The package for TEX and LATEX

tuple

v0.2

2024/12/20

Christian Tellechea[*]

This extension provides common operations for tuples of numbers, in a expandable way, with a concise and easy-to-use ”`object.method`” syntax.

---

[*] unbonpetit@netc.fr

1 Preview

We consider the list of numbers (which we will now call a ”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
```

We can define an ”object”, which we name for example ”nn” with the 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}
```

This tuple ”nn” will be used throughout this documentation and recalled in comments in all the codes where it occurs.

Here is its maximal value:

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

```
1) 13.6
2) macro:->13.6
```

We can also calculate the median of the 5 smallest values, which supposes:

1. to sort the tuple (method `sorted`);
2. to retain only the values whith index 0 to 4 (method `filter`);
3. to find the median of the 5 numbers retained (method `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
```

Without LaTeX, this package is loaded with

```
\input tuple.tex
```

and under LaTeX with

```
\usepackage{tuple}
```

This package does not rely on any other except the LaTeX3 module ”`l3fp`”, which is now part of the LaTeX kernel, in particular to take advantage of its powerful macro `\fpeval`.

If we do not use LaTeX, `tuple` will load the file `expl3-generic.tex` in order to have the `l3fp` module.

2 Declaring a tuple object

The macro `\newtuple{⟨name⟩}{⟨list of numbers⟩}` allows to construct a tuple ”object” that is then accessed by its ⟨*name*⟩.

The ⟨*name*⟩ accepts all alphanumeric characters `[az][AZ][09]`, spaces and punctuation. Spaces preceding or following it are removed. A macro is also allowed[2].

The ⟨*list of numbers*⟩ is fully expanded and then detokenized before being used by the constructor of the tuple object. Empty elements are ignored.

For the sake of simplicity or speed[3], the user is given the choice of specifying the type of numbers that make up the tuple using the macro `\tplsetmode` ;

- `\tplsetmode{int}` to request that all numbers be signed integers between $-2^{31}+1$ and $2^{31}-1$. Comparisons during sorting are made with `\ifnum`, operations on numbers with `\numexpr`.
- `\tplsetmode{dec short}` specifies “short” decimals, i.e. 8-digit intpart and 8-digit decpart. In this case, comparisons are made with `\ifdim` and operations on numbers by a rudimentary calculation engine embedded within `tuple`.
- `\tplsetmode{dec long}`, which is the default mode, specifies “long” decimals in the sense of `l3fp`. In this mode, comparisons and operations on numbers are performed by `l3fp`.

Whichever mode is selected, final calculations (standard deviation, mean, quartile, etc.) are performed by `l3fp`.

---

[2] The macro will never be modified by the package `tuple`: internally, this macro is detokenized to build a more complex name of a macro.

[3] The speed differences aren’t huge, but they do exist, see page 8

It is possible to define an empty tuple (i.e. one that does not contain any numbers), but many methods will return an error if they are executed on an empty list.

On the other hand, it's impossible to redefine an existing tuple (this requires the `store` method).

3 Methods

Here is the syntaxe to execute methods on a tuple:

```
\tplexe{⟨tuple name⟩.⟨method 1⟩.⟨method 2⟩...⟨method n⟩}
```

No spaces are allowed between the “`.`” and the name of a method. It is therefore illegal to write “`.␣sorted`”.

There are 3 types of datas for the `tuple` package:

1. numbers (and displayable datas);
2. ”tuple” objects;
3. the ”storage” type which characterizes non-expandable methods performing assignments.

All methods in this package take a tuple as input (which is the result of the previous methods) and return a result whose type determines which group the method belongs to:

- group 1 ”`tuple` ⟶ `number`”;
- group 2 ”`tuple` ⟶ `tuple`”. The methods in this group *do not modify* the initial tuple[4], they act on a temporary tuple that it is obviously possible to save with a method in the group below;
- group 3 ”`tuple` ⟶ `storage`”;

The macro `\tplexe` and its argument are expandable, provided that the methods invoked are not in the group ”`tuple` ⟶ `storage`”.

If no method is specified, an expandable, implicit and generic method of the group ”`tuple` ⟶ `number`”, is executed and returns the tuple.

```
%\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 Methods of the group `tuple` ⟶ `number`

4.1 Methods `len`, `sum`, `min`, `max`, `mean`, `med` and `stdev`

All these expandable methods do not accept any argument and return respectively the number of elements, their sum, the minimum, the maximum, the mean, the median and the standard deviation.

```
%\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

4.2 Method `quantile`

This expandable method has the syntax

```
quantile{⟨p⟩}
```

---

[4] They cannot do so, otherwise they would not be expandable!

where ⟨p⟩ must be a number between 0 and 1. The method returns the quantile according to the argument ⟨p⟩. The method used is the average method[5]; this is interpolation scheme "R7" described in this article.

```
%\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
```

Note that `quantile{0.5}` is equivalent to `med`.

It's important to note that spaces before and after method arguments are ignored. It is therefore possible to write `.quantile␣{0.5}␣`.

4.3 Method `get`

This expandable method has the syntax

`get{`⟨*index*⟩`}`

The first index is 0 and the last is $n-1$ where n is the number of elements in the tuple. Therefore, the argument of `get` must be between 0 and $n-1$. You can also use negative indexes, where -1 is the index of the last element, -2 that of the penultimate element and so on up to $-n$.

```
%\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}%
1st number  : \tplexe{nn.get{0}}\par
13th number : \tplexe{nn.get{12}}\par
last number : \tplexe{nn.get{\tplexe{nn.len}-1}}\par
last number : \tplexe{nn.get{-1}}% better than above
```

```
1st number : 12.7
13th number : 5.1
last number : 4.1
last number : 4.1
```

The argument of `get` is evaluated before being used: it is therefore possible to put the expandable macro `\tplexe` with a final method returning an integer.

4.4 Method `pos`

This expandable method has the syntax

`pos{`⟨*number*⟩`}[`⟨*n*⟩`]`

and returns the index of the ⟨*n*⟩ occurrence of the ⟨*number*⟩ in the tuple. If the tuple does not contain the ⟨*number*⟩, `-1` is returned. If the optional argument is not present, ⟨n⟩ is -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 of 12.7   : \tplexe{nn.pos{12.7}}\par
index of 2.9    : \tplexe{nn.pos{2.9}}\par
index of 2.9[2]: \tplexe{nn.pos{2.9}[2]}\par
index of 2.9[3]: \tplexe{nn.pos{2.9}[3]}\par
index of 31.8   : \tplexe{nn.pos{31.8}}
```

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

4.5 Method `show`

This expandable method does not accept any arguments and is intended to convert a tuple object into a displayable result. To do this:

- for each element, the macro `\tplformat`, requiring 2 mandatory arguments, is executed. The first argument passed is the index of the element and the 2nd argument is the element itself;
- each result from the macro `\tplformat` is separated from the next by the content of the macro `\tplsep`.

By default, these two macros have the following code:

---

[5] If p is the argument of the method, we define $h=(n-1)p+1$ where n is the length of the tuple.
The method returns the number equal to $x_{\lfloor h\rfloor}+(h-\lfloor h\rfloor)(x_{\lfloor h\rfloor}+x_{\lceil h\rceil})$, where x_k is the k^{th} number of the sorted tuple.

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

The default behavior is therefore exactly the same as the implicit method that is executed last, and in particular, the method is expandable.

```
%\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

These 2 macros can be reprogrammed to create more advanced formatting. Here we use the `\tplfpcompare` macro, which is an alias of the `\fp_compare:nNnTF` macro of the `l3fp` module, to compare an element with a given value. In the 2 examples given below, the method is no longer expandable due to the use of the `\fbox` and `\textcolor` macros.

Boxing the first 10 elements:

```
%\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}
```

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

Below-average items highlighted in red:

```
%\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 Methods of the group tuple ⟶ tuple

Each time a tuple is generated or modified, the `len`, `sum`, `min`, `max`, `mean`, `med`, `stdev` and `sorted` methods are updated.

5.1 Method `sorted`

This expandable method does not accept any arguments and returns a tuple object with its elements sorted in ascending order.

```
%\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 Method `set`

The syntax of this expandable method is

`set{⟨index1⟩:⟨number1⟩,⟨index2⟩:⟨number2⟩,...}`

In the tuple resulting from the previous methods, replaces the number at ⟨*index1*⟩ with ⟨*number1*⟩ and so on if several assignments are specified in a comma-separated list.

Each ⟨*index*⟩ must be between 0 and $n-1$, where n is the number of elements in the tuple passed as input to the method. Negative indexes from $-n$ to -1 are also permitted.

```
%\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 Method `add`

This expandable method, which adds a ⟨*insertion*⟩ at one or more specified indexes, has the syntax

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

It should be noted that in this syntax, a ⟨*insertion*⟩ can be

- a single number ”`add{⟨index⟩:⟨number⟩}`”
- a csv list of numbers that *must* be enclosed in braces ”`add{⟨index⟩:{n1,n2,n3...}}`”
- a tuple accessed by `\tplexe`: ”`add{⟨index⟩:{\tplexe{⟨tuple name⟩}}}`”.

As for ⟨*index*⟩, they can be between 0 and n, where n is the number of elements in the tuple. Negative indexes between $-n-1$ and -1 are also permitted:

- a ⟨*index*⟩ equal to 0 or $-n-1$ places the ⟨*insertion*⟩ at the beginning of the tuple passed as input;
- a ⟨*index*⟩ equal to n or -1 places the ⟨*insertion*⟩ at the end of the tuple;
- the ⟨*index*⟩ *are not* updated after each ⟨*insertion*⟩, but only after the last one. It is therefore not equivalent to write “`.add{1:100,2:200}`” and “`.add{1:100}.add{2:200}`”. Indeed, 200 will be at index 3 in the first case, whereas it will be at index 2 in the second.

```
%\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) add at first pos: \tplexe{nn.add{0:{666,667}}}\par
2) add index 11    : \tplexe{nn.add{11:{666,667}}}\par
3) add at last pos : \tplexe{nn.add{-1:{666,667}}}
```

```
1) add at first pos: 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) add 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) add at last pos : 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 Method `op`

This expandable method, which performs an ⟨*operation*⟩ on all elements of the tuple, has the syntax

`op{⟨operation⟩}`

The ⟨*operation*⟩ is an expression not containing braces, evaluable by `\fpeval` once all occurrences of ”`val`” have been replaced by the value of each element, and all occurrences of ”`idx`” by its 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 Method `filter`

Thi expandables method, which selects elements according to one or more criteria, has the syntax

`filter{⟨test⟩}`

and where ⟨*test*⟩ is a boolean not containing braces, evaluable by `\fpeval` once all occurrences of ”`val`” have been replaced by the value of each element, and all occurrences of ”`idx`” by its 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 Method `comp`

This expandable method composes two tuples of the same length with an operation that the user specifies. Its syntax is

comp{⟨*operation*⟩}{⟨*tuple name*⟩}

where the tuple whose name is passed as the second argument *must* have the same length as the tuple passed as input to the method.

The ⟨*operation*⟩ is an expression not containing braces, evaluable by `\fpeval` once all occurrences of "`xa`" have been replaced by the value of each element of the input tuple, and all occurrences of "`xb`" by that of the tuple specified in the 2nd argument.

Product of 2 tuples and their "`sumprod`":

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

product -18, -0, 12, 42, 2
sumprod: 38

Calculation of the smallest distance to point A(2.5 ; -0.5) knowing the list of abscissas and the list of ordinates of a trajectory (here elliptical):

```
\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 Methods in the group `tuple` ⟶ `storage`

As these methods don't return a result because they perform an assignment, they are not expandable, and must be placed in the last position. If this is not the case, all methods following them will be ignored.

6.1 Method `split`

This method cuts the tuple passed as input to the method after the specified index. The syntax is

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

The tuple passed as input to the method is split after the ⟨*index*⟩: the part before the split is assigned, via newtuple| to the tuple with name "`tuple1`" and the remaining part to the tuple with name "`tuple2`". No check is made on the existence of the 2 tuples, so existing tuples can be silently replaced.

The ⟨*index*⟩ must lie between 0 and $n-2$ if positive, or between $-n$ and -2 if negative, n being the number of elements in the tuple passed as input.

```
%\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 before: \tplexe{n1}\par
tuple after : \tplexe{n2}
```

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

6.2 Method `store`

This macro is used to store the result of the last method. If this result is a tuple, the syntax is

store{⟨*tuple name*⟩}

and if the result is a number or a displayable data from the show method:

store{⟨*macro*⟩}

No check is made on the existence of the tuple or macro. It is therefore possible to silently replace an existing tuple or macro.

```
%\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% generic method
```

```
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
```

To store the result of the generic method, you have to use `\edef` because `\tplexe{⟨tuple⟩.store⟨macro⟩}` is incorrect since in this case, the `store` method applies to a tuple.

7 Tuple generation

To generate a tuple, we can use the expandable macro `\gentuple` which is intended to be called in the 2nd argument of `\newtuple`. Its syntax is

`\gentuple{⟨initial values⟩},\genrule⟨generation rule⟩;\while||\until⟨condition⟩}`

where:

- the `⟨initial values⟩` are optional. If present, they *must* be followed by a comma. The macro `\gentuple` determines their number *i* by counting (*i* must be at most equal to 9). These initial values will be copied at the beginning of the tuple and subsequently, are intended to be used in the `⟨generation rule⟩` for recurrence purposes;
- the `⟨generation rule⟩` is an expression not containing braces, evaluable by `\fpeval` once in the previous *i* values, all occurrences of `\1` have been replaced by the first value, `\2` by the second value, etc. In addition, each occurrence of `\i` is replaced by the value of the current index.
- the `⟨condition⟩` is a boolean not containing braces, evaluable by `\fpeval` once all occurrences of ”`val`” have been replaced by the value of the computed element, and all occurrences of ”`\i`” by its index. If the keyword after `;` is `\while`, the loop is of the type `while⟨condition⟩...endwhile` whereas if this keyword is `\until`, it is a `repeat...until⟨condition⟩` loop.

Generating the first 10 even integers:

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

```
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
or
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

Generation of 15 random integers between 1 and 10:

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

```
5, 3, 2, 5, 4, 7, 6, 4, 3, 5, 5, 8, 7, 1, 3
```

Generate squares of integers up to 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
```

Generation of the first 10 terms of the Fibonacci sequence:

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

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

Generation of the first 10 terms of $u_0 = 1$; $u_1 = 1$; $u_2 = -1$ and $u_n = u_{n-3}u_{n-1} - u_{n-2}^2$:

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

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

Generation of the Syracuse sequence (aka the "$3n + 1$ sequence") of 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

Maximum altitude and length of the Syracuse sequence of 27:

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

length = 112

max alt = 9232

8 Conclusion

8.1 Motivation

Somewhat surprisingly, very little exists to manipulate and operate on lists of numbers[6].

The main challenge was to provide expandable macros and also an original syntax in the world of TEX of the type ⟨*object*⟩`.`⟨*method 1*⟩`.`⟨*method 2*⟩`...`⟨*method n*⟩, where you can chain methods executed on an "object" which is a tuple of numbers. I don't know if any packages offer this kind of syntax, but it's actually quite intuitive. As I'm fairly unfamiliar with programming expandable macros, I almost gave up many times. But I've finally found a mouse-hole that makes it all work pretty much, except for the numerous bugs that are probably lurking everywhere.

I'd like to thank anyone who finds one for pointing it out to me by e-mail, or even suggesting new features.

8.2 Execution speed

When you get into optimizing execution speed, especially for expandable macros, you don't get out! It's a pit of questions about macro arguments, about little — or big — tricks that save time, and a headache about how to juggle with delimited arguments and find them later!

I hope I didn't fall into this trap, because I've barely entered it. In any case, TEX is not made for massive calculations, since it is first and foremost a typesetting software. For calculations, powerful tools that beat TEX hands down exist in abundance.

In any case, ”expandable macro” goes a bit against ”execution speed”. For information, I put below the compilation times, in seconds, of the creation of tuples containing n random integers. It depends on the computer used, of course, but the orders of magnitude are quite revealing. We can clearly see that it is illusory to exceed a thousand numbers because the time to create a tuple then increases rapidly[7]. That said, who would use TEX for calculations on so many numbers?

This table illustrates the creation speeds of a tuple of random signed integers between -1000 and 1000. As a reminder, creating a tuple involves sorting its elements (by quick sort), calculating the sum of the numbers and the sum of squares. The 3 modes can thus be compared for larger or smaller tuples.

Nb items	`int`	`dec:short`	`dec:long`
25	0.001 s	0.001 s	0.006 s
50	0.002 s	0.004 s	0.016 s
100	0.005 s	0.008 s	0.031 s
200	0.019 s	0.027 s	0.082 s
400	0.082 s	0.103 s	0.227 s
800	0.427 s	0.513 s	0.769 s
1600	3.349 s	3.939 s	4.355 s

---

[6] There is one package, `commalists-tools`, but it's clearly too limited. Like all its author's packages that flood CTAN, it simply aligns, linearly and without any real programmation, the high-level macros of the packages `tikz`, `listofitems`, `xstring`, `xint` and `simplekv`.

[7] Not to mention that, in addition to that, the tuple is immediately recreated and recalculated after being modified by the methods `set`, `add`, `op`, `filter`, `split` and `comp`

8.3 Example: state population

The tuple `\Wpop` contains the population of each state in the world, in millions of inhabitants[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}
Number of state: \tplexe{\Wpop.len}\par
Mean: \tplexe{\Wpop.mean}\par
Median: \tplexe{\Wpop.med}\par
Standard deviation: \tplexe{\Wpop.stdev}\par
Quintile \#1 : \tplexe{\Wpop.quantile{0.2}}\par
Quintile \#4 : \tplexe{\Wpop.quantile{0.8}}
```

Number of state: 204
Mean: 39.66338235294118
Median: 7.6
Standard deviation: 146.8985787142461
Quintile #1 : 0.8
Quintile #4 : 37.62

We modify the tuple `\Wpop`, retaining only "moderately" populated states. We arbitrarily consider their population between 10 and 100 millions:

```
\tplexe{\Wpop.filter{val>=10 && val<=100}.store\Wpop}%
Number: \tplexe{\Wpop.len}\par
Mean: \tplexe{\Wpop.mean}\par
Median: \tplexe{\Wpop.med}\par
Standard deviation: \tplexe{\Wpop.stdev}\par
Quintile \#1 : \tplexe{\Wpop.quantile{0.2}}\par
Quintile \#4 : \tplexe{\Wpop.quantile{0.8}}

Distribution over 6 equal intervals:\par
\begin{tabular}{lc}\\\hline
    From 10 to 25 & \tplexe{\Wpop.filter{val<25}.len}\\
    From 25 to 40 & \tplexe{\Wpop.filter{val>=25 && val<40}.len}\\
    From 40 to 55 & \tplexe{\Wpop.filter{val>=40 && val<55}.len}\\
    From 55 to 70 & \tplexe{\Wpop.filter{val>=55 && val<70}.len}\\
    From 70 to 85 & \tplexe{\Wpop.filter{val>=70 && val<85}.len}\\
    From 85 to 100& \tplexe{\Wpop.filter{val>=85}.len}\\\hline
\end{tabular}
```

Number: 79
Mean: 32.33037974683544
Median: 26.7
Standard deviation: 21.22899470798109
Quintile #1 : 12.6
Quintile #4 : 48.26
Distribution over 6 equal intervals:

From 10 to 25	38
From 25 to 40	19
From 40 to 55	10
From 55 to 70	7
From 70 to 85	2
From 85 to 100	3

8.4 TODO list

To be implemented more or less quickly:

1. insertion sorting to sort almost-sorted tuples obtained using the `add`, `set`, `op` methods (risky as it depends on the operation!).
2. merge sorting to add one tuple to another;
3. other speed optimization?

---

[8] The data comes from https://www.unfpa.org/data/world-population-dashboard