Optimisation en Visual Basic Application

Adresses utiles :

- http://fordom.free.fr/tuto/OPTIMISATION.htm
- http://xcell05.free.fr/pages/prog/accvba.htm
- http://www.info-3000.com/vbvba/conseiloptimisation.php
- http://www.aivosto.com/vbtips/stringopt.html
- http://www.aivosto.com/vbtips/vbtips.html
- http://www.aivosto.com/vbtips/vbtips.html
- http://www.aivosto.com/vbtips/vbtips.html

Avant propos

Voici un petit recueil d'idées pour réaliser une optimisation des programmes en VBA. IL résulte de mes propres expériences et lectures. D'une manière générale, il n'est pas évident de savoir si telle ou telle forme optimise un algorithme. IL convient par conséquent de tester soi-même, différente forme pour en garder que les meilleurs… Ce tutoriel hiérarchise un peu ces différentes formes qu'on peut imaginer. J'indique les performances approximatives des formes non optimisées par rapport à la forme optimisée testée à "vide", et donc non compilée. La mesure du temps étant réalisée simplement par la fonction "Timer" car seul le classement importe. IL est certain, que cela peut varier beaucoup selon le contexte (appel en cache) ou selon le mode d'exécution. Mais, le classement lui varie assez peu. La notation (+5*) signifie 5 fois plus lent. JE laisse aussi un petit listing, permettant à chacun d'avoir une base pour élaborer ses propres procédures de test. Dans une certaine mesure, ces optimisations sont applicables pour un programme VB compilé.

Généralités

Par optimisation, je considère la vitesse d'exécution uniquement. Néanmoins, il me semble assez logique, qu'on puisse interpréter l'optimisation de différentes façons. Par exemple, on peut considérer l'optimisation comme l'algorithme le plus court, le plus simple, le plus clair afin d'en faciliter la relecture ultérieure, ou encore le moins gourmand en ressource matériel, etc… Ces aspects sont tout aussi louables que l'optimisation de la vitesse, et se cumulent souvent… EN réalité, le programmeur doit souvent faire des compromis entre ces différents aspects… Pour optimiser une programmation existante, il est assez évident qu'on peut opérer des priorités, dont la première consiste à optimiser les boucles les plus longues, puis en allant vers les répétitions de codes les moins courantes. Un autre principe évident, c'est de réutiliser au maximum une expression calculée stockée dans une variable, et d'utiliser le moins de variables possibles. Mais, la meilleure optimisation reste d'un autre ordre non évoqué ici, consistant à trouver le meilleur algorithme, qui dépend que de votre réflexion !

Optimisation des variables

Déclarations des variables

La déclaration du type de variable est une première optimisation extrêmement classique et indispensable. Afin de n'oublier aucune déclaration, utiliser en tête de module, ou de Form etc... la fonction :

```
Option Explicit
```

Succinctement, à retenir pour choisir le type :

- nombres entiers : préférer le type Long
- nombres décimaux : type Double
- variables de chaînes : type String

Bien sur, les déclarations dépendent aussi des besoins.

Le type Variant (autre que dans les appels de fonctions) n'est jamais indispensable, et est à éviter.

<note important>Une remarque de syntaxe :

A tort, certains pensent que :

```
Dim a, b, c As Long
```

déclare les 3 variables en Long. En réalité, seul *c* est ici en Long ; *a* et *b* sont déclarées en Variant par défaut.

La syntaxe correcte est :

```
Dim a As Long, b As Long, c As Long
```

</note>

Utilisation des tableaux (Dim)

Quelques évidences.

L’optimisation avec les tableaux consiste d’abord à déclarer son type.

Puis, autant que possible, il faut trouver une structure et déterminer le nombre d’éléments, quitte à définir un nombre en excès, pour ne jamais manipuler une nouvelle fois la déclaration d’un tableau.

Par exemple, éviter de réutiliser Redim (Preserve).

Optimisation sur les nombres

Longueur d'un nombre entier positif ou nul

Pour connaître le nombre de chiffres

```
Fix(Log(nb + 0.11) / Log(10)) + 1
```

0.11 est utile dans le cas où nb=0, ainsi la formule renvoie 1. Si nb ne peut pas être nul, retirer cette constante.

Pour un nombre entier négatif,

A partir de la forme précédente, il faut adapter la formule précédente en ajoutant +1 pour tenir compte du signe négatif, ou +1 pour la virgule.

```
Fix(Log(Abs(nb) + 0.11) / Log(10)) + 2
```

Parité d’un nombre

- Pour tester si un nombre est pair :

```
If (nb And 1) = 0 Then ...
```

- Pour tester si un nombre est impair :

```
If (nb And 1) Then ...
```

<note tip>Entre ces deux formes, cette dernière est plus rapide car il n’y a pas de test d’égalité (=0) qui pernd du temps. </note>

Calculs des puissances entières

Curieuse fonction que “ ^ ” qui s’effectue en un temps quasi-constant quelle que soit la puissance, mais reste moins rapide que la multiplication successive jusqu’à puissance 9 environ.

```
nb = lg * lg * ... * lg
```

Calcul de la racine carrée

```
lg = Sqr(nb)
```

<note tip>Comme pour les puissances, on peut cumuler les racines carrés optimisées jusqu’à 4. Ainsi lg = Sqr(Sqr(Sqr(Sqr(nb)))) est la dernière expression plus rapide que lg = nb ^ 0.0625.</note>

Division entière

Si les nombres peuvent être déclarés en type Long sans dépassement de capacité,

```
nb = nb1 \ nb2
```

Pré-calcul

Il faut sortir d'une boucle tous les calculs d’expression constante.

```
Dim nb As Long, lg As Double, z As Double
z = Sqr(2)
lg = nb * z
```

Maximum et Minimum de 2 nombres

```
if a<b then max=b else max=b
if a<b then min=a else min=b
```

Optimisation des tests

Vérification d’une condition avec renvoi Booléan

```
a=(condition)
```

Ici condition pourrait être constituée par un expression comme b=c, soit a=(b=c). L’interprétation est facile, si b=c alors VB remplace en interne cette condition par la valeur vraie (True), puis l’affecte à “a”, soit a=True. Même chose dans le cas contraire, avec False. Cette optimisation n'est valable que pour le renvoi d’une valeur Boolean.

Fonctions conditionnelles

(non exhaustif vu les très nombreuses formes imaginables)

Il vaut mieux utiliser “If Condition Then” que “If Condition = True Then”, ça accélère légèrement.

Dans une série de tests conditionnels, quel que soit le nombre de conditions et la complexité des conditions, la forme “If… Then… ElseIf… End If” est la plus performante.

<note tip>Il faut savoir aussi regarder le côté pratique du codage, ce qui privilégie sans doute Select Case…</note>

```
If (condition1) Then
  (instructions1)
ElseIf (condition2) Then
  (instructions2)
ElseIf ...
End If
```

Forme équivalente : (+1.1*) If (condition1) Then (instructions1) If (condition2) Then (instructions2)
Forme équivalente : (+1.2*) Select Case (variable) Case (condition1) (instructions1) Case (condition2) (instructions2) ... End Select

On pourrait se poser la question si on a une série de conditions rassemblées avec un opérateur logique, par exemple OR. Mais la structure “ If... Then... ElseIf... End If ” pourrait être toujours utilisée et resterait le plus rapide, malgré une très grosse lourdeur dans le codage, puisqu'il faut répéter x fois les mêmes instructions...

Les différentes écritures de If

En VBA, les différentes formes d'écriture de l'instruction “IF” ont une influence sur la vitesse d'exécution. (Cette différence n'existe pas après compilation avec VB.)

```
If (condition) Then (instructions1) Else (instructions2)
```

Optimisation des références

Lorsque des objets sont appelés plusieurs fois, il est avantageux d’utiliser “With”.

Voici un exemple tiré de l’aide de VBA sur le sujet : “L'instruction With permet de spécifier un objet ou un type défini par l'utilisateur pour une série d'instructions. Les instructions With accélèrent l'exécution des procédures et permettent d'éviter des saisies répétitives. Vous pouvez imbriquer des instructions With pour en améliorer l'efficacité. L'exemple suivant insère une formule dans la cellule A1, puis formate la police.”

Exemple :

```
Sub MyInput()
  With Workbooks("Book1").Worksheets("Sheet1").Cells(1, 1)
    .Formula = "=SQRT(50)"
    With .Font
      .Name = "Arial"
      .Bold = True
      .Size = 8
    End With
  End With
End Sub
```

Optimisation des chaînes

Certaines fonctions sur les chaînes existent en deux versions, correspondant soit à un type de données en Variant, soit en String déclaré par le signe dollar ($). Cette dernière version est la plus rapide.

Par exemple, String deviendra String$.

Les fonctions concernées sont :

- Chr$
- ChrB$
- CurDir$
- Date$
- Dir$
- Error$
- Format$
- Hex$
- Input$
- InputB$
- LCase$
- Left$
- LeftB$
- LTrim$
- Mid$
- MidB$
- Oct$
- Right$
- RightB$
- RTrim$
- Space$
- Str$
- String$
- Time$
- Trim$
- UCase$

String$

L’appel à certaines de ces fonctions peuvent être encore plus optimisée, en déclarant les paramètres de l’instruction dans une variable adaptée (Utile seulement dans une boucle).

```
Dim z As String, lg As String
z = "0"
lg = String$(1000, z)
```

Optimisation des boucles

La variable d'une boucle doit être déclarée pour être optimale. Option Explicit est donc très pratique ici...

Parmi toutes les syntaxes et conditions d'utilisation on peut faire un classement des boucles en fonction de leur rapidité d'exécution, pour réaliser exactement le même nombre de passages.

<note tip>Noter qu'on peut retenir uniquement deux formes de boucles optimales :

- FOR TO NEXT, même dans le cas où l'on connaît exactement les bornes,
- DO.. LOOP (UNTIL/WHILE) sinon.</note>

Boucle for

Forme optimisée :

```
For... To... (Step)... Next
```

Plus rapide pour le type Integer.

A défaut, si la borne > 32766 alors prendre le type Long (presque équivalent).

Forme optimisée :

```
Dim Variable As Long, borne_fin As Long
borne_fin = (un nb ou une expression)
FOR Variable = (un nb ou une expression) TO borne_fin STEP (un nb ou une
expression)
NEXT
```

Boucle do

- Do.. Loop (While/Until)
- Do (While/Until)... Loop
- Do.. Exit Do... Loop
- While... Wend
- étiquette... Goto étiquette ... Goto suite

Forme optimisée :

```
Dim A As Long
```

```
Do
  A = A + 1
Loop Until A = 10000000
```

Exécuter une instruction une fois sur x passages

Forme optimisée :

```
Dim Saut As Long, A As Long, nb As Long
Saut = 90
For nb = 0 To 10000000
  If nb = Saut Then Saut = Saut + 90: A = A + 1 'instructions
Next nb
```

Optimisation pour Excel

Pour un gain global, il peut être intéressant de désactiver certains processus intrinsèques à Excel.

Ainsi, la désactivation temporaire de la mise à jour de l'affichage, en début du code, permet un gain d'exécution important.

De même, si le projet le permet, la désactivation temporaire du recalcul des références des formules peut être désactivée. Bien sûr, il faut faire attention à ne pas faire dépendre la suite du code d'un résultat d'une formule dans une cellule. Le gain global dépendra essentiellement de la façon dont le code est construit. Il peut s'élever à 50%.

Exemple :

```
'Désactive la mise à jour de l'affichage
Application.ScreenUpdating = False
'Désactive la mise à jour des recalculs
Application.Calculation = xlCalculationManual
'code modifiant les cellules
For x = 1 To 4000
  ActiveSheet.Cells(x, 1) = x
Next x
'Ré-activations
Application.Calculation = xlCalculationAutomatic
Application.ScreenUpdating = True
```