Comment écrire des algos en LaTeX avec algorithm2e

Marie Duflot-Kremer

Dernière modification le 12 avril 2019

1 Remarques générales

L'appel du package algorithm2e ainsi que les mots clefs supplémentaires que j'ai définis se trouvent dans le fichier `macro.sty`. Pour pouvoir compiler le présent document j'ai besoin d'avoir les deux fichiers `algorithm2e.sty` et `macro.sty`.

Le premier des deux n'est absolument pas de moi, très long, et je ne l'ai pas touché. Le deuxième est de moi, plus court et normalement raisonnablement commenté. J'ai en particulier défini de nouveaux mots-clef pour écrire des algorithmes en pseudo-langage en français.

On peut faire plusieurs types d'algo : algocustom = sur fond coloré en jaune pâle, en texte un peu plus petit et avec deux arguments : la largeur (proportion de la largeur du texte (`textwidth`) utilisée) et le code de l'algo.

Cet algo utilise toute la largeur
Mais ne fait rien `# remarque, je ne lui ai rien demandé`

Cet algo ne fait rien
Mais prend moins de place en largeur

algoTD : sans fond de couleur, prend toute la largeur.

cet algo est sur fond non coloré et de taille normale `# mais commenté !`

du coup on ne se rend pas forcément compte que c'est un algo

Pour python on a les mêmes commandes `pythonTD` et `pythoncustom`.

Remarque : dans toute la suite j'écris le nom des commandes sous la forme `macommande` et pas \ `macommande` mais évidemment, comme c'est du LaTeX, il faut mettre un antislash pour toute commande.

2 Principales commandes

Les commandes utilisées

— `Var` : déclaration des variables, dans une fonction ou dans la partie principale de l'algo. Un seul argument.

Variables
| nbchiens : entier
| couleur : chaîne de caractères

— `Corps` : le corps d'une fonction/d'un algo. Un argument.

Début
| ici je mets
| mes instructions intéressantes
Fin

— `Fon` : fonction, avec deux arguments : l'en-tête et le corps

`# Fonction qui calcule la moitié d'un entier`
Fonction `moitié(`*n : entier*`) :` *réel*
| Mettre ici le corps de la fonction
Finfonction

— `Pour` : boucle pour. Deux arguments : le domaine de variation (i allant de ... à ...) et le corps de la boucle

Pour *i allant de 0 à 42* **Faire**
| Blablabla
Finpour

— `Tq` : boucle pour. Deux arguments : la condition et le corps de la boucle

Tant que *ma condition* **Faire**
| Blablabla
Fintantque

— `Si` : instruction conditionnelle, délimitée par Si et Finsi. Deux arguments : la condition et les instructions à faire

Si *condition* **Alors**
| Instructions
Finsi

— `eSi` : instruction conditionnelle avec un sinon. Trois arguments : la condition, les instructions à faire si elle est vraie, et les instructions à faire si elle est fausse.

Si *condition* **Alors**
| InstructionsA
Sinon
| InstructionsB
Finsi

— `tcs` : se met pour ajouter des commentaires « à la python » en fin de ligne. Un argument. Différentes options, comme le montrent les exemples ci-dessous (justifié à droite ou pas, avec ou sans saut de ligne après).

```
# commentaire sur une ligne, termine la ligne
m ← 12 # ou après le code
m ← 13                                                    # justifié à droite

Si condition Alors                                      # pb de saut de ligne

| Je fais ci
Sinon
| Je fais ça
Finsi
Si condition Alors                                  # pas de pb de saut de ligne
| Je fais ci
Sinon                      # et je peux même mettre des commentaires après le sinon
| Je fais ça
Finsi
```

— `Selon` : instruction de disjonction de cas. Deux arguments : l'expression à évaluer et le reste (la liste des cas, cf items suivants).

— `Cas` : un cas du selon. Délimité par Cas et Fincas. Deux arguments : la valeur et les instructions à faire si l'expression a cette valeur.

— `Autre` : cas par défaut du Selon. Un seul argument.

Selon *expression* **faire**
| **Cas** *v1*
| | Instructions1
| **Fincas**
| **Cas** *v2*
| | Instructions2
| **Fincas**
| ...
| **Autres** `# optionnel`
| | Instructions par défaut
| **Fincas**
Finselon

— Typ : pour la section de déclaration de types, que je mets en début d'algo, avant même la déclaration des fonctions :

Types
| collec_t : tableau d'entiers [10]

— `Mystruct` : pour définir les enregistrements (je ne mets pas grand chose d'autre que ça dans la section Types). Un seul argument.

tpiece_t **: enregistrement**
| largeur : entier `# en millimètres`
| longueur : entier `# en millimètres`
| hauteur : entier `# en millimètres`
| bois : chaîne de caractères

3 Exemple d'algorithme complet

Extrait d'un sujet d'examen, du coup les lignes sont numérotées avec la commande `LinesNumbered`

```
   # fonction mystère
 1 Fonction mystere(n : ______ ) : ______
 2 |  Variables
 3 |  |  i : ______
 4 |  |  res : ______
 5 |  Début
 6 |  |  res ← Faux
 7 |  |  Pour i allant de 1 à n Faire
 8 |  |  |  Si 2 × i = n Alors
 9 |  |  |  |  res ← Vrai
10 |  |  |  Finsi
11 |  |  Finpour
12 |  |  renvoyer res
13 |  Fin
14 Finfonction
   # Algorithme mystère, à vous de deviner ce qu'il fait
15 Variables
16 |  nbtest : entier
17 Début
18 |  nbtest ← Saisir("Donner un entier : ")
19 |  Si mystere(nbtest) Alors
20 |  |  afficher(" Ca fonctionne pour ", nbtest)
21 |  Sinon
22 |  |  afficher(" Ca ne fonctionne pas pour ", nbtest)
23 |  Finsi
24 Fin
```

4 Et en python ?

Le package `algorithm2e.sty` permet d'écrire des programmes dans différents langages. pour lui dire que c'est du python, on écrit juste la commande python en début d'algo. Suite à cela le package sait qu'il ne doit pas afficher de point virgule en fin de ligne, ni de mot clef de fin de bloc. Mes commandes `pythonTD` et `pythoncustom` le font déjà.

— 5 commandes pour écrire les opérateurs et valeurs booléennes en gras. Je les ai appelées `aand`, `oor`, `nnot`, `true` et `false`.
and, **or**, **not**, **True**, **False**

— `tcs` fonctionne pareil que pour les algos, mais on ajoute `tcmu` pour les commentaires multilignes

```
"""Voici un super long commentaire de python qui ne tient pas sur une seule ligne
   et du coup entre triple guillemets ça marche très bien                    """
```

— `Fon` : comme en algo, permet de définir une fonction. Deux arguments : le contenu de la signature (la première ligne) et le corps de la fonction.

def `mafonction(`*x*`):`
   `"""param : x : int, résultat : int`
   `Calcule et retourne blablabla                                            """`
   Et là on fait le calcul
   return toto

— `If` : instruction conditionnelle : deux arguments, la condition et les instructions à exécuter.

if *condition*:
   Instructions

— `eIf` : instruction conditionnelle avec un else. Trois arguments : la condition et les deux suites d'instructions à faire suivant la valeur de la condition.

if *condition*:
   Instructions1
else:
   Instructions2

— `ElseIf` (deux arguments) et `Else` (un argument) : permettent d'écrire le `elif` de python, et de mettre un `else` à la fin sans son if.

if *condition1*:
   Instructions1
elif *condition2*:
   Instructions2
else:

— `While` : aucune surprise, deux arguments

while *Condition*:
   Instructions

— `For` : on utilise le mot clef `KwTo` pour le `in` mis en gras.

for *elem* **in** *seq*:
   Instructions