L'extension pour TEX et LATEX

# tuple

v0.2

20 décembre 2024

Christian Tellechea*

Cette extension met à disposition des opérations courantes pour les tuples de nombres, de façon développable, avec une syntaxe « `objet.méthode` » concise et facile à utiliser.

*. unbonpetit@netc.fr

# 1 Aperçu

On considère la liste de nombres (que l'on appellera désormais « tuple ») :

```
12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1
```

On peut définir un « objet » de type tuple, que l'on nomme par exemple « nn » avec l'instruction :

```
\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1,
3, 4.1}
```

Ce tuple « nn » sera utilisé tout au long de cette documentation et rappelé en commentaire dans tous les codes où il intervient.

On peut en extraire son maximum :

```
1) \tplexe{nn.max} \par
2) \edef\foo{\tplexe{nn.max}}\meaning\foo
```

1) 13.6
2) macro:->13.6

On peut également calculer la médiane des 5 plus petites valeurs, ce qui suppose :

1. d'ordonner le tuple (méthode `sorted`) ;
2. de ne retenir que les valeurs dont l'index est 0 à 4 (méthode `filter`) ;
3. de trouver la médiane des 5 nombres retenus (méthode `med`).

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}
1) \tplexe{nn.sorted.filter{idx<5}.med} \par
2) \edef\foo{\tplexe{nn.sorted.filter{idx<5}.med}}\meaning\foo
```

1) 3
2) macro:->3

On charge ce package hors LaTeX par

```
\input tuple.tex
```

et sous LaTeX par

```
\usepackage{tuple}
```

Ce package ne s'appuie sur aucun autre si ce n'est le module de LaTeX3 « `l3fp` », qui fait désormais partie du noyau LaTeX, pour notamment tirer parti de sa puissante macro `\fpeval`.

Si l'on n'utilise pas LaTeX, tuple va charger le fichier `expl3-generic.tex` afin de disposer du module `l3fp`.

# 2 Déclarer un objet tuple

La macro `\newtuple{`⟨*nom*⟩`}{`⟨*liste de nombres*⟩`}` permet de construire un « objet » tuple auquel on accède ensuite par son ⟨*nom*⟩.

Ce ⟨*nom*⟩ admet tous les caractères alphanumériques `[az][AZ][09]`, espaces et ponctuations. Les espaces qui le précèdent ou le suivent sont supprimés. Il est également possible de mettre une macro[2].

La ⟨*liste de nombres*⟩ est développée au maximum puis détokénisée avant d'être exploitée par le constructeur de l'objet tuple. Les éléments vides sont ignorés.

Par souci de simplification ou de vitesse[3], le choix est donné à l'utilisateur de spécifier le type de nombre qui composent le tuple à l'aide la macro `\tplsetmode` ;

— `\tplsetmode{int}` pour indiquer que tous les nombres sont des entiers signés entre $-2^{31}+1$ et $2^{31}-1$. Les comparaisons lors du tri sont faites avec `\ifnum`, les opérations sur les nombres avec `\numexpr`.

— `\tplsetmode{dec short}` pour indiquer des décimaux « courts », c'est-à-dire comprenant 8 chiffres en partie entière et 8 chiffres en partie décimale. Dans ce cas, les comparaison sont faites avec `\ifdim` et les opérations sur les nombres par un moteur de calcul rudimentaire embarqué au sein de `tuple`.

2. La macro ne sera jamais modifiée par le package `tuple` : en interne cette macro est détokenisée pour construire le nom plus complexe d'une macro.
3. Les différences de vitesse ne sont pas énormes, mais elles existent, voir page 8

— `\tplsetmode{dec long}`, qui est le mode par défaut, spécifie des décimaux « longs » qui s'entendent au sens de `l3fp`. Dans ce mode, les comparaisons et les opérations sont faites par `l3fp`.

Quel que soit le mode sélectionné, les calculs finaux à destination de l'utilisateur (écart-type, moyenne, quartile, etc) sont effectués par `l3fp`.

Il est possible de définir un tuple vide (c'est-à-dire ne contenant donc aucun nombre), mais beaucoup de méthodes renverront une erreur si elles sont exécutées sur une liste vide.

Il est en revanche impossible de redéfinir un tuple déjà existant (il faut pour cela passer par la méthode `store`).

# 3 Les méthodes

Pour exécuter des méthodes sur un tuple, on doit utiliser la syntaxe

`\tplexe{`⟨*nom du tuple*⟩`.`⟨*méthode 1*⟩`.`⟨*méthode 2*⟩`...`⟨*méthode n*⟩`}`

Aucun espace entre le « `.` » et le nom d'une méthode n'est admis. Il est donc illicite d'écrire « `.␣sorted` ».

Il existe 3 types de donnés pour le package `tuple` :

1. les nombres (et les données affichables) ;
2. les objets « tuples » ;
3. le type « stockage » qui caractérise des méthodes non développables effectuant des assignations.

Toutes les méthodes de ce package admettent en entrée un tuple (qui est le résultat des méthodes précédentes) et renvoient un résultat dont le type détermine à quel groupe appartient la méthode :

— groupe 1 « `tuple` ⟶ `nombre` » ;

— groupe 2 « `tuple` ⟶ `tuple` ». Les méthodes de ce groupe *ne modifient pas* le tuple initial[4], elles agissent sur un tuple temporaire qu'il est évidemment possible de sauvegarder avec une méthode du groupe ci-dessous ;

— groupe 3 « `tuple` ⟶ `stockage` » ;

La macro `\tplexe` et son argument forment un tout purement développable, sous réserve que les méthodes invoquées ne soient pas dans le groupe « `tuple` ⟶ `stockage` ».

Si aucune méthode n'est spécifiée, une méthode générique implicite du groupe « `tuple` ⟶ `nombre` », est exécutée et renvoie le tuple sous forme affichable.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
\tplexe{nn}
```

12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1

# 4 Les méthodes du groupe `tuple` ⟶ `nombre`

## 4.1 Les méthodes `len`, `sum`, `min`, `max`, `mean`, `med` et `stdev`

Toutes ces méthodes développables n'admettent pas d'argument et renvoient respectivement le nombre d'éléments, leur somme, le minimum, le maximum, la moyenne, la médiane et l'écart-type.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
1) len =   \tplexe{nn.len}\par
2) sum =   \tplexe{nn.sum}\par
3) min =   \tplexe{nn.min}\par
4) max =   \tplexe{nn.max}\par
5) mean =  \tplexe{nn.mean}\par
5) med =   \tplexe{nn.med}\par
6) stdev = \tplexe{nn.stdev}
```

1) len = 17
2) sum = 125.5
3) min = 2.9
4) max = 13.6
5) mean = 7.382352941176471
5) med = 6.9
6) stdev = 3.447460325944583

4. Elles ne le peuvent pas sinon, elles perdraient leur caractère développable !

## 4.2 La méthode quantile

Cette méthode développable a pour syntaxe

quantile{⟨*p*⟩}

où ⟨*p*⟩ doit être un nombre compris entre 0 et 1. La méthode renvoie le quantile correspondant à l'argument ⟨*p*⟩. La méthode utilisée est celle de la moyenne pondérée[5] ; il s'agit de l'interpolation linéaire du type « R7 » décrite dans l'article de wikipedia.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
1) \tplexe{nn.quantile{0.25}}\par
2) \tplexe{nn.quantile{0.5}}\par
2) \tplexe{nn.quantile{0.75}}\par
```

1) 4.3
2) 6.9
2) 10.1

La méthode `quantile{0.5}` est équivalente à la méthode `med`.

Il est important de noter que les espaces avant/après les arguments d'une méthode sont ignorés. Il est donc licite d'écrire `.quantile␣{0.5}␣`.

## 4.3 La méthode get

Cette méthode développable a pour syntaxe

get{⟨*index*⟩}

Le premier index est 0 et le dernier est $n-1$ où $n$ est le nombre d'éléments du tuple. Par conséquent, l'argument de `get` doit être entre 0 et $n-1$. On peut également utiliser des index négatifs où $-1$ représente l'index du dernier élément, $-2$ celui de l'avant dernier et ainsi de suite jusqu'à $-n$.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
1er élément     : \tplexe{nn.get{0}}\par
13e élément     : \tplexe{nn.get{12}}\par
dernier élément : \tplexe{nn.get{\tplexe{nn.len}-1}}\par
dernier élément : \tplexe{nn.get{-1}}% mieux que ci-dessus
```

1er élément : 12.7
13e élément : 5.1
dernier élément : 4.1
dernier élément : 4.1

L'argument de `get` est évalué avant d'être exploité : il est donc possible d'y mettre la macro purement développable `\tplexe` avec une méthode finale renvoyant un nombre entier.

## 4.4 La méthode pos

Cette méthode développable a pour syntaxe

pos{⟨*nombre*⟩}[⟨*n*⟩]

et renvoie l'index de l'occurrence n° ⟨*n*⟩ du ⟨*nombre*⟩ dans le tuple. Si le tuple ne contient pas cette occurrence, -1 est renvoyé. Si l'argument optionnel est omis, ⟨*n*⟩ vaut 1.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
index de 12.7   : \tplexe{nn.pos{12.7}}\par
index de 2.9    : \tplexe{nn.pos{2.9}}\par
index de 2.9[2] : \tplexe{nn.pos{2.9}[2]}\par
index de 2.9[3] : \tplexe{nn.pos{2.9}[3]}\par
index de 31.8   : \tplexe{nn.pos{31.8}}
```

index de 12.7 : 0
index de 2.9 : 3
index de 2.9[2] : 9
index de 2.9[3] : -1
index de 31.8 : -1

## 4.5 La méthode show

Cette méthode n'admet pas d'argument et a pour but de convertir un objet tuple en résultat affichable. Pour ce faire :

5. Si $p$ est l'argument de la méthode, on définit $h = (n-1)p + 1$ où $n$ est la longueur du tuple.
La méthode renvoie le nombre égal à $x_{\lfloor h \rfloor} + (h - \lfloor h \rfloor)(x_{\lfloor h \rfloor} + x_{\lceil h \rceil})$, où $x_k$ est le $k^{\text{e}}$ nombre du tuple ordonné.

— pour chaque élément, la macro publique `\tplformat`, admettant 2 arguments obligatoires, est exécutée. Le premier argument transmis est l'index de l'élément et le 2e argument est l'élément lui-même ;

— chaque résultat issu de la macro `\tplformat` est séparé du suivant par le contenu de la macro `\tplsep`.

Par défaut, ces deux macros ont le code suivant :

```
\def\tplformat#1#2{#2}% #1=index en cours #2=item en cours
\def\tplsep{, }
```

Le comportement par défaut est donc exactement le même que la méthode implicite qui est exécutée en dernier, et en particulier, la méthode est purement développable.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
\tplexe{nn.show}
```

12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1

On peut reprogrammer ces 2 macros pour créer une mise en forme plus évoluée. On utilise ici la macro `\tplfpcompare` qui est alias de la macro `\fp_compare:nNnTF` du module `l3fp`, pour comparer un élément à une valeur donnée. Dans les 2 exemples donnés ci-dessous, la méthode n'est plus purement développable à cause de l'utilisation des macros `\fbox` et `\textcolor`.

Mise dans une boite des 10 premiers éléments :

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
\def\tplformat#1#2{\ifnum#1<10 \fbox{#2}\else#2\fi}
\tplexe{nn.show}
```

12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1

Mise en rouge des éléments inférieurs à la moyenne :

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
\edef\nnmean{\tplexe{nn.mean}}%
\def\tplformat#1#2{\tplfpcompare{#2}<{\nnmean}
    {\textcolor{red}{#2}}
    {#2}%
}%
\def\tplsep{~; }%
\tplexe{nn.show}
```

12.7 ; 6.3 ; 11.7 ; 2.9 ; 5.5 ; 8.1 ; 4.3 ; 9.4 ; 13.6 ; 2.9 ; 6.9 ; 11.2 ; 5.1 ; 7.7 ; 10.1 ; 3 ; 4.1

# 5 Les méthodes du groupe `tuple` ⟶ `tuple`

Chaque fois qu'un tuple est généré ou modifié, ses méthodes `len`, `sum`, `min`, `max`, `mean`, `med`, `stdev` et `sorted` sont mises à jour.

## 5.1 La méthode `sorted`

Cette méthode développable n'admet pas d'argument et retourne un objet tuple avec ses éléments rangés dans l'ordre croissant.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
\tplexe{nn.sorted}
```

2.9, 2.9, 3, 4.1, 4.3, 5.1, 5.5, 6.3, 6.9, 7.7, 8.1, 9.4, 10.1, 11.2, 11.7, 12.7, 13.6

## 5.2 La méthode `set`

Cette méthode développable a pour syntaxe

`set{`⟨*index1*⟩`:`⟨*nombre1*⟩`,`⟨*index2*⟩`:`⟨*nombre2*⟩`,...}`

Dans le tuple issu des méthodes précédentes, remplace le nombre à l'⟨*index1*⟩ par ⟨*nombre1*⟩ et ainsi de suite si plusieurs assignations sont spécifiées dans une liste séparée par des virgules.

Chaque ⟨*index*⟩ doit être compris entre 0 et $n-1$, où $n$ est le nombre d'élément du tuple passé en entrée de la méthode. Les index négatifs de $-n$ à $-1$ sont également autorisés.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
\tplexe{nn.set{1:10,5:50,-1:-666}}
```

12.7, 10, 11.7, 2.9, 5.5, 50, 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, -666

## 5.3 La méthode add

Cette méthode développable, qui ajoute une ⟨*insertion*⟩ à un ou plusieurs index spécifiés, a pour syntaxe

add{⟨*index1*⟩:⟨*insertion1*⟩,⟨*index2*⟩:⟨*insertion2*⟩,...}

Il faut noter que dans cette syntaxe, une ⟨*insertion*⟩ peut être

- un nombre seul « add{⟨*index*⟩:⟨*nombre*⟩} »
- une liste csv de nombres qui *doit* être entre accolades « add{⟨*index*⟩:{n1,n2,n3...}} »
- un tuple auquel on accède par \tplexe : « add{⟨*index*⟩:{\tplexe{⟨*nom tuple*⟩}}} ».

En ce qui concerne les ⟨*index*⟩, ils peuvent être compris entre 0 et $n$ où $n$ est le nombre d'éléments du tuple. Les index négatifs entre $-n-1$ et $-1$ sont également permis :

- un ⟨*index*⟩ égal à 0 ou à $-n-1$ place l'⟨*insertion*⟩ au début du tuple passé en entrée ;
- un ⟨*index*⟩ égal à $n$ ou $-1$ place l'⟨*insertion*⟩ à la fin du tuple ;
- les ⟨*index*⟩ *ne sont pas* recalculés après chaque ⟨*insertion*⟩, mais seulement après la dernière. Il n'est donc pas équivalent d'écrire « .add{1:100,2:200} » et « .add{1:100}.add{2:200} ». En effet, 200 sera à l'index 3 dans le premier cas alors qu'il sera à l'index 2 dans le second cas.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
1) ajout en 1er     : \tplexe{nn.add{0:{666,667}}}\par
2) ajout index 11   : \tplexe{nn.add{11:{666,667}}}\par
3) ajout en dernier : \tplexe{nn.add{-1:{666,667}}}
```

1) ajout en 1er : 666, 667, 12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1
2) ajout index 11 : 12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4, 13.6, 2.9, 6.9, 666, 667, 11.2, 5.1, 7.7, 10.1, 3, 4.1
3) ajout en dernier : 12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1, 666, 667

```
\newtuple{X}{10,20,30,40,50,60}%
\newtuple{Y}{1,2,3,4}%
1) \tplexe{X.add{0:{\tplexe{Y}}}}\par
2) \tplexe{X.add{2:{\tplexe{Y}}}}\par
3) \tplexe{X.add{-1:{\tplexe{Y}}}}\par
4) \tplexe{X.add{1:100,2:200}}\par
5) \tplexe{X.add{1:100}.add{2:200}}
```

1) 1, 2, 3, 4, 10, 20, 30, 40, 50, 60
2) 10, 20, 1, 2, 3, 4, 30, 40, 50, 60
3) 10, 20, 30, 40, 50, 60, 1, 2, 3, 4
4) 10, 100, 20, 200, 30, 40, 50, 60
5) 10, 100, 200, 20, 30, 40, 50, 60

## 5.4 La méthode op

Cette méthode développable, qui effectue une ⟨*opération*⟩ sur tous les éléments du tuple, a pour syntaxe

op{⟨*opération*⟩}

L'⟨*opération*⟩ est une expression ne contenant pas d'accolades, évaluable par \fpeval une fois que toutes les occurrences de « val » aient été remplacées par la valeur de chaque élément, et toutes les occurrences de « idx » par son index.

```
\newtuple{X}{10,20,30,40,50,60}%
1) \tplexe{X.op{val+5}}\par
2) \tplexe{X.op{val*val}}\par
3) \tplexe{X.op{val+idx}}\par
4) \tplexe{X.op{idx<4 ? val-1 : val+1 }}
```

1) 15, 25, 35, 45, 55, 65
2) 100, 400, 900, 1600, 2500, 3600
3) 10, 21, 32, 43, 54, 65
4) 9, 19, 29, 39, 51, 61

## 5.5 La méthode filter

Cette méthode développable, qui sélectionne certains éléments selon un ou plusieurs critères, a pour syntaxe

filter{⟨*test*⟩}

et où ⟨*test*⟩ est un booléen ne contenant pas d'accolades, évaluable par `\fpeval` une fois que toutes les occurrences de « `val` » aient été remplacées par la valeur de chaque élément, et toutes les occurrences de « `idx` » par son index.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
1) \tplexe{nn.filter{val<10}}\par
2) \tplexe{nn.filter{val>5 && val<10}}\par
3) \tplexe{nn.filter{idx<3 || idx>13}}\par
4) \tplexe{nn.filter{val!=6.3 && val!=2.9 && val!=4.1}}
```

1) 6.3, 2.9, 5.5, 8.1, 4.3, 9.4, 2.9, 6.9, 5.1, 7.7, 3, 4.1
2) 6.3, 5.5, 8.1, 9.4, 6.9, 5.1, 7.7
3) 12.7, 6.3, 11.7, 10.1, 3, 4.1
4) 12.7, 11.7, 5.5, 8.1, 4.3, 9.4, 13.6, 6.9, 11.2, 5.1, 7.7, 10.1, 3

### 5.6 La méthode `comp`

Cette méthode développable compose deux tuples de même longueur avec une opération que l'utilisateur spécifie. Sa syntaxe est

`comp{`⟨*opération*⟩`}{`⟨*nom tuple*⟩`}`

où le tuple dont le nom est passé en deuxième argument *doit* avoir la même longueur que le tuple passé en entrée de la méthode.

L'⟨*opération*⟩ est une expression ne contenant pas d'accolades, évaluable par `\fpeval` une fois que toutes les occurrences de « xa » aient été remplacées par la valeur de chaque élément du tuple d'entrée, et toutes les occurrences de « xb » par celle du tuple spécifié en 2e argument.

Produit de 2 tuples et leur « `somprod` » :

```
\newtuple{A}{2,-4,3,7,-1}%
\newtuple{B}{-9,0,4,6,-2}%
produit : \tplexe{A.comp{xa*xb}{B}}\par
somprod : \tplexe{A.comp{xa*xb}{B}.sum}
```

produit : -18, -0, 12, 42, 2
somprod : 38

Calcul de la plus petite distance au point A(2.5 ; -0.5) connaissant la liste des abscisses et la liste des ordonnées d'une trajectoire (ici elliptique) :

```
\newtuple{ListX}{4,2,0.5,1,3,6.5}%
\newtuple{ListY}{2,1.5,0,-1.5,-2,0.5}%
\tplexe{ListX.comp{sqrt((xa-2.5)**2+(xb+0.5)**2)}{ListY}.min}
```

1.58113883008419

## 6 Les méthodes du groupe `tuple` ⟶ `stockage`

Comme ces méthodes ne renvoient pas un résultat puisqu'elles effectuent une assignation, elle ne sont pas développables, et *doivent* se trouver en dernière position. Si ce n'est pas le cas, toutes les méthodes qui les suivent seront ignorées.

### 6.1 La méthode `split`

Cette méthode coupe le tuple passé en entré à la méthode après l'index spécifié. La syntaxe est

`split{`⟨*index*⟩`}{`⟨*tuple1*⟩`}{`⟨*tuple2*⟩`}`

Le tuple passé en entrée de la méthode est coupé après l'⟨*index*⟩ : la partie avant la coupure est assignée, via `\newtuple` au tuple de nom « `tuple1` » et la partie restant au tuple de nom « `tuple2` ». Aucune vérification n'est faite sur l'existence des 2 tuples ; il est donc possible de remplacer silencieusement des tuples existants.

L'⟨*index*⟩ doit se trouver entre 0 et $n-2$ s'il est positif ou entre $-n$ et $-2$ s'il est négatif, $n$ étant le nombre d'éléments du tuple passé en entré.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
\tplexe{nn.split{5}{n1}{n2}}%
tuple avant : \tplexe{n1}\par
tuple après : \tplexe{n2}
```

tuple avant : 12.7, 6.3, 11.7, 2.9, 5.5, 8.1
tuple après : 4.3, 9.4, 13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1

### 6.2 La méthode store

Cette macro a pour but de stocker le résultat de la dernière méthode. Si ce résultat est un tuple, la syntaxe est

store{⟨*nom tuple*⟩}

et si le résultat est un nombre ou une donnée affichable venant de la méthode show :

store{⟨*macro*⟩}

Aucune vérification n'est faite sur l'existence du tuple ou de la macro. Il est donc possible de remplacer silencieusement un tuple déjà créé ou une macro existante.

```
%\newtuple{nn}{12.7, 6.3, 11.7, 2.9, 5.5, 8.1, 4.3, 9.4,
%              13.6, 2.9, 6.9, 11.2, 5.1, 7.7, 10.1, 3, 4.1}%
1) \tplexe{nn.sorted.filter{val>11}.store{nn1}}%
\tplexe{nn1}\par
2) \tplexe{nn1.len.store\nnlen}%
\meaning\nnlen\par
3) \def\tplformat#1#2{\fbox{#2}}\def\tplsep{ }%
\tplexe{nn1.show.store\nnshow}%
\meaning\nnshow\par
4) \edef\nndisp{\tplexe{nn1}}%
\meaning\nndisp% méthode générique
```

1) 11.2, 11.7, 12.7, 13.6
2) macro:->4
3) macro:->\fbox {11.2} \fbox {11.7} \fbox {12.7} \fbox {13.6}
4) macro:->11.2, 11.7, 12.7, 13.6

Pour stocker le résultat de la méthode générique, il faut utiliser \edef car \tplexe{⟨*tuple*⟩.store⟨*macro*⟩} est incorrect puisque dans ce cas, la méthode store s'applique à un tuple.

## 7 Génération de tuples

Pour générer un tuple, on peut utiliser la macro développable \gentuple qui est destinée à être appelée dans le 2e argument de \newtuple. Sa syntaxe est

\gentuple{⟨*valeurs initiales*⟩},\genrule⟨*règle de génération*⟩;\while||\until⟨*condition*⟩}

où :

- les ⟨*valeurs initiales*⟩ sont facultatives. Si elles sont présentes, elles *doivent* être suivies d'une virgule. La macro \gentuple détermine leur nombre *i* par comptage (*i* devant être au plus égal à 9). Ces valeurs initiales seront recopiées en début de tuple et par la suite, ont vocation à être utilisées dans la ⟨*règle de génération*⟩ à des fins de récurrence ;
- la ⟨*règle de génération*⟩ est une expression ne contenant pas d'accolades, évaluable par \fpeval une fois que dans les *i* valeurs précédentes, toutes les occurrences de \1 aient été remplacées par la valeur n°1, \2 par la valeur n°2, etc. De plus, chaque occurrence de \i est remplacée par la valeur de l'index en cours.
- la ⟨*condition*⟩ est un booléen ne contenant pas d'accolades, évaluable par \fpeval une fois que toutes les occurrences de « val » aient été remplacées par la valeur de l'élément calculé, et toutes les occurrences de « \i par son index. Si le mot-clé après ; est \while, la boucle est du type while⟨*condition*⟩...endwhile alors que si ce mot-clé est \until, il s'agit d'une boucle repeat...until⟨*condition*⟩.

Génération des 10 premiers entiers pairs :

```
\gentuple{\genrule (\i+1)*2 ; \until \i=9 }\par
ou\par
\gentuple{\genrule (\i+1)*2 ; \while \i<10 }
```

2, 4, 6, 8, 10, 12, 14, 16, 18, 20
ou
2, 4, 6, 8, 10, 12, 14, 16, 18, 20

Génération de 15 entiers aléatoires entre 1 et 10 :

```
\gentuple{\genrule randint(1,10) ; \until \i=14}
```

10, 7, 2, 4, 1, 1, 9, 3, 3, 7, 2, 3, 2, 9, 4

Génération des carrés d'entiers jusqu'à 500 :

```
\gentuple{\genrule\i*\i; \while val<500 }
```

0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484

Génération des 10 premiers termes la suite de Fibonacci :

```
\gentuple{1,1,\genrule\1+\2; \until \i=9 }
```

1,1,2, 3, 5, 8, 13, 21, 34, 55

Génération des 10 premiers termes de $u_0 = 1$ ; $u_1 = 1$ ; $u_2 = -1$ et $u_n = u_{n-3}u_{n-1} - u_{n-2}^2$ :

```
\gentuple{1,1,-1,\genrule\1*\3-\2*\2; \until \i=10}
```

1,1,-1,-2, -3, -1, -7, 20, -69, 83, -3101

Génération de la suite de Syracuse (connue aussi comme « suite $3n + 1$ ») de 15 :

```
\gentuple{15,\genrule \1/2=trunc(\1/2) ? \1/2 : 3*\1+1 ; \until val=1}%
\meaning\syr
```

15,46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1undefined

Altitude maximale et longueur de la suite de Syracuse de 27 :

```
\newtuple{syr27}
{\gentuple{27,\genrule \1/2=trunc(\1/2) ? \1/2 : 3*\1+1 ; \until val=1}}%
longueur = \tplexe{syr27.len}\par
alt max  = \tplexe{syr27.max}
```

longueur = 112
alt max = 9232

# 8 Conclusion

## 8.1 Motivation

D'une façon un peu étonnante, assez peu de choses existent pour manipuler et opérer sur des listes de nombres[6].

Le principal défi a surtout été d'offrir des macros développables et aussi une syntaxe originale dans le monde de TEX du type ⟨*objet*⟩`.`⟨*méthode 1*⟩`.`⟨*méthode 2*⟩`...`⟨*méthode n*⟩, où l'on peut enchainer les méthodes exécutées sur un « objet » qui est un tuple de nombres. J'ignore si des packages proposent ce genre de syntaxe, mais elle est finalement assez intuitive. Comme je suis assez peu familier avec la programmation de macros purement développables, j'ai voulu de nombreuses fois renoncer. Mais j'ai fini par trouver un trou de souris qui permet que le tout fonctionne à peu près, à l'exception des nombreux bugs qui doivent encore se cacher un peu partout.

Je remercie ceux qui en trouvent de me le signaler par email, voire même de proposer de nouvelles fonctionnalités.

## 8.2 Vitesse d'exécution

Lorsqu'on met un pied dans l'optimisation de la vitesse d'exécution, surtout de macros purement développables, on n'en ressort pas ! C'est un puits sans fond de questionnements sur les arguments des macros, sur des petites — ou grosses — astuces qui font gagner du temps, et un casse tête sur la façon de jongler avec les arguments délimités et les retrouver ensuite !

J'espère ne pas être tombé dans ce piège, car j'y suis à peine entré. De toutes façons, TEX n'est pas fait pour du calcul massif, car c'est avant tout un logiciel de composition. Pour le calcul, des outils performants qui battent TEX à plate couture existent à foison.

Quoiqu'il en soit, « macro développable » va un peu à l'encontre de « vitesse d'exécution ». À titre d'information, je mets ci-dessous les temps de compilation, en secondes, de la création de tuples contenant $n$ entiers aléatoires. Cela dépend de l'ordinateur utilisé, bien sûr, mais les ordres de grandeur sont bien révélateurs. On voit clairement qu'il est illusoire de dépasser le millier de nombres car le temps de création d'un tuple croit ensuite rapidement[7]. Ceci dit, qui utiliserait TEX pour des calculs sur autant de nombres ?

Ce tableau illustre les vitesses de création d'un tuple d'entiers signés aléatoires compris entre −1000 et 1000. Pour rappel la création d'un tuple suppose le tri de ses éléments (par tri rapide), le calcul de la somme des nombres et celui de la somme des carrés. Les 3 modes peuvent ainsi être comparés pour des tuples plus ou moins grands.

6. Il y a bien un package `commalists-tools`, mais il est clairement trop limité. Comme tous les packages de son auteur qui floode le CTAN, il ne fait qu'aligner *ad nauseam*, linéairement et sans véritable programmation, les macros de haut niveau des packages `tikz`, `listofitems`, `xstring`, `xint` et `simplekv`.

7. Sans compter que le tuple est immédiatement recréé et recalculé après avoir été modifié par les méthodes `set`, `add`, `op`, `filter`, `split` et `comp`

| Nb items | int | dec:short | dec:long |
|---|---|---|---|
| 25 | 0.001 s | 0.001 s | 0.006 s |
| 50 | 0.001 s | 0.003 s | 0.014 s |
| 100 | 0.006 s | 0.009 s | 0.037 s |
| 200 | 0.02 s | 0.028 s | 0.087 s |
| 400 | 0.092 s | 0.114 s | 0.254 s |
| 800 | 0.551 s | 0.643 s | 0.978 s |
| 1600 | 3.138 s | 3.538 s | 3.818 s |

## 8.3 Exemple : population des états

Le tuple \Wpop contient la population de chaque état dans le monde, en millions d'habitants[8] :

```
\newtuple\Wpop{
43.4, 2.8, 46.3, 37.8, 0.1, 46.1, 2.8, 0.1, 26.7, 9.0, 10.5, 0.4, 1.5,
174.7, 0.3, 9.5, 11.7, 0.4, 14.1, 0.8, 12.6, 3.2, 2.7, 217.6, 0.5, 6.6,
23.8, 13.6, 0.6, 17.1, 29.4, 39.1, 5.9, 18.8, 19.7, 1425.2, 7.5, 0.7,
52.3, 0.9, 6.2, 5.2, 29.6, 4.0, 11.2, 0.2, 1.3, 10.5, 26.2, 105.6, 5.9,
1.2, 0.1, 11.4, 18.4, 114.5, 6.4, 1.8, 3.8, 1.3, 1.2, 129.7, 0.9, 5.5,
64.9, 0.3, 0.3, 2.5, 2.8, 3.7, 83.3, 34.8, 10.3, 0.1, 0.4, 0.2, 18.4,
14.5, 2.2, 0.8, 11.9, 10.8, 10.0, 0.4, 1441.7, 279.8, 89.8, 46.5, 5.1,
9.3, 58.7, 2.8, 122.6, 11.4, 19.8, 56.2, 0.1, 4.3, 6.8, 7.7, 1.8, 5.2,
2.4, 5.5, 7.0, 2.7, 0.7, 31.1, 21.5, 34.7, 0.5, 24.0, 0.5, 0.4, 5.0,
1.3 ,129.4, 0.1, 3.5, 0.6, 38.2, 34.9, 55.0, 2.6, 31.2, 17.7, 0.3, 5.3,
7.1 ,28.2, 229.2, 2.1, 5.5, 4.7, 245.2, 4.5, 10.5, 6.9, 34.7, 119.1,
40.2 ,10.2, 3.3, 2.7, 51.7, 3.3, 1.0, 19.6, 144.0, 14.4, 0.0, 0.2, 0.1,
0.2 ,0.03, 0.2, 37.5, 18.2, 7.1, 0.1, 9.0, 6.1, 0.0, 5.7, 2.1, 0.8, 18.7,
61.0 ,11.3, 47.5, 21.9, 5.5, 49.4, 0.6, 10.7, 8.9, 24.3, 10.3, 71.9, 1.4,
9.3 ,0.1, 1.5, 12.6, 86.3, 6.6, 0.0, 0.0, 49.9, 37.9, 9.6, 68.0, 69.4,
341.8 ,0.1, 3.4, 35.7, 0.3, 29.4, 99.5, 0.6, 35.2, 21.1, 17.0}
Nombre : \tplexe{\Wpop.len}\par
Moyenne : \tplexe{\Wpop.mean}\par
Médiane : \tplexe{\Wpop.med}\par
Écart type : \tplexe{\Wpop.stdev}\par
Quintile \#1 : \tplexe{\Wpop.quantile{0.2}}\par
Quintile \#4 : \tplexe{\Wpop.quantile{0.8}}
```

Nombre : 204
Moyenne : 39.66338235294118
Médiane : 7.6
Écart type : 146.8985787142461
Quintile #1 : 0.8
Quintile #4 : 37.62

On modifie \Wpop, en ne retenant que les états « moyennement » peuplés. On considère arbitrairement leur population entre 10 et 100 millions d'habitants :

```
\tplexe{\Wpop.filter{val>=10 && val<=100}.store\Wpop}%
Nombre : \tplexe{\Wpop.len}\par
Moyenne : \tplexe{\Wpop.mean}\par
Médiane : \tplexe{\Wpop.med}\par
Écart type : \tplexe{\Wpop.stdev}\par
Quintile \#1 : \tplexe{\Wpop.quantile{0.2}}\par
Quintile \#4 : \tplexe{\Wpop.quantile{0.8}}

Répartition sur 6 intervalles égaux :\par
\begin{tabular}{lc}\\\hline
    De 10 à 25 & \tplexe{\Wpop.filter{val<25}.len}\\
    De 25 à 40 & \tplexe{\Wpop.filter{val>=25 && val<40}.len}\\
    De 40 à 55 & \tplexe{\Wpop.filter{val>=40 && val<55}.len}\\
    De 55 à 70 & \tplexe{\Wpop.filter{val>=55 && val<70}.len}\\
    De 70 à 85 & \tplexe{\Wpop.filter{val>=70 && val<85}.len}\\
    De 85 à 100& \tplexe{\Wpop.filter{val>=85}.len}\\\hline
\end{tabular}
```

Nombre : 79
Moyenne : 32.33037974683544
Médiane : 26.7
Écart type : 21.22899470798109
Quintile #1 : 12.6
Quintile #4 : 48.26
Répartition sur 6 intervalles égaux :

| | |
|---|---|
| De 10 à 25 | 38 |
| De 25 à 40 | 19 |
| De 40 à 55 | 10 |
| De 55 à 70 | 7 |
| De 70 à 85 | 2 |
| De 85 à 100 | 3 |

## 8.4 TODO list

À implémenter plus ou moins rapidement :

1. utiliser le tri par insertion pour trier les tuples presque triés obtenus suite aux méthodes add, set, op (risqué car dépend de l'opération !)
2. utiliser le tri fusion pour ajouter un tuple ou des nombres à un tuple ;
3. autres optimisation de la rapidité ?

8. Les données proviennent de https://www.unfpa.org/data/world-population-dashboard