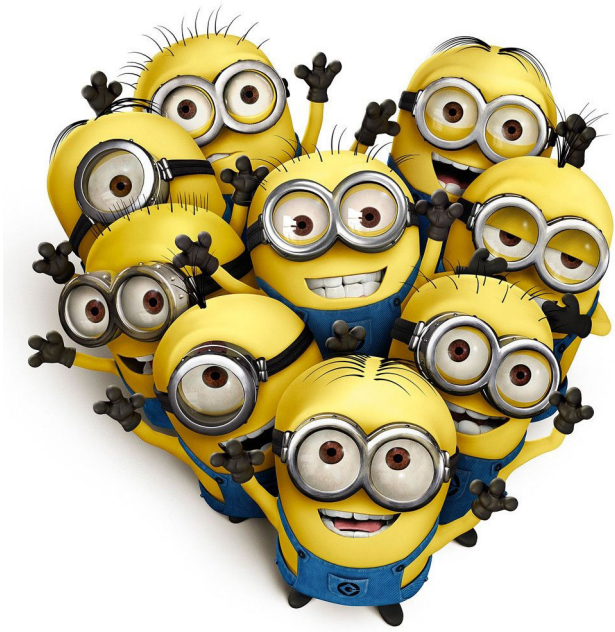
Vectorization de code Python avec Pythran

Serge Guelton, Joël Falcou, Pierrick Brunet

avec l'aimable collaboration de Mehdi Amini, Eliott Coyac, Yuancheng Pang, Adrien Merlini et Alan Raynaud

```
from IPython.core.display import Image
Image('team.jpg', width=1000)
```



```
Après un gros effort, et moult développement,

Nous avons essayé, c'est vraiment étonnant,

D'écrire, de publier, de nombreux beaux papiers,

Et quel enchantement, ils furent tous acceptés !
```

- Pythran: Enabling Static Optimization of Scientific Python Programs *S. Guelton, P. Brunet, A. Raynaud, A. Merlini, M. Amini*, Scipy2013
- Compiling Python Modules to Native Parallel Modules Using Pythran and OpenMP Annotations, *S. Guelton, P. Brunet, M. Amini*, PyHPC2013

```
%%file euler06c.c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <sys/time.h>

int main(int argc, char *argv[])
{
    if(argc !=2) return 1;
    else
```

```
   {
        struct timeval beg, end;
        int64_t n = atoi(argv[1]),
                  s0= 0,
                  s1= 0;
        gettimeofday(&beg, NULL);
        for(int64_t i = 0; i< n; i++)
        {
              s0 += i *i ;
              s1 += i;
        }
        gettimeofday(&end, NULL);
        printf("%" PRId64 "\n", s0 - s1 * s1);
        fprintf(stderr, "%ld\n",
                  (end.tv_sec - beg.tv_sec) * 1000000 + (end.tv_usec - beg.tv_usec));
        return 0;
    }
}
```

```
Overwriting euler06c.c
```

```
!make euler06c CFLAGS="-O3 -std=c99"
```

```
cc -O3 -std=c99    euler06c.c   -o euler06c
```

```
!./euler06c 50000
```

```
-1562395835208325000
44
```

```
!for i in `seq 1 10` ; do ./euler06c 50000 > /dev/null ; done 2>&1 | numaverage
```

```
41.5
```

Brève incursion dans Python

```
def euler06(n):
    r = range(n)
    return sum(x * x for x in r) - sum(r) ** 2
```

```
euler06(50000)
```

```
-1562395835208325000
```

```
%timeit euler06(50000)
```

```
100 loops, best of 3: 4.03 ms per loop
```

```
import numpy as np
```

```
def np_euler06(n):
    r = np.arange(n, dtype=np.uint64)
    return np.sum(r*r) - np.sum(r) **2
```

```
np_euler06(50000)
```

```
-1.5623958352083249e+18
```

```
%timeit np_euler06(50000)
```

```
10000 loops, best of 3: 152 µs per loop
```

Pythran à la rescousse !

```
%%file euler06p.py
#pythran export euler06(int)
#pythran export np_euler06(int)
def euler06(n):
    r = range(n)
    return sum(x * x for x in r) - sum(r) ** 2

import numpy as np
def np_euler06(n):
    r = np.arange(0,n, 1,np.int64)
    return np.sum(r*r) - np.sum(r) **2
```

```
Overwriting euler06p.py
```

```
!pythran euler06p.py
```

```
from euler06p import euler06, np_euler06
```

```
euler06(50000), np_euler06(50000)
```

```
(-1562395835208325000, -1562395835208325000)
```

```
%timeit euler06(50000)
```

```
10000 loops, best of 3: 88.8 µs per loop
```

```
%timeit np_euler06(50000)
```

```
10000 loops, best of 3: 56.4 µs per loop
```

Boucles implicites = vectorisation facile

```
import numpy as np
x = np.random.rand(1000000)
```

```
%timeit np.sum(100 * (x[1:] - x[:-1] ** 2) ** 2 + (1 - x[:-1]) ** 2)
```

```
100 loops, best of 3: 14.5 ms per loop
```

```
%%file test_numpy.py
#pythran export rosen(float[])
import numpy as np
def rosen(x):
    t0 = 100 * (x[1:] - x[:-1] ** 2) ** 2
    t1 = (1 - x[:-1]) ** 2
    return np.sum(t0 + t1)
```

```
Overwriting test_numpy.py
```

```
!pythran -O2 test_numpy.py
```

```
from test_numpy import rosen
%timeit rosen(x)
```

```
100 loops, best of 3: 2.91 ms per loop
```

```
!pythran -msse4.2 -DUSE_BOOST_SIMD -O2  test_numpy.py -o test_vnumpy.so
```

```
import test_vnumpy
%timeit test_vnumpy.rosen(x)
```

```
1000 loops, best of 3: 1.29 ms per loop
```

Encore plus de boucles implicites !

```
import numpy as np
n = 100000
r0, r1 = np.random.rand(n), np.random.rand(n)
v0, v1 = np.random.rand(4), np.random.rand(4)
```

```
def foo(x, y):
    return 3 * x + y
```

```
foo(4, 2), foo(4., 2), foo("4", "2"), foo([4], [2]), foo(v0,v1)
```

```
(14,
 14.0,
 '4442',
 [4, 4, 4, 2],
 array([ 3.28297224,  2.69871158,  2.90063754,  3.38314177]))
```

```
def foo(x): return 5 * x + 3
%timeit map(foo, r0)
```

```
10 loops, best of 3: 160 ms per loop
```

```
%%file test_map.py
#pythran export test(float [])
def test(v):
    l = vmap(lambda x: 5 * x + 3, v)
    return l[len(l)/2]
```

```
Overwriting test_map.py
```

```
!pythran test_map.py
```

```
import test_map
%timeit test_map.test(r0)
```

```
10000 loops, best of 3: 152 µs per loop
```

```
!pythran test_map.py -DUSE_BOOST_SIMD -march=native -O2 -o test_vmap.so
```

```
import test_vmap
%timeit test_vmap.test(r0)
```

```
10000 loops, best of 3: 67.1 µs per loop
```

Sélection automatique `vmap` vs. `map` à l'aide d'un *type trait* `vectorisable`

La suite au prochain numéro...

- Réflexion pour élargir l'inférence de `map` -> `vmap`
- Généraliser aux tableaux multidimensionnels avec coupe
- Généraliser le critère de vectorisabilité d'une fonction

- Réflexion sur `reduce`
- Basculer de AVX à SSE pour les entiers

```
Image('wolvie.jpg', width=1000)
```