

Le package `lyalgo` : taper facilement de jolis algorithmes

Code source disponible sur <https://github.com/bc-latex/ly-algo>.

Version 0.0.0-beta développée et testée sur Mac OS X.

Christophe BAL

2019-10-16

Table des matières

1	Introduction	3
2	lymath, un package qui vous veut du bien	3
3	Écriture pseudo-verbatim	3
4	Algorithmes en langage naturel	4
4.1	Comment taper les algorithmes	4
4.2	Numérotation des algorithmes	6
4.3	Des algorithmes encadrés	7
4.4	Un titre minimaliste	8
4.5	Un premier ensemble de macros additionnelles ou francisées	8
4.5.1	Entrée / Sortie	8
4.5.2	Bloc principal	9
4.5.3	Boucles POUR et TANT QUE	9
4.5.4	Boucles RÉPÉTER JUSQU'À	10
4.5.5	Disjonction de cas SELON QUE	10
4.5.6	Tests conditionnelles SI / SINON SI / SINON	10
4.5.7	Diverses commandes	11
4.6	Citer les outils de base en algorithmique	11
5	Des outils additionnels pour les algorithmes	12
5.1	Convention en bosses de chameau	12
5.2	Affectations	12
5.3	Listes	12
5.3.1	Opérations de base.	12
5.3.2	Modifier une liste – Versions textuelles	13
5.3.3	Modifier une liste – Versions POO	13
5.3.4	Modifier une liste – Versions symboliques	13

5.3.5	Parcourir une liste	14
5.4	Boucles	14
6	Historique	16

1 Introduction

Le but de ce package est d'avoir facilement des algorithmes¹ ainsi que des contenus *verbatim* un peu flexibles². Les algorithmes mis en forme ne sont pas des flottants, par choix, et ils utilisent une mise en forme proche de la syntaxe Python.

2 lymath, un package qui vous veut du bien

Le package `lymath` est un bon complément à `lyalgo` : voir à l'adresse <https://github.com/bc-latex/ly-math>. Il est utilisé par cette documentation pour simplifier la saisie des formules.

3 Écriture pseudo-verbatim

En complément à l'environnement `verbatim` est proposé l'environnement `pseudoverb`, pour « pseudo verbatim », qui permet d'écrire du contenu presque verbatim : ci-après, la macro `\squaremacro` définie par `\newcommand\squaremacro{x^2}` est interprétée mais pas la formule mathématique.

Code L^AT_EX

```
\begin{pseudoverb*}
  Prix1 = 14 euros
+ Prix2 = 30 euros
-----
  Total = 44 euros

=====
Remarque
=====
Attention car les macros comme \squaremacro{} sont interprétées mais pas les formules
de maths comme $x^2$ !
\end{pseudoverb*}
```

La mise en forme correspondante est la suivante sans cadre autour.

Rendu réel

ATTENTION ! Le cadre ne fait pas partie de la mise en forme.

```
Prix1 = 14 euros
+ Prix2 = 30 euros
-----
Total = 44 euros

=====
Remarque
=====
Attention car les macros comme  $x^2$  sont interprétées mais pas les formules
de maths comme  $x^2$  !
```

Il est en fait plus pratique de pouvoir taper quelque chose comme ci-dessous avec un cadre autour où le titre est un argument obligatoire (*voir plus bas comment ne pas avoir de titre*).

1. Le gros du travail est fait par `algorithm2e`.
2. Tout, ou presque, est géré par `alltt`.

Une sortie console

```
Prix1 = 14 euros
+ Prix2 = 30 euros
-----
Total = 44 euros
```

Le contenu précédent s'obtient via le code suivant.

Code $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$

```
\begin{pseudoverb}{Une sortie console}
  Prix1 = 14 euros
+ Prix2 = 30 euros
-----
  Total = 44 euros
\end{pseudoverb}
```

Finissons avec une version bien moins large et sans titre de la sortie console ci-dessus. Le principe est de donner un titre vide via {}, **c'est obligatoire**, et en utilisant l'unique argument optionnel pour indiquer la largeur relativement à celle des lignes. En utilisant juste `\begin{pseudoverb}[.275]{}` au lieu de `\begin{pseudoverb}{Une sortie console}`, le code précédent nous donne ce qui suit.

```
Prix1 = 14 euros
+ Prix2 = 30 euros
-----
Total = 44 euros
```

À RETENIR

*C'est la version étoilée de `pseudoverb` qui en fait le moins.
Ce principe sera aussi suivi pour les algorithmes.*

4 Algorithmes en langage naturel

4.1 Comment taper les algorithmes

Le package `algorithm2e` permet de taper des algorithmes avec une syntaxe simple. La mise en forme par défaut de `algorithm2e` utilise des flottants, chose qui peut poser des problèmes pour de longs algorithmes ou, plus gênant, pour des algorithmes en bas de page. Dans `lyalgo`, il a été fait le choix de ne pas utiliser de flottants, un choix lié à l'utilisation faite de `lyalgo` par l'auteur pour rédiger des cours de niveau lycée.

Dans la section suivante, nous verrons comment encadrer les algorithmes. Pour l'instant, voyons juste comment taper l'algorithme suivant où tous les mots clés sont en français. Indiquons au passage l'affichage du titre de l'algorithme en haut et non en bas comme cela est proposé par défaut.

Algorithme 1 : Suite de Collatz (u_k) – Conjecture de Syracuse

Donnée : $n \in \mathbb{N}$

Résultat : le premier indice $i \in \llbracket 0; 10^5 \rrbracket$ tel que $u_i = 1$ ou (-1) en cas d'échec

Actions

```
 $i, imax \leftarrow 0, 10^5$   
 $u \leftarrow n$   
 $continuer \leftarrow \top$   
Tant Que  $continuer = \top$  et  $i \leq imax$  :  
  Si  $u = 1$  :  
    # C'est gagné !  
     $continuer \leftarrow \perp$   
  Sinon  
    # Calcul du terme suivant  
    Si  $u \equiv 0 \pmod{2}$  :  
       $u \leftarrow u/2$  ; # Quotient de la division euclidienne.  
    Sinon  
       $u \leftarrow 3u + 1$   
     $i \leftarrow i + 1$   
Si  $i > imax$  :  
   $i \leftarrow (-1)$   
Renvoyer  $i$ 
```

La rédaction d'un tel algorithme est facile car il suffit de taper le code suivant proche de ce que pourrait proposer un langage classique de programmation. Le code utilise certaines des macros additionnelles proposées par `lyalgo` (voir la section 5) ainsi que la macro `\ZintervalC` du package `lymath`. Nous donnons juste après le squelette de la syntaxe propre à `algorithm2e`.

Code $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$

```
\begin{algo*}  
  \caption{Suite de Collatz  $(u_k)$  -- Conjecture de Syracuse}  
  
  \Data{$n$ \in \mathbb{N}}  
  \Result{le premier indice  $i$  \in \ZintervalC{0}{10^5} tel que  $u_i = 1$  ou  $(-1)$   
    en cas d'échec}  
  
  \addalgoblank    % Pour aérer un peu la mise en forme.  
  
  \Actions{  
    $i, imax$ \Store 0, 10^5  
    \\  
    $u$ \Store $n$  
    \\  
    $continuer$ \Store \top  
    \\  
    \While{$continuer = \top$ \And $i \leq imax$}{  
      \uIf{$u = 1$}{  
        \Comment{C'est gagné !}  
        $continuer$ \Store \bot  
      } \Else {  
        \Comment{Calcul du terme suivant}
```

```

        \uIf{$u \equiv 0 \,,\,, [2]$}{
            $u \Store u / 2$
            \Comment*{Quotient de la division euclidienne.}
        } \Else {
            $u \Store 3u + 1$
        }
        $i \Store i + 1$
    }
}
\If{$i > imax$}{
    $i \Store (-1)$
}
\Return{$i$}
}
\end{algo*}

```

Le squelette du code précédent est le suivant.

Squelette du code algorithm2e

```

\caption{...}

\Data{...}
\Result{...}

\Actions{
    ...
    \While{...}{
        \uIf{...}{
            ...
        } \Else {
            ...
            \uIf{...}{
                ...
            } \Else {
                ...
            }
        }
    }
}
\If{...}{
    ...
}
\Return{...}
}

```

4.2 Numérotation des algorithmes

Avant de continuer les présentations, il faut savoir que les algorithmes sont numérotés globalement à l'ensemble du document. C'est plus simple et efficace pour une lecture sur papier.

4.3 Des algorithmes encadrés

La version non étoilée de l'environnement `algo` ajoute un cadre, comme ci-dessous, afin de rendre plus visibles les algorithmes.

Algorithme 2 : Un truc bidon

Donnée : $n \in \mathbb{N}^*$

Résultat : $\sum_{i=1}^n i$

Actions

```
|  $s \leftarrow 0$   
| Pour  $i$  allant de 1 jusqu'à  $n$  :  
|   |  $s \leftarrow s + i$   
| Renvoyer  $s$ 
```

Le code utilisé pour obtenir le rendu précédent est le suivant où sont utilisées certaines des macros additionnelles proposées par `lyalgo` (voir la section 5) où les macros `\NNs` et `\dsum` viennent du package `lymath`.

Code $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$

```
\begin{algo}  
  \caption{Un truc bidon}  
  \Data{$n$ \in \NNs$}  
  \addalgorithblank  
  \Result{$\dsum_{i = 1}^n i$}  
  \Actions{  
    $s$ \Store 0$  
    \\  
    \ForRange{$i$}{$1$}{$n$}{  
      $s$ \Store $s + i$  
    }  
    \Return{$s$}  
  }  
\end{algo}
```

L'environnement `algo` propose un argument optionnel pour indiquer la largeur relativement à celle des lignes. Ainsi via `\begin{algo}[.45] ... \end{algo}`, on obtient la version suivante bien moins large de l'algorithme précédent.

Algorithme 3 : Un truc bidon

Donnée : $n \in \mathbb{N}^*$

Résultat : $\sum_{i=1}^n i$

Actions

```
|  $s \leftarrow 0$   
| Pour  $i$  de 1 à  $n$  :  
|   |  $s \leftarrow s + i$   
| Renvoyer  $s$ 
```

On peut utiliser un environnement `multicols` pour un effet sympa.

Algorithme 4 : Un truc bidon

Donnée : $n \in \mathbb{N}^*$

Résultat : $\sum_{i=1}^n i$

Actions

```
|  $s \leftarrow 0$   
| Pour  $i$  de 1 à  $n$  :  
|   |  $s \leftarrow s + i$   
| Renvoyer  $s$ 
```

Algorithme 5 : Un truc bidon

Donnée : $n \in \mathbb{N}^*$

Résultat : $\sum_{i=1}^n i$

Actions

```
|  $s \leftarrow 0$   
| Pour  $i$  allant de 1 jusqu'à  $n$  :  
|   |  $s \leftarrow s + i$   
| Renvoyer  $s$ 
```

4.4 Un titre minimaliste

Dans l'exemple ci-dessous on voit un problème à gauche où l'on a utilisé `\caption{}` avec un argument vide pour la macro `\caption`. Si vous avez besoin juste de numéroter votre algorithme comme ci-dessous à droite, utiliser à la place `\algovoidcaption`.

Algorithme 6 :

Données : Un titre vide ci-dessus !

Résultat : ...

Actions

```
| ...
```

Algorithme 7

Données : ...

Résultat : ...

Actions

```
| ...
```

4.5 Un premier ensemble de macros additionnelles ou francisées

À SAVOIR – Le préfixe *u*

*Certains macros peuvent être préfixées par un u pour « unclosed » qui signifie « non fermé ».
Ceci sert à ne pas refermer un bloc via un trait horizontal.*

4.5.1 Entrée / Sortie

Nous donnons ci-dessous les versions au singulier de tous les mots disponibles de type « entrée / sortie ». Excepté pour `\InState` et `\OutState`, toutes les autres macros ont une version pour le pluriel obtenu en rajoutant un `s` à la fin du nom de la macro. Par exemple, le pluriel de `\In` s'obtient via `\Ins`.

Code $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$

```
\begin{algo}  
  \In{donnée 1}  
  \Out{donnée 2}  
\end{algo}
```

Mise en forme correspondante.

Entrée : donnée 1
Sortie : donnée 2

Code L^AT_EX

```
\begin{algo}
  \Data{donnée 1}
  \Result{donnée 2}
\end{algo}
```

Mise en forme correspondante.

Donnée : donnée 1
Résultat : donnée 2

Code L^AT_EX

```
\begin{algo}
  \InState{donnée 1}
  \OutState{donnée 2}
\end{algo}
```

Mise en forme correspondante.

État initial : donnée 1
État final : donnée 2

Code L^AT_EX

```
\begin{algo}
  \PreCond{donnée 1}
  \PostCond{donnée 2}
\end{algo}
```

Mise en forme correspondante.

Précondition : donnée 1
Postcondition : donnée 2

4.5.2 Bloc principal

Voici comment indiquer le bloc principal d'instructions avec deux textes au choix pour le moment.

Code L^AT_EX

```
\begin{algo}
  \Actions{Instruction 1}
  \Begin{Instruction 2}
\end{algo}
```

Mise en forme correspondante.

Actions
└ Instruction 1
Début
└ Instruction 2

4.5.3 Boucles POUR et TANT QUE

Voici les boucles de type POUR et TANT QUE proposées par le package.

Code L^AT_EX

```
\begin{algo}
  \For{$i$ \in uneliste$}{
    Instruction 1
  }
  \ForAll{$i$ \in uneliste$}{
    Instruction 2
  }
  \ForEach{$i$ \in uneliste$}{
    Instruction 3
  }
  \While{$i$ \in uneliste$}{
    Instruction 4
  }
\end{algo}
```

Mise en forme correspondante.

Pour $i \in \text{uneliste}$:
 └ Instruction 1
Pour Tout $i \in \text{uneliste}$:
 └ Instruction 2
Pour Chaque $i \in \text{uneliste}$:
 └ Instruction 3
Tant Que $i \in \text{uneliste}$:
 └ Instruction 4

4.5.4 Boucles RÉPÉTER JUSQU'À

Voici comment rédiger une boucle du type RÉPÉTER JUSQU'À.

Code L^AT_EX

```
\begin{algo}
  \Repeat{$i$ \in uneliste$}{
    Instruction
  }
\end{algo}
```

Mise en forme correspondante.

Répéter
 | Instruction
Jusqu'à Avoir $i \in \text{uneliste}$;

4.5.5 Disjonction de cas SELON QUE

La syntaxe pour les blocs conditionnels du type SELON QUE ne pose pas de difficultés de rédaction.

Code L^AT_EX

```
\begin{algo}
  \Switch{$i$}{
    \uCase{$i = 0$}{Instruction 1}
    \uCase{$i = 1$}{Instruction 2}
    \Case{$i = 2$}{Instruction 3}
  }
\end{algo}
```

Mise en forme correspondante.

Suivant i :
 └ **Cas** $i = 0$:
 | Instruction 1
 └ **Cas** $i = 1$:
 | Instruction 2
 └ **Cas** $i = 2$:
 | Instruction 3

4.5.6 Tests conditionnelles SI / SINON SI / SINON

Les blocs conditionnels SI / SINON SI / SINON se rédigent très naturellement.

Code L^AT_EX

```
\begin{algo}
  \uIf{$i = 0$}{
    Instruction 1
  }
  \uElseIf{$i = 1$}{
    Instruction 2
  }
  \Else{
    Instruction 3
  }
\end{algo}
```

Mise en forme correspondante.

Si $i = 0$:
| Instruction 1
Sinon Si $i = 1$:
| Instruction 2
Sinon
└ Instruction 3

4.5.7 Diverses commandes

Pour finir voici un ensemble de mots supplémentaires qui pourront vous rendre service. Le préfixe **m** permet d'utiliser des versions maculines des textes proposés.

Code L^AT_EX

```
\begin{algo}
  A \And B \Or C
  \\ \Return RÉSULTAT
  \\ \Ask "Quelque chose"
  \\ \Print "Quelque chose"
  \\ $k$ \From $1$ \To $n$
  \\ $k$ \ComingFrom $1$ \GoingTo $n$
  \\ $L$ \InThis
  \\ $L$ \LToR
  \\ $L$ \LToRm
  \\ $L$ \RToL
  \\ $L$ \RToLm
\end{algo}
```

Mise en forme correspondante.

A et B **ou** C
Renvoyer RÉSULTAT
Demander "Quelque chose"
Afficher "Quelque chose"
 k **de** 1 **à** n
 k **allant de** 1 **jusqu'à** n
 L **dans**
 L **parcourue de gauche à droite**
 L **parcouru de gauche à droite**
 L **parcourue de droite à gauche**
 L **parcouru de droite à gauche**

4.6 Citer les outils de base en algorithmique

Pour faciliter la rédaction de textes sur les algorithmes, des macros standardisent l'impression des noms des outils classiques de contrôle. Dans les exemples qui suivent, les préfixes **TT** et **AL** font référence à « *True Type* » pour une police à chasse fixe, et à « *AL-gorithme* » pour une écriture similaire à celle utilisée dans les algorithmes.

Liste des commandes de type « *True Type* ».

1. **\TTif** donne **SI-SINON**.
2. **\TTfor** donne **POUR**.
3. **\TTwhile** donne **TANT QUE**.
4. **\TTrepeat** donne **RÉPÉTER-JUSQU'À AVOIR**.

5. `\TTswitch` donne SUIVANT-CAS.

Liste des commandes de type « *algorithme* ».

1. `\ALif` donne **Si-Sinon**.
2. `\ALfor` donne **Pour**.
3. `\ALwhile` donne **Tant Que**.
4. `\ALrepeat` donne **Répéter-Jusqu'à Avoir**.
5. `\ALswitch` donne **Suivant-Cas**.

5 Des outils additionnels pour les algorithmes

5.1 Convention en bosses de chameau

Le package `algorithm2e` utilise, et abuse³, de la notation en bosses de chameau comme par exemple avec `\uIf` et `\Return` au lieu de `\uif` et `\return`. Par souci de cohérence, les nouvelles macros ajoutées par `lyalgo` en lien avec les algorithmes utilisent aussi cette convention même si l'auteur aurait préféré proposer `\putin` et `\forrange` à la place de `\PutIn` et `\ForRange` par exemple.

5.2 Affectations

Les affectations $x \leftarrow 3$ et $3 \rightarrow x$ se tapent `$x \Store 3$` et `$3 \PutIn x$` respectivement où chacune des macros `\Store` et `\PutIn` est de type mathématique.

5.3 Listes

AVERTISSEMENT – Premier indice

Pour le package, les indices des listes commencent toujours à un.

5.3.1 Opérations de base.

Voici les premières macros pour travailler avec des listes c'est à dire des tableaux de taille modifiable.

1. *Liste vide.*
`\EmptyList` imprime une liste vide `[]`.
2. *Liste en extension.*
`\List{4 ; 7 ; 7 ; -1}` produit `[4;7;7;-1]`.
3. *Le k^e élément d'une liste.*
`\ListElt{L}{1}` produit `L[1]`.
4. *La sous-liste des éléments jusqu'à celui à la position k .*
`\ListUntil{L}{2}` produit `L[..2]`.
5. *La sous-liste des éléments à partir de celui à la position k .*
`\ListFrom{L}{2}` produit `L[2..]`.
6. *Concaténer deux listes.*
`\AddList` est l'opérateur binaire \boxplus qui permet d'indiquer la concaténation de deux listes.
7. *Taille ou longueur d'une liste.*
La macro `\Len(L)` produit `taille(L)`.

3. Ce type de convention est un peu pénible à l'usage.

5.3.2 Modifier une liste – Versions textuelles

1. *Ajout d'un nouvel élément à droite.*
`\Append{L}{5}` produit « Ajouter le nouvel élément 5 après la fin de la liste L. »⁴.
2. *Ajout d'un nouvel élément à gauche.*
`\Prepend{L}{5}` produit « Ajouter le nouvel élément 5 avant le début de la liste L. »⁵.
3. *Extraction d'un élément.*
`\PopAt{L}{3}` produit « Élément à la position 3 dans la liste L, cet élément étant retiré de la liste. ».

5.3.3 Modifier une liste – Versions POO

Les versions étoilées des macros précédentes fournissent une autre mise en forme à la fois concise et aisée à comprendre⁶ avec une syntaxe de type POO⁷.

1. *Ajout d'un nouvel élément à droite.*
`\Append*{L}{5}` fournit `L.ajouter-droite(5)`.
2. *Ajout d'un nouvel élément à gauche.*
`\Prepend*{L}{5}` fournit `L.ajouter-gauche(5)`.
3. *Extraction d'un élément.*
`\PopAt*{L}{3}` fournit `L.extraire(3)`.

5.3.4 Modifier une liste – Versions symboliques

Des versions doublement étoilées permettent d'obtenir des notations symboliques qui sont très efficaces lorsque l'on rédige les algorithmes à la main⁸.

1. *Ajout d'un nouvel élément à droite.*
`\Append**{L}{5}` donne $L \leftarrow L \boxplus [5]$.
2. *Ajout d'un nouvel élément à gauche.*
`\Prepend**{L}{5}` donne $L \leftarrow [5] \boxplus L$.
3. *Extraction d'un élément – Version pseudo-automatique.*
`\PopAt**{L}{3}` donne $L \leftarrow L[..2] \boxplus L[4..]$ avec un calcul fait automatiquement par la macro. Bien entendu `\PopAt**{L}{1}` produit $L \leftarrow L[2..]$ au lieu de $L \leftarrow L[..0] \boxplus L[2..]$ puisque pour le package les indices des listes commencent toujours à un.
Il est autorisé de taper `\PopAt**{L}{k}` pour obtenir $L \leftarrow L[..k-1] \boxplus L[k+1..]$. Par contre, `\PopAt**{L}{k-1}` aboutit au truc très moche $L \leftarrow L[..k-1-1] \boxplus L[k-1+1..]$. Les items suivants expliquent comment gérer à la main les cas problématiques via des macros plus généralistes.

Attention ! On notera que contrairement aux versions `\PopAt` et `\PopAt*`, l'écriture symbolique agit juste sur la liste d'un point de vue algorithmique. Si besoin, avec `\PopAt**` il faudra donc indiquer au préalable où stocker l'élément extrait via $\dots \leftarrow L[k]$.

4. *Extraction d'éléments consécutifs.*
Lorsque les calculs automatiques ne sont pas faisables, on devra tout indiquer comme dans `\KeepLR{L}{k-2}{k}`⁹ afin d'avoir $L \leftarrow L[..k-2] \boxplus L[k..]$ qui est bien mieux que ce que nous avons obtenu ci-dessus : $L \leftarrow L[..k-1-1] \boxplus L[k-1+1..]$.

4. Le verbe anglais « *append* » signifie « ajouter ».

5. Le verbe anglais « *prepend* » signifie « préfixer ».

6. L'opérateur point `.` est défini dans la macro `\POOpoint`.

7. « *POO* » est l'acronyme de « *Programmation Orientée Objet* ».

8. L'opérateur \boxplus est défini dans la macro `\AddList`.

9. Le nom de la macro vient de « *keep left and right* » soit « garder à droite et à gauche ».

5. *Extractions juste à droite, ou juste à gauche.*

`\KeepL{L}{k}` permet d'afficher $L \leftarrow L[..k]$ et `\KeepR{L}{k}` permet quant à lui d'écrire $L \leftarrow L[k..]$ ¹⁰.

5.3.5 Parcourir une liste

Les macros `\ForInList` et `\ForInListRev` facilitent la rédaction de boucle sur une liste parcourue de façon déterministe.

Code L^AT_EX

```
\begin{algo}
  \ForInList{e}{L}{
    Instruction 1
  }
\end{algo}
```

Mise en forme correspondante.

Pour e dans L parcourue de gauche à droite :
 └ Instruction 1

Code L^AT_EX

```
\begin{algo}
  \ForInListRev{e}{L}{
    Instruction 1
  }
\end{algo}
```

Mise en forme correspondante.

Pour e dans L parcourue de droite à gauche :
 └ Instruction 1

5.4 Boucles

Une boucle POUR peut s'écrire de façon succincte via `\ForRange*`, ou bien de façon non ambiguë via `\ForRange` non étoilée. Voici ce que cela donne.

Code L^AT_EX

```
\begin{algo}
  \ForRange*{a}{0}{12}{
    Instruction 1
  }
\end{algo}
```

Mise en forme correspondante.

Pour a de 0 à 12 :
 └ Instruction 1

Code L^AT_EX

```
\begin{algo}
  \ForRange{a}{0}{12}{
    Instruction 1
  }
\end{algo}
```

Mise en forme correspondante.

Pour a allant de 0 jusqu'à 12 :
 └ Instruction 1

10. Les noms des macros viennent de « *keep left* » et « *keep right* » soit « *garder à gauche* » et « *garder à droite* ».

Pour en finir avec les boucles, l'exemple suivant montre comment obtenir une écriture symbolique. On utilise la macro `\CSinterval` proposée par le package `lymath`.

Code $L^A T_E X$

```
\begin{algo}
  \For{$a$ \in \CSinterval{0}{12}$}{
    Instruction 1
  }
\end{algo}
```

Mise en forme correspondante.

Pour $a \in 0..12$:
 ⊢ Instruction 1

6 Historique

Nous ne donnons ici qu'un très bref historique de `lyalgo` côté utilisateur principalement. Tous les changements sont disponibles uniquement en anglais dans le dossier `change-log` : voir le code source de `lyalgo` sur `github`.

2019-10-18 Le documentation a enfin son journal des changements principaux.

2019-09-03 Première version `0.0.0-beta` du package.