The two language problem in vectorized python expressions: Memory or Speed.

David Buchaca Prats

Definition

Vectorized code

- Vectorized code definition: In the context of high-level languages like Python, Matlab, and R, the term vectorization describes the use of optimized, pre-compiled code written in a low-level language (e.g. C) to perform mathematical operations over a sequence of data.

- In contrast, non vectorized code implies explicit loop traversal over data.

The hypothesis

You have to choose a pill

- Python coders usually face the dichotomy:
  - A) Write code that runs faster but is memory hungry (vectorized).
  - B) Write code that runs slower but is memory efficient (non vectorized).
- This dichotomy is often found when:
  - A) Your code is vectorized.
  - B) Your code traverses data with python iterators (for/while) loops and/or depends on applying different code depending on properties of the data.

The self told little lies (I)

- Who cares about memory?
  - Memory is cheap these days!
    - There is a point at which increasing more memory is VERY expensive (motherboards cost much more, memory modules cost much more).
  - I don't care about memory as long as my code is easy to write, easy to maintain and fast.
    - What happens when the amount of data that your code has to process is 10x bigger?

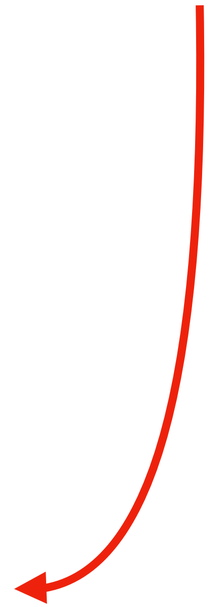
The self told little lies (II)

- Only in the extreme cases memory matters, if I allocate just hundreds of MB I don´t care, we have GB now.
  - This is partially true if now that can buy 64 GB of ram for 200 euros, but...
  - Even if you don't care about the monetary cost, there is the compute cost.
    - If you could run the same code but that uses only 1 MB of memory, it could potentially be much faster.
      - In many computations data movement (read and write) from and to memory takes much more time than the time needed for compute instructions.

Latency Comparison Numbers (~2012)

L1 cache reference	0.5 ns
Execute "standard" instruction	1 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns (3 us)
Send 1K bytes over 1 Gbps network	10,000 ns (10 us)
Read 4K randomly from SSD (1GB/sec)	150,000 ns (150 us)
Read 1 MB sequentially from memory	250,000 ns (250 us)

You might do many operations for the cost of reading 1 MB from memory



Example 1:
Reduce operation with array that does not need to be allocated.

Example 1: vectorized operation

Time to allocate the array is takes 60% of the compute time

```
%%file my_file.py
import numpy as np

def vectorized_sum_allocating_array(n=1_000_000):
    x = np.arange(n)
    return np.sum(x)
```

```
Overwriting my_file.py
```

```
from my_file import vectorized_sum_allocating_array

%timeit vectorized_sum_allocating_array(1_000_000)
```

```
316 µs ± 3.12 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

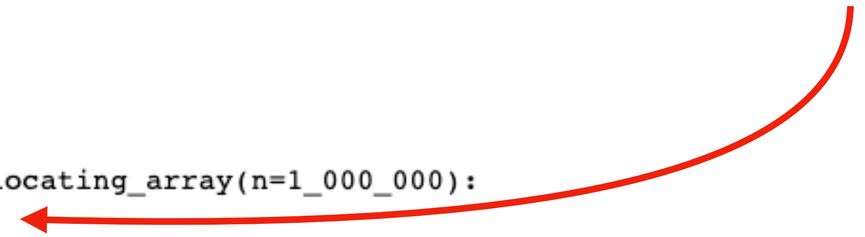
```
%mprun -T mprof0 -f vectorized_sum_allocating_array vectorized_sum_allocating_array(n=1_000_000)
print(open('mprof0', 'r').read())
```

```
%timeit x = np.arange(1_000_000)
```

```
191 µs ± 968 ns per loop (mean ± s
```

```
*** Profile printout saved to text file mprof0.
Filename: /Users/dbuchaca/personal/presentation/my_file.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     3    167.2 MiB    167.2 MiB           1   def vectorized_sum_allocating_array(n=1_000_000):
     4    174.9 MiB      7.7 MiB           1       x = np.arange(n)
     5                                             return np.sum(x)
```



Example 1: vectorized operation without alloc

Even if we do not pay the time time to allocate it is not worth a python iterator

```
%%file my_file.py
import numpy as np

def vectorized_sum_allocating_array(n=1_000_000):
    x = np.arange(n)
    return np.sum(x)
```

```
Overwriting my_file.py
```

```
from my_file import vectorized_sum_allocating_array

%timeit vectorized_sum_allocating_array(1_000_000)
```

```
316 µs ± 3.12 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops ea
```

```
%mprun -T mprof0 -f vectorized_sum_allocating_array vectorized_sum_al
print(open('mprof0', 'r').read())
```

```
*** Profile printout saved to text file mprof0.
Filename: /Users/dbuchaca/personal/presentation/my_file.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     3    167.2 MiB    167.2 MiB           1   def vectorized_sum_all
     4    174.9 MiB      7.7 MiB           1       x = np.arange(n)
     5                                             return np.sum(x)
```

```
%%file my_file3.py
import numpy as np

def vectorized_sum_over_iterator(n=1_000_000):
    x = range(n)
    return np.sum(x)
```

```
Writing my_file4.py
```

```
from my_file3 import vectorized_sum_over_iterator

%timeit vectorized_sum_over_iterator(1_000_000)
```

```
62.2 ms ± 416 µs per loop (mean ± std. dev. of 7 runs, 10 loops e
```

```
%mprun -T mprof0 -f vectorized_sum_over_iterator vectorized_sum_o
print(open('mprof0', 'r').read())
```

```
*** Profile printout saved to text file mprof0.
Filename: /Users/dbuchaca/personal/presentation/my_file3.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     3    517.5 MiB    517.5 MiB           1   def vectorized_sum
     4    517.5 MiB      0.0 MiB           1       x = range(n)
     5    517.5 MiB      0.0 MiB           1       return np.sum(
```

Example 1: devectorized operation

Faster than using a vectorized operation if using an iterator

```python
def sum_up_to_n(n:int):
    s = 0
    for i in range(n):
        s += i
    return s
```

```
Overwriting my_file2.py
```

```python
from my_file2 import sum_up_to_n
%timeit sum_up_to_n(1_000_000)
```

```
49.5 ms ± 642 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```python
%mprun -T mprof0 -f sum_up_to_n sum_up_to_n(n=1_000_000)
print(open('mprof0', 'r').read())
```

```
*** Profile printout saved to text file mprof0.
Filename: /Users/dbuchaca/personal/presentation/my_file2.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     2    167.1 MiB    167.1 MiB           1   def sum_up_to_n(n:int):
     3    167.1 MiB      0.0 MiB           1       s = 0
     4    167.1 MiB      0.0 MiB     1000001       for i in range(n):
     5    167.1 MiB      0.0 MiB     1000000           s += i
     6    167.1 MiB      0.0 MiB           1       return s
```

Example 1

```python
def vectorized_sum_allocating_array(n=1_000_000):
    x = np.arange(n)
    return np.sum(x)
```

```
Overwriting my_file.py
```

```python
from my_file import vectorized_sum_allocating_array

%timeit vectorized_sum_allocating_array(1_000_000)
```

```
316 µs ± 3.12 µs per loop (mean ± std. dev. of 7 runs,
```

- Vectorized ✅
- Fastest 🚀
- Bad in terms of memory

```python
def sum_up_to_n(n:int):
    s = 0
    for i in range(n):
        s += i
    return s
```

```
Overwriting my_file2.py
```

```python
from my_file2 import sum_up_to_n
%timeit sum_up_to_n(1_000_000)
```

```
49.5 ms ± 642 µs per loop (mean ± std. d
```

- Vectorized ❌
- Slower
- Good in terms of memory

```python
def vectorized_sum_over_iterator(n=1_000_000):
    x = range(n)
    return np.sum(x)
```

```
Writing my_file4.py
```

```python
from my_file3 import vectorized_sum_over_iterator

%timeit vectorized_sum_over_iterator(1_000_000)
```

```
62.2 ms ± 416 µs per loop (mean ± std. dev. of 7 r
```

- Vectorized ✅
- Slowest
- Good in terms of memory

The hypothesis

You have to choose a pill

- A) Write code that runs faster but is memory hungry.

```
import numpy as np
%timeit np.sum(np.arange(1_000_000))

320 µs ± 3.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- B) Write code that runs slower but is memory efficient.

```
def sum_up_to_n(n:int):
    s = 0
    for i in np.arange(n):
        s += i
    return s

%timeit sum_up_to_n(1_000_000)

49.3 ms ± 732 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Example 1: Cython

Basic cython syntax

- In a notebook one can compile cython code using %%cython at the top of a cell.
- Note variables should be typed

```
def sum_up_to_n(n=1_000_000):
    s = 0
    for i in range(n):
        s +=i
    return s
```

```
%%cython

cpdef long sum_up_to_n_cython(long n):
    cdef:
        long s = 0
        long i

    for i in range(n):
        s +=i
    return s
```

Example 1: Cython

```
%%cython

cpdef long sum_up_to_n_cython(int n):
    cdef:
        long s = 0
        long i

    for i in range(n):
        s +=i
    return s
```

```
%timeit sum_up_to_n_cython(1_000_000)
```

```
47.7 ns ± 0.839 ns per loop (mean ± std. dev.
```

- Vectorized ❌
- Fastest !
- Good in terms of memory

```
def vectorized_sum_allocating_array(n=1_000_000):
    x = np.arange(n)
    return np.sum(x)
```

```
Overwriting my_file.py
```

```
from my_file import vectorized_sum_allocating_array

%timeit vectorized_sum_allocating_array(1_000_000)
```

```
316 µs ± 3.12 µs per loop (mean ± std. dev. of 7 runs,
```

- Vectorized ✅
- Not Fastest
- Bad in terms of memory

```
def sum_up_to_n(n:int):
    s = 0
    for i in range(n):
        s += i
    return s
```

```
Overwriting my_file2.py
```

```
from my_file2 import sum_up_to_n
%timeit sum_up_to_n(1_000_000)
```

```
49.5 ms ± 642 µs per loop (mean ± std. d
```

- Vectorized ❌
- Slower
- Good in terms of memory

Example 1: Cython

Cython code might not be compatible with standard python code

- Note `sum_up_to_n_cython` is not valid python code. One needs to compile it to use it in any python program.

```
%%cython

cpdef long sum_up_to_n_cython(long n):
    cdef:
        long s = 0
        long i

    for i in range(n):
        s +=i
    return s
```

- Cython 3.0 allows "pure python mode"
  - Pure python mode allows to write valid python code that the user can decide to compile the code or not

Example 1: Pure python mode

Pure python type code is 100% compatible with python

- Instead of using **def func(n:int)** you use **def func(n:cython.int)**

```
def sum_up_to_n_cython_pure_python(n: cython.long):
    s: cython.long = 0
    n: cython.long
    i: cython.long

    for i in range(n):
        s +=i
    return s
```

```
sum_up_to_n_cython_pure_python(1_000_000)
```

```
499999500000
```

```
%timeit sum_up_to_n_cython_pure_python(1_000_000)
```

```
50.2 ms ± 445 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%cython

import cython

def sum_up_to_n_cython_pure_python(n: cython.long):
    s: cython.long = 0
    n: cython.long
    i: cython.long

    for i in range(n):
        s +=i
    return s
```

```
sum_up_to_n_cython_pure_python(1_000_000)
```

```
499999500000
```

```
%timeit sum_up_to_n_cython_pure_python(1_000_000)
```

```
49.3 ns ± 0.729 ns per loop (mean ± std. dev. of 7 r
```

over 9000x faster !!

Example 1: Julia and Cython

The compiler is smart enough to make the execution time constant

```
julia> function sum_up_to_n(n)
          res = 0
          for i in 1:n
             res+=i
          end
          return res
       end
sum_up_to_n (generic function with 1 method)

julia> sum_up_to_n(100_000_000)
5000000050000000

julia> @benchmark sum_up_to_n(100_000_000)
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.541 ns … 18.375 ns  | GC (min … max): 0.00% … 0.00%
 Time  (median):     1.625 ns              | GC (median):    0.00%
 Time  (mean ± σ):   1.631 ns ±  0.270 ns  | GC (mean ± σ):  0.00% ± 0.00%

 1.54 ns      Histogram: log(frequency) by time      1.96 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

- Vectorized ❌
- Fastest !
- Good in terms of memory

```
: %%cython

cpdef long sum_up_to_n_cython(long n):
    cdef:
        long s = 0
        long i

    for i in range(n):
        s +=i
    return s
```

```
ld: warning: duplicate -rpath '/Users/dbucha
```

```
: %timeit sum_up_to_n_cython(1_000_000)
```

```
45.5 ns ± 0.431 ns per loop (mean ± std. dev
```

```
: %timeit sum_up_to_n_cython(100_000_000)
```

```
45.7 ns ± 0.424 ns per loop (mean ± std. dev
```

- Vectorized ❌
- Fast!
- Good in terms of memory

Example 2:
Computing distances between a query vector and a collection of vectors

Example 2: Vectorized Solution

Distance between a vector of 10 elements and 1 million other vectors

```
n_samples = 1_000_000
n_queries = 1
n_features = 10

query_vector = np.random.random((1, n_features))
X = np.random.random((n_samples,n_features))
X.shape, query_vector.shape
```

```
((1000000, 10), (1, 10))
```

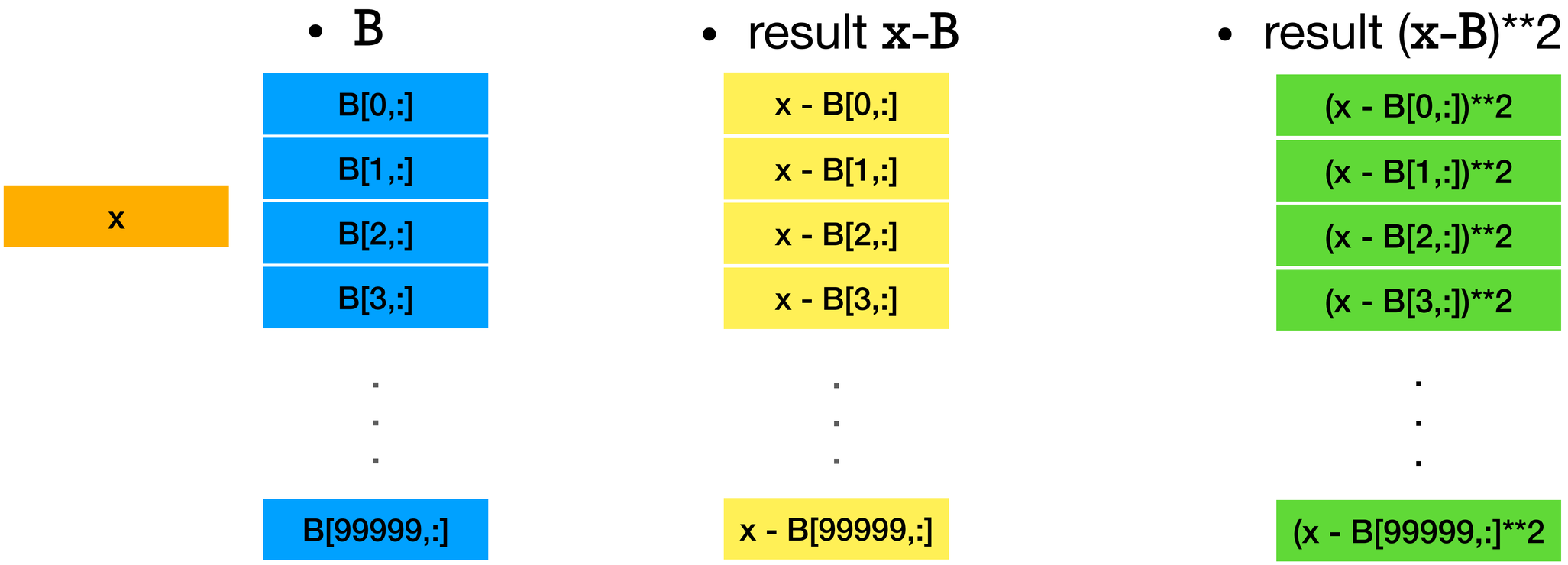
```
def euclidean_naive(x,B):
    return np.sqrt(np.sum((x-B)**2,axis=1))
```

```
%timeit euclidean_naive(query_vectors, X)
```

```
24.8 ms ± 440 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Example 2: Vectorized Solution "hidden" problems

```
def euclidean_naive(x,B):
    return np.sqrt(np.sum((x-B)**2,axis=1))
```



Example 2: Vectorized Solution "hidden" problems

```
def euclidean_naive(x,B):
    return np.sqrt(np.sum((x-B)**2,axis=1))
```

- `x-B` creates an array of size 1000_000 x 10
  - Then this array is Garbaged Collected
- `(x-B)**2` creates another array of size 1000_000 x 10
  - Then this array is Garbaged Collected
- np.sum(*, axis=1) creates a vector of size 1000_000
  - Then this array is Garbaged Collected

Example 2: Cython Solution

```
%%cython
import numpy as np
import cython
from libc.math cimport pow, sqrt

@cython.boundscheck(False)  # Deactivate bounds checking
def cy_euclidean(double[:,:] q,double[:,:] X):
    cdef int n_samples = X.shape[0]
    cdef int n_features = q.shape[1]
    cdef double res=0
    cdef double[:] result = np.zeros(len(X), dtype="double")

    for m in range(n_samples):
        res = 0.
        for i in range(n_features):
            res += pow(q[0,i]- X[m, i],2)
        result[m] = sqrt(res)

    return np.array(result)
```

(x - B[0,0])**2 + (x - B[0,1])**2 + ... (x - B[0,10])**2

(x - B[1,0])**2 + (x - B[1,1])**2 + ... (x - B[1,10])**2

(x - B[2,0])**2 + (x - B[2,1])**2 + ... (x - B[2,10])**2

(x - B[-1,0])**2 + (x - B[-1,1])**2 + ... (x - B[-1,10])**2

```
cy_time = %timeit -o cy_euclidean(query_vectors, X)
```

```
4.52 ms ± 48.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Doing Nothing is better than being busy doing nothing

Lao Tsu

VECTORIZE NOOB