

L'extension pour  $\text{T}_{\text{E}}\text{X}$  et  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$

# tuple

v0.1

16 novembre 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](mailto: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	1) 13.6
2) \edef\foo{\tplexe{nn.max}}\meaning\foo	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) 3
1) \tplexe{nn.sorted.filter{idx<5}.med} \par	2) macro:->3
2) \edef\foo{\tplexe{nn.sorted.filter{idx<5}.med}}\meaning\foo	

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  $\LaTeX$ 3 « `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>` :

- le `<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<sup>2</sup>.
- les nombres composant la `<liste de nombres>` s'entendent au sens du module `l3fp`, c'est-à-dire qu'ils couvrent une plage beaucoup plus étendue et disposent d'une précision très supérieure aux nombres décimaux (pris au sens de dimensions) de  $\TeX$  (voir la documentation de  $\LaTeX$ 3, chapitre `l3fp`);
- la `<liste de nombres>` est développée au maximum puis détokenisée avant d'être exploitée par le constructeur de l'objet tuple. Les éléments vides sont ignorés.

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`).

---

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 méthodes

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

```
\tuplexe{⟨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 données 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<sup>3</sup>, 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 `\tuplexe` 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}%
\tuplexe{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 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 = \tuplexe{nn.len}\par          1) len = 17
2) sum = \tuplexe{nn.sum}\par          2) sum = 125.5
3) min = \tuplexe{nn.min}\par          3) min = 2.9
4) max = \tuplexe{nn.max}\par          4) max = 13.6
5) mean = \tuplexe{nn.mean}\par        5) mean = 7.382352941176471
5) med = \tuplexe{nn.med}\par          5) med = 6.9
6) stdev = \tuplexe{nn.stdev}          6) stdev = 3.447460325944583
```

#### 4.2 La méthode quantile

Cette méthode a pour syntaxe

```
quantile{⟨p⟩}
```

où  $\langle p \rangle$  doit être un nombre compris entre 0 et 1. La méthode renvoie le quantile correspondant à l'argument  $\langle p \rangle$ . La méthode utilisée est celle de la moyenne pondérée<sup>4</sup>; il s'agit de l'interpolation linéaire du type « R7 » décrite dans

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

4. 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  $x_{[h]} + (h - [h])(x_{[h]} + x_{[h+1]})$ , où  $x_k$  est le  $k^{\text{e}}$  nombre du tuple ordonné.

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) \tplex{nn.quantile{0.25}}\par          1) 4.3
2) \tplex{nn.quantile{0.5}}\par          2) 6.9
2) \tplex{nn.quantile{0.75}}\par          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 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  $n - 1$  sans quoi une erreur de compilation sera renvoyée.

```
%\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 : \tplex{nn.get{0}}\par          1er élément : 12.7
13e élément : \tplex{nn.get{12}}\par          13e élément : 5.1
dernier élément : \tplex{nn.get{\tplex{nn.len}-1}}
```

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

### 4.4 La méthode `pos`

Cette méthode a pour syntaxe

```
pos{<nombre>}
```

et renvoie l'index de la première occurrence du `<nombre>` dans le tuple. Si le tuple ne contient pas le `<nombre>`, `-1` est renvoyé.

```
%\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 : \tplex{nn.pos{12.7}}\par          index de 12.7 : 0
index de 2.9 : \tplex{nn.pos{2.9}}\par          index de 2.9 : 3
index de 31.8 : \tplex{nn.pos{31.8}}\par          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 :

- 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 2<sup>e</sup> 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}%
\tplex{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 `\tplfpcmpare` 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 boîte 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}
\tplxe{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{\tplxe{nn.mean}}%
\def\tplformat#1#2{\tplfpcmpare{#2}<{\nnmean}
{\textcolor{red}{#2}}
{#2}}%
}%
\def\tplsep{~; }%
\tplxe{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 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}%
\tplxe{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 a pour syntaxe

$$\text{set}\{\langle index1 \rangle : \langle nombre1 \rangle, \langle index2 \rangle : \langle nombre2 \rangle, \dots\}$$

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

Chaque  $\langle index \rangle$  doit être compris entre 0 et  $n - 1$ , où  $n$  est le nombre d'élément du tuple passé en entrée de la méthode.

```
%\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}%
\tplxe{nn.set{1:10,5:50}}
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, 4.1
```

### 5.3 La méthode add

Cette méthode, qui ajoute une  $\langle insertion \rangle$  à un ou plusieurs index spécifiés, a pour syntaxe

$$\text{add}\{\langle index1 \rangle : \langle insertion1 \rangle, \langle index2 \rangle : \langle insertion2 \rangle, \dots\}$$

Il faut noter que dans cette syntaxe, une  $\langle insertion \rangle$  peut être

- un nombre seul «  $\text{add}\{\langle index \rangle : \langle nombre \rangle\}$  »
- une liste csv de nombres qui doit être entre accolades «  $\text{add}\{\langle index \rangle : \{n1, n2, n3, \dots\}\}$  »
- un tuple auquel on accède par `\tplxe` : «  $\text{add}\{\langle index \rangle : \{\text{tplxe}\{\langle nom tuple \rangle\}\}$  ».

En ce qui concerne les  $\langle index \rangle$  :

- si un  $\langle index \rangle$  est inférieur à 0, il est compris comme étant 0 ;
- si un  $\langle index \rangle$  est « \* » ou supérieur à  $n - 1$ , l'insertion est placée à la fin du tuple passé en entrée ;
- les  $\langle index \rangle$  ne sont pas recalculés après chaque insertion, mais seulement après la dernière. Il n'est donc pas équivalent d'écrire « .add{1:10,5:50} » et « .add{1:10}.add{5:50} ». En effet, 5 sera à l'index 6 dans le premier cas alors qu'il sera à l'index 5 dans le second.

<pre>%\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 16 : \tplexe{nn.add{16:{666,667}}}\par 3) ajout en dernier : \tplexe{nn.add{*:{666,667}}}</pre>	<pre>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 16 : 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, 666, 667, 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</pre>
--	---

<pre>\newtuple{X}{10,20,30,40,50,60}% \newtuple{Y}{1,2,3,4}% 1) \tplexe{X.add{-3:{\tplexe{Y}}}\par 2) \tplexe{X.add{1:1000,3:{\tplexe{Y}}}\par 3) \tplexe{X.add{1:1000}.add{3:{\tplexe{Y}}}}</pre>	<pre>1) 1, 2, 3, 4, 10, 20, 30, 40, 50, 60 2) 10, 1000, 20, 30, 1, 2, 3, 4, 40, 50, 60 3) 10, 1000, 20, 1, 2, 3, 4, 30, 40, 50, 60</pre>
--	--

## 5.4 La méthode op

Cette méthode, qui effectue une  $\langle opération \rangle$  sur tous les éléments du tuple, a pour syntaxe

`op{ $\langle opération \rangle$ }`

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.

<pre>\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&lt;4 ? val-1 : val+1}}</pre>	<pre>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</pre>
--	--

## 5.5 La méthode filter

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

`filter{ $\langle test \rangle$ }`

et où  $\langle test \rangle$  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.

<pre>%\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&lt;10}}\par 2) \tplexe{nn.filter{val&gt;5 &amp;&amp; val&lt;10}}\par 3) \tplexe{nn.filter{idx&lt;3    idx&gt;13}}\par 4) \tplexe{nn.filter{val!=6.3 &amp;&amp; val!=2.9 &amp;&amp; val!=4.1}}</pre>	<pre>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</pre>
---	--

## 5.6 La méthode comp

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

`comp{ $\langle opération \rangle$ }{ $\langle nom tuple \rangle$ }`

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 2<sup>e</sup> argument.

Produit de 2 tuples et leur « somprod » :

```
\newtuple{A}{2,-4,3,7,-1}%
\newtuple{B}{-9,0,4,6,-2}%
produit : \tplexex{A.comp{xa*xb}{B}}\par
somprod : \tplexex{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}%
\tplexex{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, elles 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 entrée 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  $n - 2$  où  $n$  est le nombre d'éléments du tuple passé en entrée.

```
%\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}%
\tplexex{nn.split{5}{n1}{n2}}%
tuple avant : \tplexex{n1}\par
tuple après : \tplexex{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) \tplexex{nn.sorted.filter{val>11}.store{nn1}}%
   \tplexex{nn1}\par
2) \tplexex{nn1.len.store\nnlen}%
   \meaning\nnlen\par
3) \def\tplformat#1#2{\fbox{#2}}\def\tplsep{ }%
   \tplexex{nn1.show.store\nnshow}%
   \meaning\nnshow\par
4) \edef\ndisp{\tplexex{nn1}}%
   \meaning\ndisp% 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 `\tplexex{<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 2<sup>e</sup> 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 :

<code>\gentuple{\genrule {i+1}*2 ; \until {i=9 } \par</code>	2, 4, 6, 8, 10, 12, 14, 16, 18, 20
<code>ou \par</code>	ou
<code>\gentuple{\genrule {i+1}*2 ; \while {i&lt;10 } }</code>	2, 4, 6, 8, 10, 12, 14, 16, 18, 20

Génération de 15 entiers aléatoires entre 1 et 10 :

<code>\gentuple{\genrule randint(1,10) ; \until {i=14 } }</code>	2, 2, 7, 2, 5, 2, 10, 7, 4, 6, 9, 7, 3, 4, 8
--	--

Génération des carrés d'entiers jusqu'à 500 :

<code>\gentuple{\genrule {i*i} ; \while {val&lt;500 } }</code>	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 :

<code>\gentuple{1,1,\genrule {1+\2} ; \until {i=9 } }</code>	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  $u_n = u_{n-3}u_{n-1} - u_{n-2}^2$  :

<code>\gentuple{1,1,-1,\genrule {1*\3-\2*\2} ; \until {i=10} }</code>	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 :

<code>\edef\syr{%   \gentuple{15,\genrule {1/2=trunc(1/2) ? 1/2 : 3*\1+1 ; \until {val=1}}% }% \meaning\syr</code>	macro:->15,46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
--	--

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

<code>\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}</code>	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<sup>5</sup>.

Le principal défi a surtout été d'offrir des macros développables et aussi une syntaxe originale dans le monde de  $\TeX$  du type  $\langle \text{objet} \rangle . \langle \text{méthode 1} \rangle . \langle \text{méthode 2} \rangle . \dots \langle \text{méthode } n \rangle$ , où l'on peut enchaîner 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 croît ensuite rapidement<sup>6</sup>. Ceci dit, qui utiliserait  $\TeX$  pour des calculs sur autant de nombres ?

Nombre d'éléments $n$	Durée de <code>\newtuple</code> en s
50	0,011
100	0,026
500	0,224
1000	0,703
1500	1,727
2000	2,876
5000	22,907

---

5. 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`.

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

### 8.3 Exemple : population des états

Le tuple `\Wpop` contient la population de chaque état dans le monde, en millions d'habitants<sup>7</sup> :

```
\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 : \tplexex{\Wpop.len}\par
Moyenne : \tplexex{\Wpop.mean}\par
Médiane : \tplexex{\Wpop.med}\par
Écart type : \tplexex{\Wpop.stdev}\par
Quintile \#1 : \tplexex{\Wpop.quantile{0.2}}\par
Quintile \#4 : \tplexex{\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 :

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

Répartition sur 6 intervalles égaux :

```
\begin{tabular}{lc}\hline
De 10 à 25 & \tplexex{\Wpop.filter{val<25}.len}\\
De 25 à 40 & \tplexex{\Wpop.filter{val>=25 && val<40}.len}\\
De 40 à 55 & \tplexex{\Wpop.filter{val>=40 && val<55}.len}\\
De 55 à 70 & \tplexex{\Wpop.filter{val>=55 && val<70}.len}\\
De 70 à 85 & \tplexex{\Wpop.filter{val>=70 && val<85}.len}\\
De 85 à 100 & \tplexex{\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. permettre à l'utilisateur de choisir `l3fp` ou `xint` comme moteur de calcul;
2. utiliser le tri par insertion pour trier les tuples presque triés obtenus suite aux méthodes `add`, `set`, `op` (risqué car dépend de l'opération!)
3. utiliser le tri fusion pour ajouter un tuple à un autre;
4. autres optimisation de la rapidité?

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