1 Bootstrapping Functions

1.1 Basic Algorithm

According to Wasserman 2004 we can define $T_n = g(X_1, \ldots, X_n)$ as an statistic that depends on the data, and apply bootstrapping for estimating the standard error and confidence intervals of T_n for statistical inference. First, Wasserman 2004 proves that by using the law of large numbers, and assuming that we draw an IID sample of $Y_1, \ldots, Y_B$, we can conclude that as $B \to \infty$

$$\bar{Y} \xrightarrow{p} E(Y) \tag{1}$$

$$\frac{1}{B}\sum_{j=1}^{B}(Y_j - \bar{Y})^2 \xrightarrow{p} V(Y) \tag{2}$$

Therefore, we can use the sample mean and the sample variance as an approximation for the population mean and variance. Following the procedures stated by Wasserman 2004, we first assume that the data obtained initially follows a distribution $\hat{F}_n$, from where we can take samples $X_1^*, \ldots, X_n^*$ and compute $T_n^* = g(X_1^*, \ldots, X_n^*)$ B times. From this step we will get a vector of $T_{n,1}^*, \ldots, T_{n,B}^*$, from where we can compute the variance (Wasserman 2004):

$$v_{boot} = \frac{1}{B}\sum_{b=1}^{B}\left(T_{n,b}^* - \frac{1}{B}\sum_{r=1}^{B}T_{n,r}^*\right)^2 \tag{3}$$

The standard error of the statistic T_n is the square root of the variance, $SE = \sqrt{v_{boot}}$. Although there are several methods to estimate the confidence interval of the statistics, we are going to use the percentiles of the statistic's distribution. Consequently, the interval is defined as $C_n = \left(T_{\alpha/2}^*, T_{1-\alpha/2}^*\right)$.

The previous can be resumed in Algorithm 1 shows the steps followed more generally. One can observe that it divides the process into three main parts: the first part generates B samples, of n_b size, from the original data using replacement, the second part estimates the statistic T_n for each b sample, and, finally, the algorithm estimates the standard error and the confidence interval. The sampling phase is a loop of size B that applies a sampling function[1], therefore we can assume that the time complexity of this process should be linear: $O(n)$, where n is equal to B. (Analysis based on Cormen et al. 2009) Similarly, the second phase, is a process that estimates the statistic for each sample b, which should be bounded by the same time complexity as the first phase if the statistic is simple enough, $O(n)$. Finally, the last step estimates the variance and the confidence interval, which, assuming as given steps without their own time complexity, have a constant time complexity $O(1)$. Under these assumptions, we can conclude that we expect for the algorithm to have a linear time complexity $O(n)$, where the slope is determined by the number of samples taken from the original data and the improvements made by using tools like parallel computing or optimized functions.

Algorithm 1: Bootstrapping

```
Data: A sample of a random variable $X$
Result: A standard error and confidence interval
/* Sampling phase                                                  */
for $b$ in range($B$):
    select $X^*_{1,b}, \dots, X^*_{n,b}$ elements of the original data using
      replacement;
/* Estimation phase                                                */
for $b$ in range($B$):
    estimate $T^*_{n,b}$ for the $b$-th sample;
/* Results phase                                                   */
estimate the square root of the variance of the statistics and the
  confidence interval
```

This algorithm could be translated into the following Python code following the steps mentioned by Wasserman 2004. As an example, we are going to assume that the original data comes from a normal distribution with $\mu = 5$ and $\sigma^2 = 1$ and want to estimate the standard error and the confidence interval for the statistic, which in this case is the mean.

```python
import numpy as np
from scipy.stats import norm
# Assume we have a random sample from a normal distribution
np.random.seed(11)
sample_normal = np.random.normal(5, 1, 10000)
```

[1] We are assuming that the sampling function has a linear time complexity, since it can be understood as a loop that generates a random number, under some specific conditions, to select an index of the input.

```python
# 1. Generate 1000 random samples with replacement
samples = np.array([np.random.choice(sample_normal,
                    size=100, replace=True) for _ in range(1000)])

# 2. Estimate the mean (statistic) for each random sample
mean_dist = np.array([np.mean(x) for x in samples])

# 3. Estimate the standard error and the confidence interval
se_mean = np.sqrt(np.var(mean_dist))
confint_mean = np.percentile(mean_dist, [2.5, 97.5])
print("Standard Error: ",  round(se_mean, 3),
        "\n95% Confidence Interval: ", round(confint_mean[0], 3),
        " - ", round(confint_mean[1], 3))
```

```
Standard Error:  0.099
95% Confidence Interval:  4.8  -  5.201
```

The main results from the algorithm are the standard error and the confidence interval of the statistic (i.e. mean), additionally we can observe the behavior of the bootstrap results in a histogram which, in the case of estimating the mean, illustrates the central limit theorem. Figure 1 shows the histogram of the bootstrap samples for the mean with the density kernel of the observed data (blue line), and the normal distribution (black line). Theory tells us that (Wasserman 2004), by the central limit theorem, the distribution of the sample mean converges in distribution to a $N(\mu, \sigma^2)$. With this, one can establish probability statements about the mean of the random variable X.

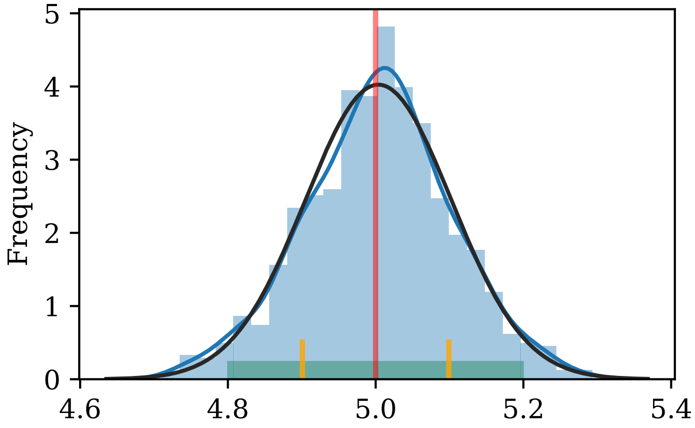

Figure 1: Histogram and density plot of the bootstrap samples. *Note*: confidence interval in green area, standard errors in orange and normal distribution in black

The algorithm implemented shows the expected results with respect to the statistical objectives, however, the main focus of this study is to analyze the performance of bootstrapping when using parallel computing. The next sections will divide the problem between a non-parallel version (i.e. serial) and a parallel implementation of the bootstrapping algorithm.

1.2 Serial Algorithms

The serial version of bootstrapping replicates the steps followed by the Algorithm 1 with only one processor of the computer. At this point, to construct a function that executes the algorithm, a main aspect is the generation of random numbers for the sampling phase. We decided to use two types of generators to understand how the performance of the algorithm can change based on specific changes on each step. The two options are: the `random` package of python core libraries (Algorithm 2) and `Numpy` package (Algorithm 3). It is noteworthy that the first implementation uses a nested loop (line 4), incrementing the time complexity to a polynomial form $O(mn)$, where m is the size of the samples. Therefore, we can expect that the time complexity of the function that uses `numpy` is bounded by the function without `numpy`.

Algorithm 2: Serial Bootstrapping without Numpy

```
Input  : A sample of a random variable $X$
Output: A object with the standard error and confidence interval
def bootstrap(input, statistic function, number of samples, size of sample):
    /* Sampling phase                                                    */
    for $b$ in range($B$):
        for $j$ in range(size of samples):
            // Using random.randint module
            index = create a random integer $\in$ [0, len(data[$n$]) - 1];
            sample$_b$[$j$] = data[index];
    /* Estimation phase                                                  */
    for $b$ in range($B$):
        result_array[$b$] = stat_function(array_of_samples[$b$]);
    /* Results phase                                                     */
    estimate the square root of the variance of the statistics and the
      confidence interval;
```

On the other hand, on Algorithm 3 we replicated the Algorithm 2 using the `numpy` package to generate the samples from the original data. Since `numpy` is a scientific package made to optimize processes like bootstrapping, we expect a improvement in terms of performance when both functions are compared. The Algorithm 3 enumerates the steps taken by the function to execute the bootstrapping. The main difference with the previous function resides in the `for loop` at line 3 in Algorithm 3, since we avoided the introduction of an additional `for loop`. Therefore, we could assume that the time complexity of this algorithm is $O(n)$, since it just performs the sample and estimation in one `for loop`.

Algorithm 3: Serial Bootstrapping with `Numpy`

```
Input  : A sample of a random variable $X$
Output: A object with the standard error and confidence interval
def bootstrap_np(input, statistic function, number of samples, size
  of sample):
    /* Sampling phase and Estimation phase                        */
    for $b$ in range($B$):
        // Using numpy.random.choice
        array_of_samples[$b$] = create a random sample of determined
          size from data;
        result_array[$b$] = stat_function(array_of_samples[$b$]);
    /* Results phase                                              */
    estimate the square root of the variance of the statistics and the
      confidence interval;
```

To further analyze the performance of the algorithms, we first implemented a test to generate a range of B samples, since this is the main argument of the bootstrap function that could affect the performance of the functions. The range of the test has a minimum of $B = 1,000$ samples up to $B = 1,000,000$ samples by steps of $10,000$. For each implementation of the algorithm, we time the performance using the `timeit` module of Python libraries, which returns an average time for each run. The results are presented in the Figure 2, where the straight blue line is the function without `numpy`, and the dashed line is the function in Algorithm 3. In average, the function with Numpy is 8.64 times faster than the function without this package, confirming our hypothesis on the difference of the time complexity between both algorithms.

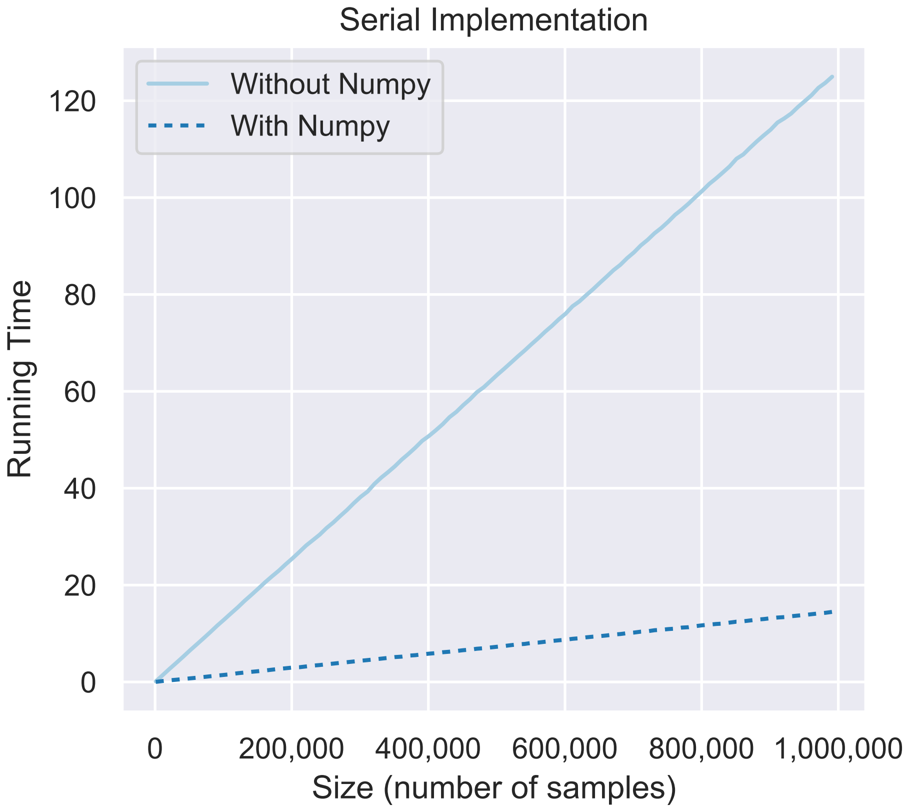

Figure 2: Bootstrapping Test for Serial Functions: Performance depending on number of samples

1.3 Parallel Algorithms

According to Eubank and Kupresanin 2011 ”parallel processing ’languages’ provide ways of managing the work performed by different processors in a multi-processor environment.” Furthermore, they say that by using parallel computing one should be able to divide the problem into smaller ones and then use the different available processors to solve the entire problem. We can interpret from these authors that in order to use parallel computing, the problem has to be divided into smaller parts. As seen in Algorithm 1, there are 2 main loops that could work as one loop and independently between each of it’s steps. Consequently, we could be able to divide the process into smaller problems and then join them to get the results (See Algorithm 4 line 2).

Algorithm 4: Parallel Bootstrapping

```
  Data: A sample of a random variable $X$
  Result: A standard error and confidence interval
1 /* Sampling phase and Estimation phase                    */
2 for $b$ in range($B$):
3     // Subproblem $b$
4     select $X^*_{1,b}, \dots, X^*_{n,b}$ elements of the original data using
        replacement;
5     estimate $T^*_{n,b}$ for the $b$-th sample;
6 /* Results phase                                           */
7 estimate the square root of the variance of the statistics and the
    confidence interval
```

To implement the parallel computing version of the Serial Bootstrapping algorithms, we are going to use the `multiprocessing` library, which allows the possibility of creating a `Pool` of ”workers” to divide the problem and then join the results to generate the final solution. In terms of the algorithm, the `for loop`, in line 2 of Algorithm 4, is going to be replaced by a `Pool` object that will implement the function a determined number of times (B) to obtain the underlying empirical distribution of the statistic. Since we have two options for the sampling phase, i.e. with `numpy` and without it, we are going to test the performance of the functions implementing both cases under parallel computing and compare the results, and then we will analyze the results between serial and parallel computing.

Algorithm 5: Parallel Bootstrapping Example with `multiprocessing module`

```
Data: A sample of a random variable $X$
Result: A standard error and confidence interval
/* Sampling phase and Estimation phase */
Parallel: with mp.Pool(workers) as pool:
    // Since there are $B$ sub-problems, the
       multiprocessing Pool object will divide the
       implementation into the number of workers in equal
       chunk sizes
    select $X^*_{1,b}, \ldots, X^*_{n,b}$ elements of the original data using
      replacement;
    estimate $T^*_{n,b}$ for the $b$-th sample;
end
/* Results phase */
estimate the square root of the variance of the statistics and the
  confidence interval
```

It is important to mention at this point that the implementation of the function under the `multiprocessing` module of Python is based on the repetition of a function among an iterable object. Since we are repeating the same process independently i.e. sampling and estimating, over the same iterable objects, i.e. the arguments[2], a memory problem may appear. If the original data is too large and we want to generate a significant amount of bootstrapping samples from it, then a list of arguments would be a vector with a repeated copy of the data vector B times. To avoid this problem we created a shared object using the `Array` object of the `multiprocessing` module where we stored the original data array. Therefore, no copy of the original data should get generated at each repetition.

[2] Please refer to the code to understand the meaning of the use of arguments as iterable.

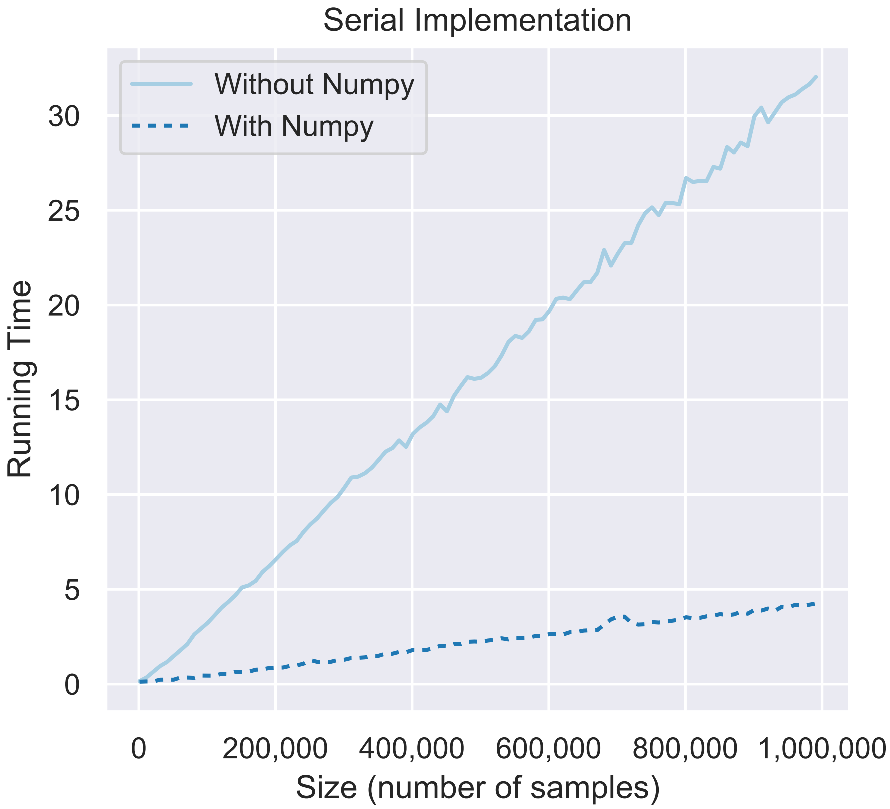

Figure 3: Bootstrapping Test for Parallel Functions: Performance depending on number of samples

Figure 3 shows the results from the implementation of the parallel version of both functions. As seen before in the serial case, the function that uses the `numpy` package is bounded, in terms of time complexity, by the one without it. The performance of the function with `numpy` is in average 7.26 faster than the one without it, which is a similar result we obtained when using the serial functions. For this test, we used the total number of cores of the computer where it was tested (i.e. 6 cores), but another important question at this point is how the performance changes when the number of computer cores change as well. To answer the latter, we did a test of performance by using an increasing number of cores up until the total number available in the computer (i.e. 6 cores) and leaving at 10,000 the number of samples.

The results are presented in the Figure 4 and illustrate that the number of cores have a more significant impact in the parallel version of the algorithm that does not use `numpy`, specially when it uses more than 1 core. Contrarily, the use of `numpy` has a relevant impact for more than 2 cores, although it is not as significant as the the case without the scientific package.

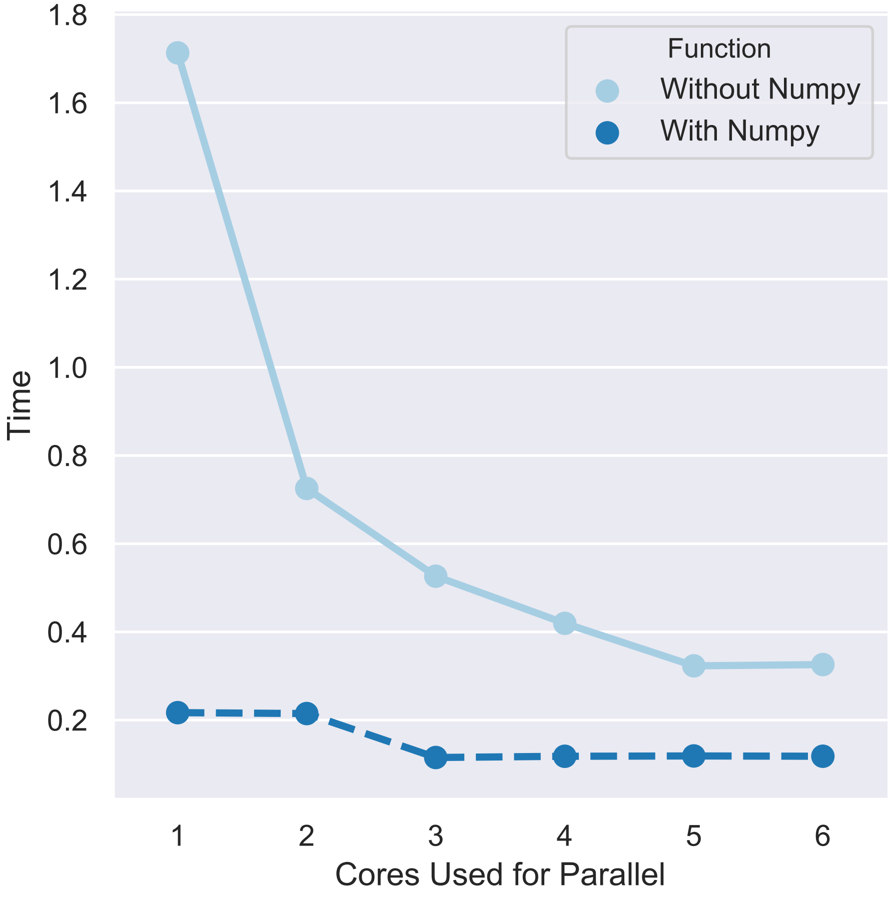

Figure 4: Bootstrapping Test for Parallel Functions: Performance depending on number of Cores and executing $10,000$ samples

1.4 Comparing Serial to Parallel Functions

After observing the results of using the `numpy` package on both serial and parallel functions, we are going to analyze the performance between the serial and the parallel version of each function. In this case, we should expect an improvement in both cases, from serial to parallel, and faster performance for the `numpy` version of each function.

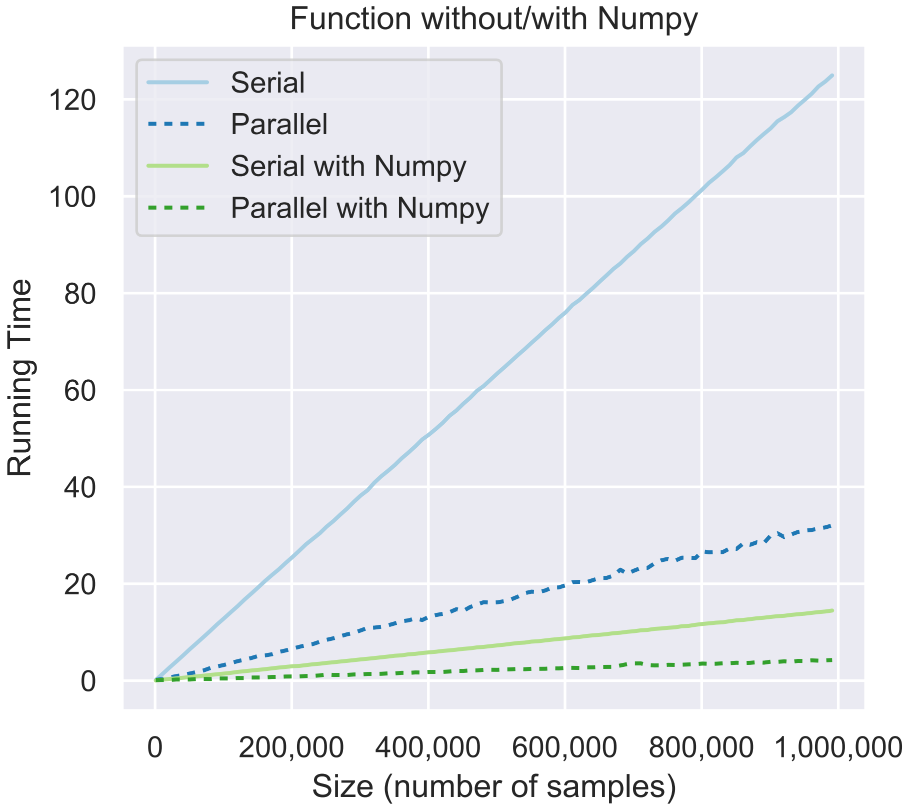

Figure 5: Bootstrapping Test for Serial and Parallel Functions: Performance depending on number of samples

Figure 5 includes all the previous results, and, as expected, the parallel versions have a faster performance than the serial ones. Nonetheless, it is noteworthy the fact that the serial version with `numpy` has a better performance (2.29 times) than the parallel version without it. This suggests that the numpy package improves the performance of the algorithm even when it is parallelised. Additionally, when comparing the serial version without numpy and the parallel with it, the performance of the algorithm is, in average, 28.4 faster. Although the results seem consistent along the number of samples, we analyze the performance for smaller number of samples. Figure 6 shows that the serial versions of the function seems to have better performance when the size of samples is lower than $10,000$. This results could be related with the series of steps that the multiprocessing module has to execute to create the Pool object and the tasks related to it. This overhead can be seen as a cost for the algorithm when a low level of samples are generated.

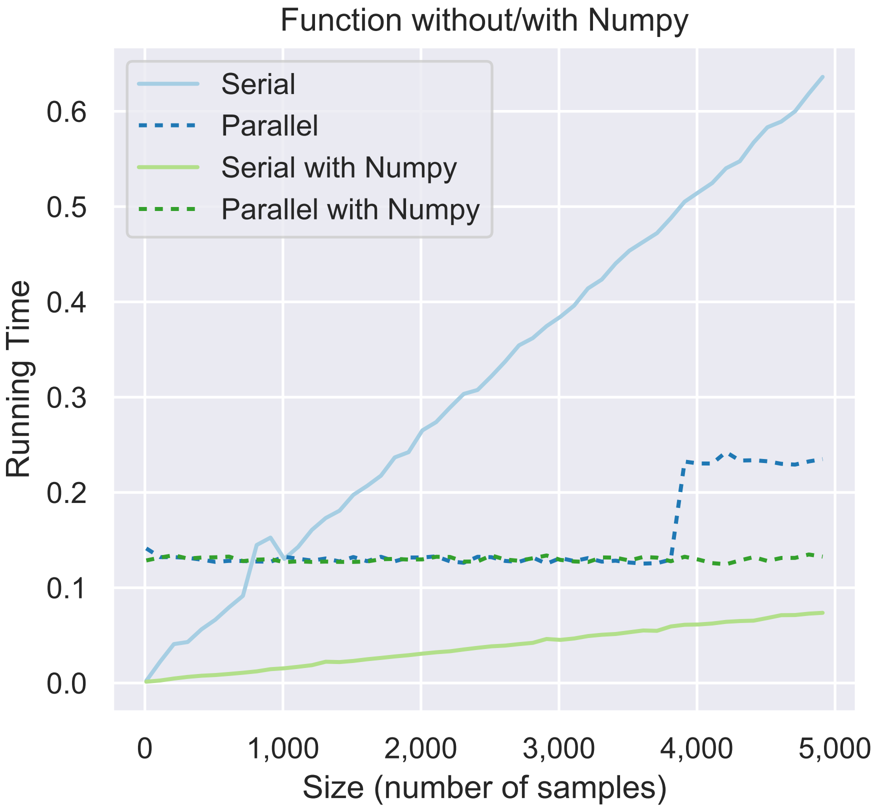

Figure 6: Bootstrapping Test for Serial and Parallel Functions: Performance depending on number of samples between 10 and $5,000$

1.5 Task Divition: Separating Sampling and Estimation

The previous section illustrates that the parallel versions of the bootstrapping algorithm proposed in the the present study has have better performance, specially when special scientific packages are used to optimize it. However, when the number of samples was lower than $10,000$ the serial version had a better performance, demonstrating that parallel computing has limitations. These limitations seems to be related with a time overhead needed to develop the framework for the parallel implementation. To further analyze this limitations we divided the sampling phase from the estimation phase and construct a function for each of these phases, in a serial and a parallel version.

References

Cormen, Thomas H. et al. (2009). *Algorithms*. MIT Press.

Eubank, Randall L. and Ana Kupresanin (2011). *Statistical Computing in C++ and R*. Chapman and Hall-CRC.

Wasserman, Larry (2004). *All of Statistics*. Springer Science+Business Media.