Vk

Zeste de savoir

Accélérer l'exécution de ses macros VBA sur Excel

28 avril 2023

Table des matières

Introduction

Qui ne connait pas Microsoft Excel? Ce tableur né dans les années 80 a su se faire une place de choix au fil des années.

Une de ses forces est le langage de macro VBA permettant à tout un chacun de réaliser des programmes simples ou complexes. Comme certains aspects de ce langage sont parfois méconnus et qu'il y a une interface de programmation (objets, méthodes, …) propre au logiciel, il arrive bien souvent d'obtenir des macros qui fonctionnent… mais qui prennent du temps, voire beaucoup de temps pour s'exécuter!

Ça tombe bien, car à travers ce billet nous allons voir quelques astuces liées au code pour que nos macros prennent moins de temps.

C'est parti!

1. Les bonnes pratiques de programmation

Nous commençons ce billet en douceur avec la notion des bonnes pratiques de développement. Ces dernières feront surtout gagner du temps au niveau de la conception et de la réalisation du programme, tout en réduisant les risques d'erreur.

1.1. Déclarer les types des variables

Donner un type à une variable, c'est aussi lui donner une certaine taille en mémoire. Le but est de donner le type adéquate à la donnée. De cette manière sa taille sera fixe, ni trop grande, ni trop petite.

```
1  Private Sub Exemple_Typage_Variables()
2      Dim uneVariable ' Variable non typée
3      Debug.Print VarType(uneVariable) ' => 0 : Empty
4      uneVariable = "Toto"
5      Debug.Print VarType(uneVariable) ' => 8 : String
6
7      Dim iUnEntier As Integer ' Variable typée
8      iUnEntier = 100
9      Debug.Print VarType(iUnEntier)  ' => 2 : Integer
10     ' iUnEntier = uneVariable ' => Erreur d'exécution :
           incompatibilité de types
11 End Sub
```

i Pour être sûr de ne pas oublier de déclarer le type de vos variables, vous pouvez ajouter `Option Explicit` au début de votre code. En conséquence, une fenêtre s'ouvrira au moment de l'exécution si des variables ne sont pas déclarées.

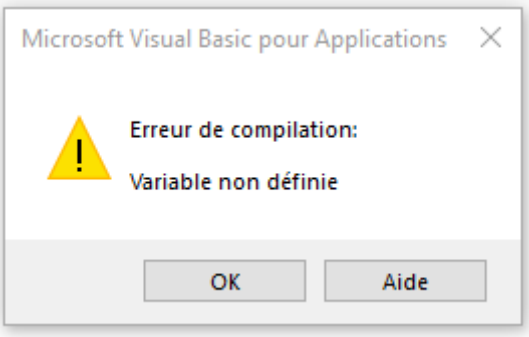


FIGURE 1.1. – Fenêtre erreur de compilation

1.2. Utiliser des variables

Ne lésinez pas sur les variables, en particulier lorsque les mêmes données reviennent régulièrement. Par exemple, une feuille de calcul utilisée à plusieurs endroits dans le code pourra être chargée une fois.

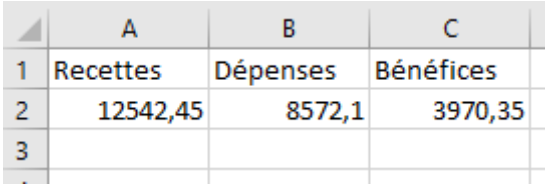
```
1  Option Explicit
2
```

```vba
Public Sub Exemple_Utilisation_Variables()
    Dim shPage1 As Worksheet
    Set shPage1 = Sheets("Feuil1")

    Dim dRecettes As Double, dDepenses As Double
    dRecettes = 12542.45
    dDepenses = 8572.1

    shPage1.Range("A1") = "Recettes"
    shPage1.Range("B1") = "Dépenses"
    shPage1.Range("C1") = "Bénéfices"

    shPage1.Range("A2") = dRecettes
    shPage1.Range("B2") = dDepenses
    shPage1.Range("C2") = dRecettes - dDepenses
End Sub
```



	A	B	C
1	Recettes	Dépenses	Bénéfices
2	12542,45	8572,1	3970,35
3			

FIGURE 1.2. – Résultat exemple utilisation variables

1.3. Utiliser des blocs With

Dans le même genre, le bloc `With` permet d'appliquer de multiples opérations à la suite à un élément en y accédant en une fois.

```vba
Public Sub Exemple_Utilisation_Blocs_With()
    With Worksheets("Feuil1")
        With .Range("A1:C1").Font
            .Bold = True
            .Size = 12
        End With

        With .Range("A1:C2").Borders
            .LineStyle = xlContinuous
            .ColorIndex = 23
            .Weight = xlThin
        End With
    End With
End Sub
```

	A	B	C
1	**Recettes**	**Dépenses**	**Bénéfices**
2	12542,45	8572,1	3970,35
3			

FIGURE 1.3. – Résultat exemple utilisation blocs With

1.4. Structurer son code

1.4.1. Portée des variables

Si une variable est utilisée à plusieurs endroits, il peut être intéressant de lui donner une portée plus large si ce n'est pas déjà fait afin de la déclarer qu'une seule fois (constantes, …).

1.4.2. Utilisation de procédures et fonctions

Découper son code permet de le rendre plus modulable, donc plus réutilisable, donc plus compréhensible aussi. Chaque procédure ou fonction doit avoir son propre rôle.

1.4.3. Complexité algorithmique

Il existe un domaine de l'informatique pour évaluer la complexité algorithmique . Parfois, la lenteur vient de l'algorithme! Par exemple, il convient d'éviter d'imbriquer plus de deux boucles ensemble.

```
1  Public Sub Exemple_Complexite_Algorithmique()
2      Dim i As Integer, j As Integer, k As Integer
3      For i = 0 To 10000
4          For j = 0 To 10000
5              For k = 0 To 10000
6                  ' à éviter !
7              Next k
8          Next j
9      Next i
10 End Sub
```

Au cours de cette première section, nous avons vu qu'en utilisant des variables typées, en intégrant des blocs `With` et en organisant notre code, nous pouvions gagner du temps à l'exécution, mais surtout à l'écriture de nos programmes.

2. Les paramètres de l'application

Dans cette seconde section, nous allons nous intéresser aux paramètres de l'application qui nous permettront de gagner du temps dans certains cas.

2.1. Les options de calcul des formules

Vous connaissez peut-être les options de calcul des formules dans l'onglet "Formules".

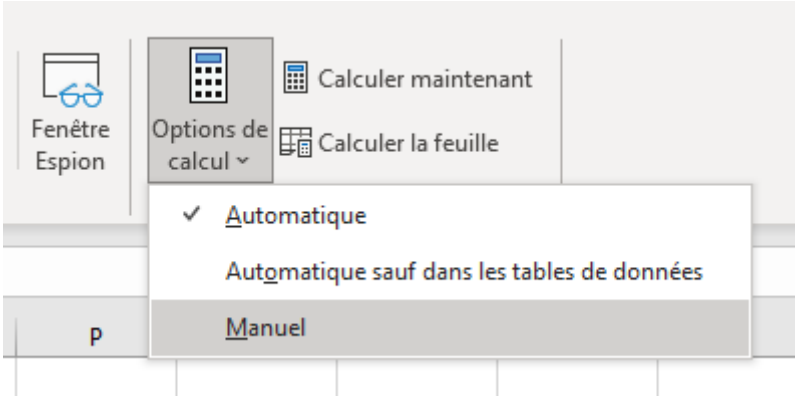


FIGURE 2.4. – Options de calcul de l'onglet Formules

En mode automatique, les formules sont recalculées à chaque changement dans les dépendances.

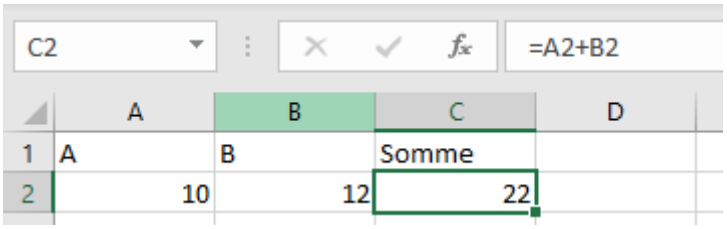


FIGURE 2.5. – Résultat exemple options calcul des formules

Imaginons une formule faisant la somme d'une colonne. Si au cours du programme, des données sont ajoutées, modifiées ou supprimées dans cette colonne, la formule sera recalculée à chaque fois. Parfois, il est donc judicieux de passer en mode manuel le temps du programme afin de gagner en performance. Cela se fait avec la propriété `Calculation` de l'objet `Application`.

```
1  Public Sub Exemple_CalculationMode()
2      Dim iCalculation As Integer
3      iCalculation = Application.Calculation ' Sauvegarde du
           paramètre initial
4
5      Application.Calculation = xlCalculationManual ' Mode manuel
6
7      ' Traitement ...
8
9      Application.Calculation = iCalculation ' Rétablissement du
           paramètre initial
10 End Sub
```

Si durant le programme, vous souhaitez recalculer des formules pour accéder aux nouvelles valeurs, vous pouvez utiliser la méthode `Calculate` sur une feuille ou encore sur l'application.

Voici un récapitulatif des valeurs possibles pour `Calculation`:

Nom	Description
xlCalculationManual	calcul lancé par utilisateur
xlCalculationSemiautomatic	calcul géré par Excel en ignorant les modifications dans les tableaux
xlCalculationAutomatic	calcul géré par Excel

2.2. L'option de mise à jour de l'écran

Par défaut, les opérations du programme se déroulent sous nos yeux. Si le programme insère des centaines voire des milliers de lignes, nous les verrons être écrites au fur et à mesure.

Pour désactiver la mise à jour de l'affichage le temps du programme, il est possible de jouer sur le paramètre `ScreenUpdating` de l'objet `Application`.

```
1  Public Sub Exemple_ScreenUpdating()
2      Application.ScreenUpdating = False ' Désactivé
3
4      Dim shPage2 As Worksheet
5      Dim i As Integer
6
7      Set shPage2 = Sheets("Feuil2")
8      For i = 1 To 10000
9          shPage2.Cells(i, 1) = i
10     Next i
11
12     Application.ScreenUpdating = True ' Activé
13 End Sub
```

En reprenant notre exemple, le résultat sera que les lignes apparaîtront en un coup à l'écran.

2.3. L'option des alertes

Selon les instructions, il arrive que l'exécution de macros entraîne l'ouverture de boîtes d'alerte. Ce qui est gênant, c'est que celles-ci bloquent la continuation du programme dans l'attente d'une réponse humaine.

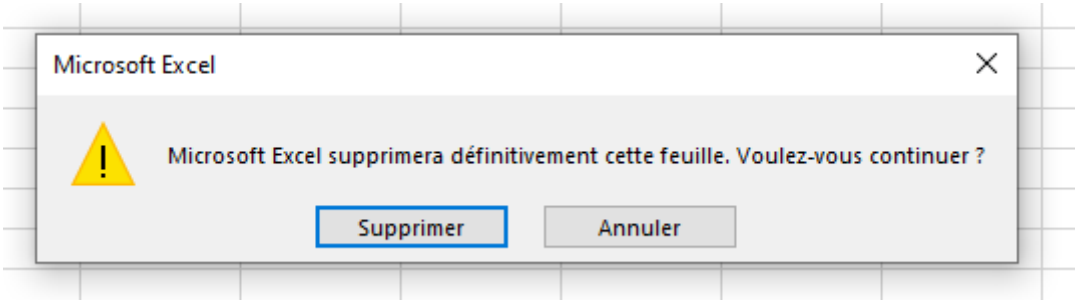


FIGURE 2.6. – Fenêtre d'alerte

Si vous êtes sûr qu'il n'y a rien de dangereux dans votre programme, vous pouvez désactiver celles-ci avec la propriété `DisplayAlerts` de l'objet `Application`.

```vba
Public Sub Exemple_DisplayAlerts()
    Application.DisplayAlerts = False ' Désactivé

    ' Traitement ...

    ' Application.DisplayAlerts = True ' Remis automatiquement à
      True à la fin
End Sub
```

Au cours de cette section, nous avons vu que nous pouvions désactiver le calcul automatique des formules, la mise à jour de l'affichage et l'affichage des alertes pendant l'exécution de nos macros, en fonction de nos besoins afin de réduire de façon importante le temps d'exécution.

3. Lecture et écriture de grands ensembles de données

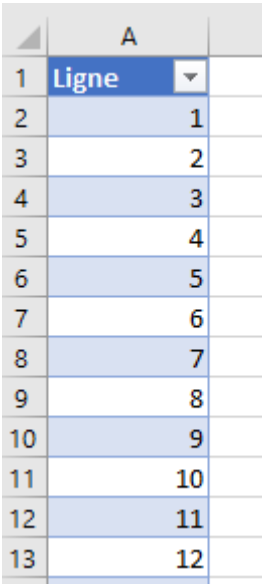
Dans cette dernière section, nous allons voir qu'il est préférable de travailler sur les plus grands ensembles de données possibles tant en lecture qu'en écriture.

Lire ou écrire une ligne d'un tableau d'un coup est plus rapide que de lire ou écrire celle-ci cellule par cellule. De même, lire ou écrire un tableau par bloc de X lignes est plus rapide que de lire ou écrire celui-ci ligne par ligne. De même, lire ou écrire un tableau d'un coup est plus rapide que lire ou écrire celui-ci par bloc de X lignes.

En VBA, les variables de type `Array` nous aideront à effectuer cela.

3.1. Lecture des données d'un coup

Imaginons que nous ayons un tableau de 10000 lignes contenant le numéro de la ligne.



	A
1	Ligne
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8
10	9
11	10
12	11
13	12

FIGURE 3.7. – Tableau de 10000 lignes à lire

Nous pouvons lire l'intégralité de celui-ci d'un coup comme ceci:

```vba
Public Sub Exemple_Lecture_Donnees()
    Dim shPage4 As Worksheet
    Set shPage4 = Sheets("Feuil4")

    Dim loTableau1 As ListObject
    Set loTableau1 = shPage4.ListObjects("Tableau1")

    Dim vDonnees As Variant
    vDonnees = loTableau1.DataBodyRange.Value
    Debug.Print (VarType(vDonnees)) '=> 8204: Tableau (8192) de
        variant (12)
End Sub
```

3.2. Écriture des données d'un coup

Voici un exemple VBA écrivant l'intégralité d'un tableau d'un coup:

```vba
Public Sub Exemple_Ecriture_Donnees()
    Application.ScreenUpdating = False

    ' Construction du tableau 2D
    Dim vDonnees(9999, 2) As Variant
    Dim i As Long
    For i = LBound(vDonnees, 1) To UBound(vDonnees, 1)
        vDonnees(i, 0) = i + 1
```

```
        vDonnees(i, 1) = 9999 - i
    Next i

    Dim shPage5 As Worksheet
    Set shPage5 = Sheets("Feuil5")
    Dim loTableau2 As ListObject
    Set loTableau2 = shPage5.ListObjects("Tableau2")
    ' Suppression contenu tableau
    If Not loTableau2.DataBodyRange Is Nothing Then
        loTableau2.DataBodyRange.Rows.Delete
    End If

    ' Écriture du tableau d'un coup
    loTableau2.ListRows.Add.Range.Resize(UBound(vDonnees, 1),
        UBound(vDonnees, 2)).Value = vDonnees

    Application.ScreenUpdating = True
End Sub
```

Pendant cette section, nous avons vu qu'il est plus intéressant en terme de performance de travailler sur de grands ensembles de données en lecture et écriture lorsque c'est possible.

Conclusion

C'est déjà la fin de ce billet.

Au cours de celui-ci, nous avons vu trois grandes façons d'accélérer l'exécution de nos macros VBA sur Excel: en respectant les bonnes pratiques de programmation, en jouant sur les paramètres de l'application et en utilisant des arrays pour lire et écrire de grands ensembles de données.

En dehors du code, d'autres sources de lenteur sont possibles, notamment la configuration matérielle.

À bientôt!

Quelques ressources:

- Page Wikipédia
- Site Excel-Pratique
- Livre Programmation VBA pour Excel pour les nuls (Excel 2010, 2013 et 2016), de John Walkenbach

Liste des abréviations

VBA Visual Basic for Applications. 1