en séquence en utilisant notre SynthDef "example2".

```
(

	var func, arr ;

	func = { arg ratio_arr, baseFreq = 440, detune = 10 ;
		var pitch ;

		Routine.new({
			ratio_arr.do({arg ratio, index ;
				hauteur = (ratio * baseFreq) + detune ;
				//("Ratio : " ++ ratio ++ " Pitch : " ++
				pitch).postln ; Synth.new("example3", [\freq, pitch,
				\dur, 1]) ; 1.wait ;
```

```
            }) ;
        }) ;
    } ;

    arr = [ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5] ;

    func.value(arr, 440, 10).play ;
)
```

Nous pouvons changer cela pour qu'il joue les hauteurs dans un ordre aléatoire, en envoyant un message de brouillage à notre tableau de ratio, soit en changeant l'appel de notre fonction à :

```
    func.value(arr.scramble, 440, 10).play ;
```

Ou en modifiant notre boucle "do" :

```
    ratio_arr.scramble.do({ arg ratio, index ;
```

Cela ressemble de plus en plus à la musique de ! Essayons de le répéter plusieurs f o i s , avec un argument spécifiant le nombre de fois.

```
(

    var func, arr ;

    func = { arg ratio_arr, baseFreq = 440, detune = 10, repeats= 1 ;
        var pitch ;

        Routine.new({
            repeats.do ({
                ratio_arr.scramble.do({ arg ratio, index ;
```

```
                        hauteur = (ratio * baseFreq) + detune ;
                        //("Ratio :  " ++ ratio ++ " Hauteur :  " ++
pitch).postln ;
                        Synth.new("example3", [\freq, pitch, \dur, 1])
                        ; 1.wait ;
                    })
                    ;
                }) ;
            }) ;
        } ;

        arr = [ 1/1, 3/2, 4/3, 9/8, 16/9, 5/4, 8/5] ;

        func.value(arr, 440, 10, 2).play ;
)
```

Et si nous voulions jouer une centaine de notes ? Il y a souvent plus d'une façon de faire quelque chose.

Nous avons pu déterminer la taille de notre Array :

```
var arr_size ;
arr_size = ratio_arr.size ;
```

Il faut ensuite diviser ce nombre en cent pour obtenir un nombre pour une boucle SimpleNumber.do. Cependant, il se peut que le tableau ne soit pas divisé uniformément en cent.

Ou nous pourrions simplement compter jusqu'à 100.

```
100.do({arg count ;

    count = count % arr_size ;
```

```
    pitch = ratio_arr.at(count) ;
    hauteur = (hauteur * baseFreq) + désaccord ;

    . . .
}) ;
```

Rappelez-vous que le module (%) nous donne un reste. Cela signifie que si le compte devient supérieur au nombre d'éléments du tableau, l'utilisation d'un module le ramènera à zéro lorsqu'il dépassera la taille du tableau. Il existe également un message que vous pouvez envoyer aux tableaux qui fait cela pour vous :

```
arr.wrapAt(index) == arr.at(index % arr.size)
```

Problèmes

1. Essayez les différentes enveloppes à durée fixe et essayez-les avec différents oscillateurs. Essayez en particulier l'enveloppe Env.perc avec différents générateurs de bruit.

2. Créer un programme qui joue des notes à partir d'un tableau d'accord. Modifiez la durée des notes en utilisant la valeur \dur dans Synth.new. Essayez également de modifier la longueur des temps d'attente. Créez un rythme en utilisant des tableaux pour les durées et les attentes.

3. Vous pouvez naviguer dans des tableaux de différentes longueurs en utilisant des variables comme index.

```
(

  var arr, arr2, arr2_index ;

  arr = [1, 2, 3] ;
```

```
  arr2 = [3, 4, 5, 6] ;
  arr2_index = 0 ;

  2.do({
       arr.do({arg item ;
            ("arr " ++ item).postln ;
            ("\t arr2 " ++ arr2.wrapAt(arr2_index)).postln ;
            arr2_index = arr2_index + 1 ;
       }) ;
  }) ;
)
```

("\t" signifie onglet et rend l'impression plus agréable).

Utilisez cette idée pour créer des boucles de durée, de hauteur et d'attente de différentes longueurs.

4. Au lieu de calculer le désaccord dans votre routine, faites-le dans le SynthDef en utilisant une enveloppe de durée fixe. Créez un argument pour votre SynthDef pour la quantité de désaccord de crête.

Projet

Écrire une pièce d'une ou deux minutes en utilisant les routines.