

Algorithmic complexity and graphs: complexities

22 septembre 2024

Complexity

- ▶ Today we will **quantify** the **complexity** of several problems :
how many operations are required in order to answer a given question, as a function of the size of the input ?

Complexity

- ▶ Today we will **quantify** the **complexity** of several problems : how many operations are required to answer a given question, as a function of the size of the input ? Is it possible to **compute** an answer with a computer ?
- ▶ Importantly, this is called the **time complexity** of the problem. It does not take the memory usage into account.
- ▶ However, we will also (shortly) discuss **space complexity**, that quantifies memory usage.

Complexity

- ▶ The answer is that **it depends on the problem**. For some problems, it is very likely that there exists **no exact "fast" (polynomial)** solution (for instance the NP-hard problems)

Average and worst case complexities

- ▶ Often, for a given algorithm, the exact number of operations needed will **depend on the instance of the problem**.
- ▶ It is possible to compute several complexities given a problem size n :
 - ▶ **worst-case** the maximum number of operation needed
 - ▶ **average-case** average complexity, averaged over a **distribution** on the input. Thus this distribution is to be known, or assumed.

- └ The problem of complexity
- └ Measuring time complexities

Measuring complexities

- ▶ Let us start by measuring the complexity of some simple programs.
- ▶ We can first measure the computing time.

Measuring execution times

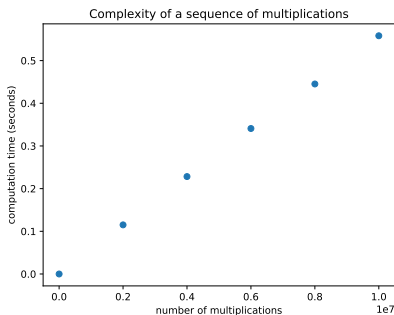
Exercice 1 : Linear complexity

- ▶ **cd complexity/** and use **linear_complexity.py** to verify that the complexity of a sequence of multiplications is proportionnal to its length.

Measuring execution times

Exercise 1 : Linear complexity

- ▶ `cd complexity/` and use `linear_complexity.py` to verify that the complexity of a sequence of multiplications is proportionnal to its length.
- ▶ It should look like this :



Measuring execution times

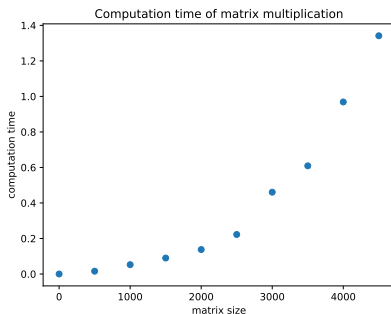
Exercise 2 : Non linear complexity

- ▶ What happens with matrix multiplication ? Use **matrix_multiplication.py** to estimate the computing time as a function of the size of the matrix.

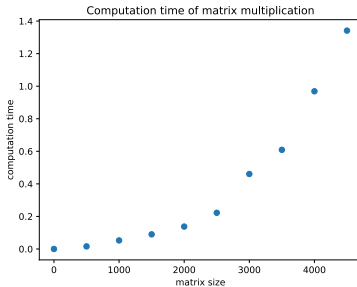
Measuring execution times

Exercise 2 : Non linear complexity

- What happens with matrix multiplication ? Use **matrix_multiplication.py** to estimate the computing time as a function of the size of the matrix.
- It should look like this :

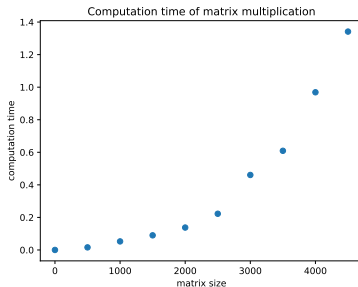


Matrix multiplication



- ▶ Let's give a rough approximation of the number of operations as a function of the size n of the matrix.

Matrix multiplication



- ▶ Matrix multiplication is of order $\mathcal{O}(n^3)$.
- ▶ However, some **sub-cubic** algorithms exists : n^t with $t < 3$.
But with very large constants (notion of **galactic algorithms** : https://en.wikipedia.org/wiki/Galactic_algorithm)

- └ The problem of complexity
- └ Measuring time complexities

timeit

For a measurement of the execution time, timeit is also available :
<https://docs.python.org/3/library/timeit.html>

Measuring the time ?

- ▶ The measured time is a valid measure of the runtime, and can be useful to compare different algorithms on a given machine.
- ▶ However, it directly depends on the machine
- ▶ We could count the number of elementary operations instead.

- └ The problem of complexity
- └ Measuring time complexities

Experimental evaluation

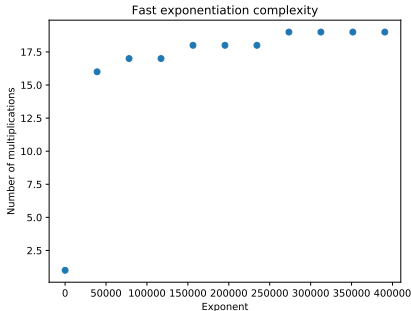
Exercise 3 : Counting the number of elementary operations

- ▶ Please use a variable in **exponentiation_complexity.py** to compute the number of operations in fast exponentiation.

Experimental evaluation

Exercise 3 : Counting the number of elementary operations

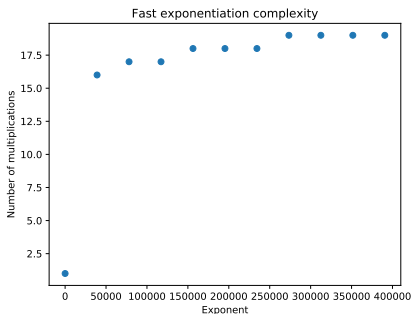
- ▶ Please use a variable in `exponentiation_complexity.py` to compute the number of operations in fast exponentiation.
- ▶ It should look like :



Experimental evaluation

Exercise 3: Counting the number of elementary operations

- We recognize the **logarithmic complexity** $\mathcal{O}(\log n)$



- └ The problem of complexity
- └ Measuring time complexities

Asymptotic behavior

- ▶ We study the **asymptotic** behavior, when $n \rightarrow \infty$

- └ The problem of complexity
- └ Measuring time complexities

Asymptotic behavior

- ▶ We study the **asymptotic** behavior, when $n \rightarrow \infty$
- ▶ This tells if the algorithm **scales** (still works when the instance of the problem is larger)

Asymptotic behavior : \mathcal{O} notation (notation de Landau)

- ▶ Mathematically speaking, we say that $f = \mathcal{O}(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ is **bounded**.

$$\exists A \geq 0, \forall n \in \mathbb{N}, \left| \frac{f(n)}{g(n)} \right| \leq A \quad (1)$$

- ▶ $||$ means "absolute value"
- ▶ intuitively, this means that f is "not bigger" than g

Asymptotic behavior : examples



$$n^2 + n = \mathcal{O}(?) \quad (2)$$



$$5 \times n^4 + 2178 \times n^3 + \log 3n = \mathcal{O}(?) \quad (3)$$

Asymptotic behavior : examples



$$n^2 + n = \mathcal{O}(n^2) \quad (4)$$



$$5 \times n^4 + 2178 \times n^3 + \log 3n = \mathcal{O}(n^4) \quad (5)$$

Asymptotic behavior : o notation

- ▶ Mathematically speaking, we say that $f = o(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ converges to 0 when $n \rightarrow +\infty$

$$\lim_{n \rightarrow +\infty} \left| \frac{f(n)}{g(n)} \right| = 0 \quad (6)$$

- ▶ intuitively, this means that f is "smaller" than g

Asymptotic behavior : o notation

- ▶ Mathematically speaking, we say that $f = o(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ converges to 0 when $n \rightarrow +\infty$

$$\lim_{n \rightarrow +\infty} \left| \frac{f(n)}{g(n)} \right| = 0 \quad (7)$$

- ▶ Please define this limit with quantifiers (quantificateurs) ?

Asymptotic behavior : o notation

- Mathematically speaking, we say that $f = o(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ converges to 0 when $n \rightarrow +\infty$

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0 \quad (8)$$



$$\forall \epsilon > 0, \exists A \in \mathbb{R}, \forall n \geq A, \left| \frac{f(n)}{g(n)} \right| \leq \epsilon \quad (9)$$

Asymptotic behavior : general rules

When $n \rightarrow +\infty$:

- ▶ if $\alpha > 0$, $\beta \in \mathbb{R}$, $(\log n)^\beta = o(n^\alpha)$
- ▶ if $\alpha < \beta$, $n^\alpha = o(n^\beta)$
- ▶ if $a > 1$, $n^\alpha = o(a^n)$
- ▶ if $0 < a < b$, $a^n = o(b^n)$

Asymptotic behavior : equivalence

► We say that $f(n) \underset{n \rightarrow +\infty}{\sim} g(n)$ when

$$f(n) \underset{n \rightarrow +\infty}{=} g(n) + o(g(n)) \quad (10)$$

Asymptotic behavior : equivalence

- ▶ We say that $f(n) \underset{n \rightarrow +\infty}{\sim} g(n)$ when

$$f(n) \underset{n \rightarrow +\infty}{=} g(n) + o(g(n)) \quad (11)$$

- ▶ When talking about complexities, we will be interested in the **simplest equivalent**.

Equivalence

Exercise 3 : Find equivalents and the limits for the following functions :

- ▶ $u_n = 3n^3 - n^2(\sqrt{n} \sin n) + \cos(\sqrt{n})$
- ▶ $v_n = -0.2 * n^n + 10 * n^2 * n!$
- ▶ Maximum number of edges in a simple directed graph
- ▶ $n!$

- └ The problem of complexity
- └ Measuring time complexities

Examples of algorithms

- ▶ Fast exponentiation
- ▶ Naive exponentiation
- ▶ Merge sort
- ▶ Insertion sort
- ▶ Matrix multiplication
- ▶ Enumeration of subsets, TSP, coloring
- ▶ Enumeration of permutations

- └ The problem of complexity
- └ Measuring time complexities

Examples of algorithms

- ▶ Fast exponentiation $\mathcal{O}(\log n)$
- ▶ Naive exponentiation
- ▶ Merge sort
- ▶ Insertion sort
- ▶ Matrix multiplication
- ▶ Enumeration of subsets, TSP, coloring
- ▶ Enumeration of permutations

- └ The problem of complexity
- └ Measuring time complexities

Examples of algorithms

- ▶ Fast exponentiation $\mathcal{O}(\log n)$
- ▶ Naive exponentiation $\mathcal{O}(n)$
- ▶ Merge sort
- ▶ Insertion sort
- ▶ Matrix multiplication
- ▶ Enumeration of subsets, TSP, coloring
- ▶ Enumeration of permutations

- └ The problem of complexity
- └ Measuring time complexities

Examples of algorithms

- ▶ Fast exponentiation $\mathcal{O}(\log n)$
- ▶ Naive exponentiation $\mathcal{O}(n)$
- ▶ Merge sort $\mathcal{O}(n \log n)$
- ▶ Insertion sort
- ▶ Matrix multiplication
- ▶ Enumeration of subsets, TSP, coloring
- ▶ Enumeration of permutations

- └ The problem of complexity
- └ Measuring time complexities

Examples of algorithms

- ▶ Fast exponentiation $\mathcal{O}(\log n)$
- ▶ Naive exponentiation $\mathcal{O}(n)$
- ▶ Merge sort $\mathcal{O}(n \log n)$
- ▶ Insertion sort $\mathcal{O}(n^2)$
- ▶ Matrix multiplication
- ▶ Enumeration of subsets, TSP, coloring
- ▶ Enumeration of permutations

- └ The problem of complexity
- └ Measuring time complexities

Examples of algorithms

- ▶ Fast exponentiation $\mathcal{O}(\log n)$
- ▶ Naive exponentiation $\mathcal{O}(n)$
- ▶ Merge sort $\mathcal{O}(n \log n)$
- ▶ Insertion sort $\mathcal{O}(n^2)$
- ▶ Matrix multiplication $\mathcal{O}(n^{2.37})$
- ▶ Enumeration of subsets, TSP, coloring
- ▶ Enumeration of permutations

- └ The problem of complexity
- └ Measuring time complexities

Examples of algorithms

- ▶ Fast exponentiation $\mathcal{O}(\log n)$
- ▶ Naive exponentiation $\mathcal{O}(n)$
- ▶ Merge sort $\mathcal{O}(n \log n)$
- ▶ Insertion sort $\mathcal{O}(n^2)$
- ▶ Matrix multiplication $\mathcal{O}(n^{2.37})$
- ▶ Enumeration of subsets, TSP, coloring $\mathcal{O}(2^n)$
- ▶ Enumeration of permutations

Examples of algorithms

- ▶ Fast exponentiation $\mathcal{O}(\log n)$
- ▶ Naive exponentiation $\mathcal{O}(n)$
- ▶ Merge sort $\mathcal{O}(n \log n)$
- ▶ Insertion sort $\mathcal{O}(n^2)$
- ▶ Matrix multiplication $\mathcal{O}(n^{2.37})$
- ▶ Enumeration of subsets, TSP, coloring $\mathcal{O}(2^n)$
- ▶ Enumeration of permutations $\mathcal{O}(n!)$

- └ The problem of complexity
- └ Measuring time complexities

Orders of magnitude

Orders of magnitude

Taille	$n \log n$	n^3	2^n
$n = 20$	60	8000	1048576
$n = 50$	196	125000	1125899907000000
$n = 100$	461	1000000	12676506000000000000000000000000

⇒ Hence the idea of a border between polynomial and exponential algorithms.

Profiling

- ▶ Another useful tool to monitor the execution of a program is **profiling**
- ▶ From the python docs : "A profile is a set of statistics that describes how often and for how long various parts of the program executed"
- ▶ <https://docs.python.org/3.6/library/profile.html>

Profiling

Exercise 4 : Profiling a piece of code

- ▶ **cd profiling** and profile some programs that we used before
- ▶ However note that when profiling **profiling_demo.py**, the elementary multiplications are not taken into account in the profiling output.

Time complexities in python

<https://wiki.python.org/moin/TimeComplexity>

Computing complexities

We now want to compute some complexities with paper and pen.
Let us focus on some intuitive rules :

- ▶ For a sequence of blocks :
- ▶ For a loop :

Computing complexities

We now want to compute some complexities with paper and pen.
Let us focus on some intuitive rules :

- ▶ For a sequence of blocks : complexities sum up
- ▶ For a loop :

Computing complexities

We now want to compute some complexities with paper and pen.
Let us focus on some intuitive rules :

- ▶ For a sequence of blocks : complexities sum up
- ▶ For a loop : complexities of all iterations sum up
- ▶ If a loop consists in similar iterations, its complexity is the product of the complexity of one iteration by the size of the loop.

Exercise 5 : Computing a running time I

Please compute the complexity of the following algorithm.

```
result = 0
for i in range(n):
    result += i**2
```

Exercise 6 : Computing a running time II

Please compute the complexity of the following algorithm.

```
for i in range(n):  
    for j in range(i):  
        l = [i+j+k for k in range(n)]
```

Running times

Exercise 7 : Computing a running time II

Could we have known that is was polynomial without performing the exact computation ?

```
for i in range(n):  
    for j in range(i):  
        l = [i+j+k for k in range(n)]
```

Some mathematical concepts

- ▶ Mathematical induction
- ▶ Applications : prime factors decomposition, $\sum_{k=1}^n k$
- ▶ Optional

$$\sum_{k=1}^n k^2 ? \quad (12)$$

$$\sum_{k=1}^n k^3 ? \quad (13)$$

Insertion Sort

- ▶ We will study the classic **Insertion sort algorithm**, in order to illustrate the concept of **average-case complexity**.

Insertion Sort

Exercise 8 : Insertion sort :

cd insertion_sort/ and fix the function in **insertion_sort.py** in order to perform the algorithm.

A test file **test_insertion_sort.py** is provided in order to check the correctness of the function.

Average-case complexity

- ▶ We assume a **uniform distribution** on the integer that we want to sort. All values have the same probability.
- ▶ What is the average-case complexity of the algorithm ?

Complexity

Exercise 9 : use the file **complexity.py** in order to check if our theoretical result is correct. You will need to fix the function **number_of_operations()**

Python sorting

In python, `sort()` uses a variant of mergesort.

<https://github.com/python/cpython/blob/master/Objects/listsort.txt>

Horner Algorithm

- ▶ Let us consider the case of evaluating polynoms
- ▶ A polynom is a function of the form
$$f : x \rightarrow a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$
- ▶ How many multiplications are involved with the naive method ?

Horner Algorithm

- ▶ Let us consider the case of evaluating polynoms
- ▶ A polynom is a function of the form
$$f : x \rightarrow a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$
- ▶ How many multiplications are involved with the naive method?
- ▶ We look for an algorithm that is faster than the naive solution.

Horner Algorithm

- ▶ Example of Horner algorithm when
 $P : x \rightarrow 7x^4 + 2x^3 - 5x + 1 :$

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \quad (14)$$

Horner Algorithm

- ▶ Example of Horner algorithm when
 $P : x \rightarrow 7x^4 + 2x^3 - 5x + 1 :$

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \quad (15)$$

- ▶ How many multiplications are now involved ?

Horner Algorithm

- ▶ Example of Horner algorithm when
 $P : x \rightarrow 7x^4 + 2x^3 - 5x + 1 :$

$$P(a) = (((7a + 2)a + 0)a - 5)a + 1 \quad (16)$$

- ▶ How many multiplications are now involved? $\mathcal{O}(n)$.
- ▶ So we went from quadratic to linear.

Horner Algorithm

- ▶ Example of Horner algorithm when

$$P : x \rightarrow 7x^4 + 2x^3 - 5x + 1 :$$

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \quad (17)$$

- ▶ We input the polynom to the algorithm as the list of the coefficients $[a_n, a_{n-1}, \dots, a_0]$

Evaluating polynoms

Exercise 9 : Implementation of Horner Algorithm

- ▶ Example of Horner algorithm when

$$P : x \rightarrow 7x^4 + 2x^3 - 5x + 1 :$$

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \quad (18)$$

- ▶ We input the polynom to the algorithm as the list of the coefficients $[a_n, a_{n-1}, \dots, a_0]$
- ▶ Please modify **complexity/horner.py** so that it performs the horner algorithm.
- ▶ In order to test that our method is correct, we will test it against the method **polyval** from **numpy**.

Horner

► `help(numpy.polyval)` :

```
Horner's scheme [1]_ is used to evaluate the polynomial. Even so,
for polynomials of high degree the values may be inaccurate due to
rounding errors. Use carefully.
```

References

```
.. [1] I. N. Bronshtein, K. A. Semendyayev, and K. A. Hirsch (Eng.
trans. Ed.), *Handbook of Mathematics*, New York, Van Nostrand
Reinhold Co., 1985, pg. 720.
```

Figure – The Horner algorithm is actually the method used by numpy !

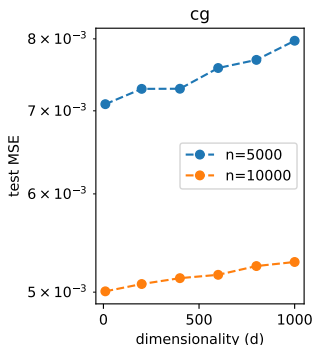
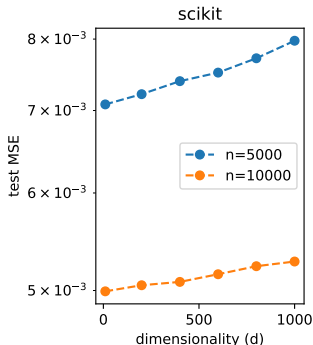
- └ The problem of complexity
- └ Conjugate gradient algorithm

Linear systems

In many applications, we must solve **linear systems** (backboard). To do so, several algorithms exist and the optimal one will depend on the input (similarly to sorting). In the following slides, the **conjugate gradient algorithm (CG)** from scipy is compared to the scikit-learn implemenration which uses `numpy.linalg.lstsq`.

Uniform inputs : result quality (test MSE)

Comparison between scikit and cg
Resolution of OLS
5 repeats per simulation
 $\sigma = 5.00\text{E-}01$
uniform inputs $\in [-5, 5]$
test MSE

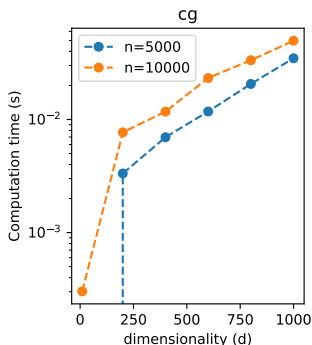
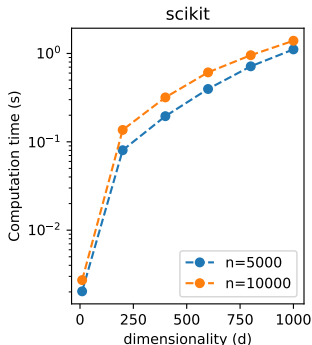


Overview

- └ The problem of complexity
- └ Conjugate gradient algorithm

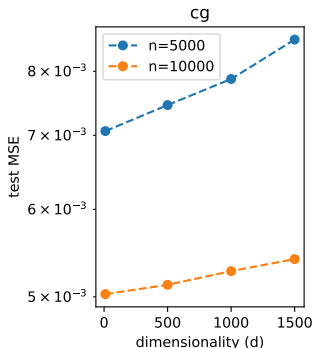
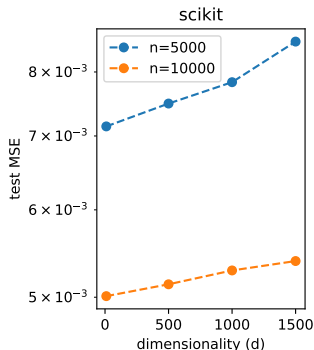
Uniform inputs : speed

Comparison between scikit and cg
Resolution of OLS
5 repeats per simulation
 $\sigma = 5.00\text{E-}01$
uniform inputs $\in [-5, 5]$
Computation time (s)



Normally distributed inputs : result quality (test MSE)

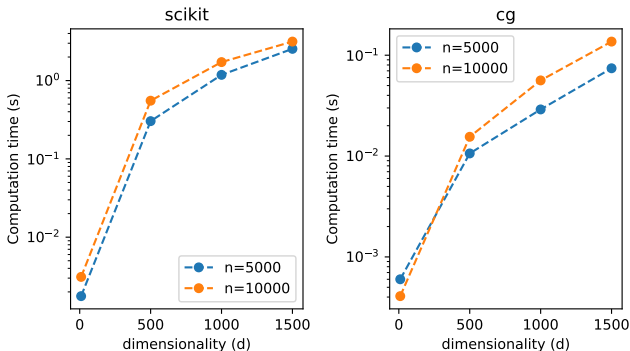
Comparison between scikit and cg
Resolution of OLS
5 repeats per simulation
 $\sigma = 5.00\text{E-}01$
standard_normal inputs
test MSE



- └ The problem of complexity
- └ Conjugate gradient algorithm

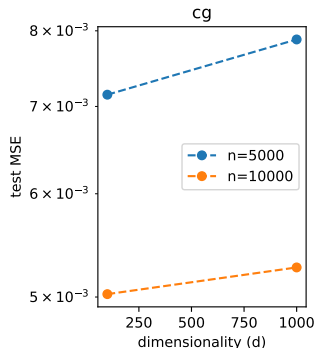
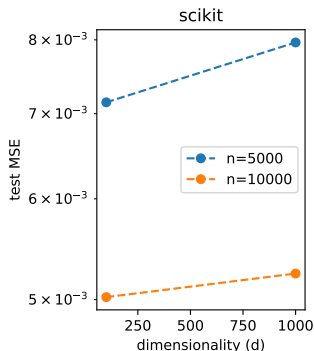
Normally distributed inputs : speed

Comparison between scikit and cg
Resolution of OLS
5 repeats per simulation
 $\sigma = 5.00\text{E-}01$
standard_normal inputs
Computation time (s)



Binary inputs : result quality (test MSE)

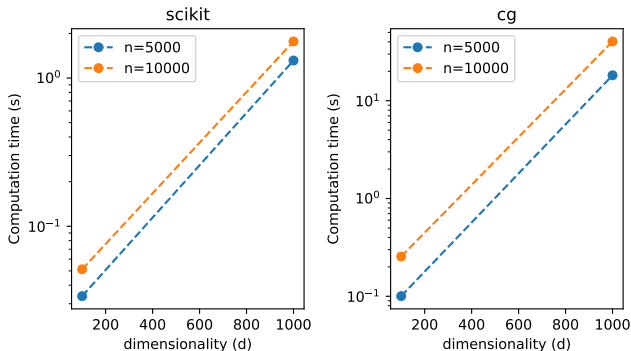
Comparison between scikit and cg
Resolution of OLS
5 repeats per simulation
 $\sigma = 5.00\text{E-}01$
binary inputs
test MSE



- └ The problem of complexity
- └ Conjugate gradient algorithm

Binary inputs : speed

Comparison between scikit and cg
Resolution of OLS
5 repeats per simulation
 $\sigma = 5.00\text{E-}01$
binary inputs
Computation time (s)



Space complexity

Space complexity is the sum of :

- ▶ input space
- ▶ auxiliary space : temporary space used during the algorithm