7c. Benchmarking Execution Time

Martin Alfaro
PhD in Economics

INTRODUCTION

This section presents standard tools for benchmarking code. In particular, our website will provide results based on the package `BenchmarkTools`, which is currently the most mature and reliable option. Nonetheless, the newer `Chairmarks` package has shown impressive speed gains. Once it's been around enough to fix any potential bugs, I'd recommend it as the default option.

TIME METRICS

Regardless of the package employed, Julia uses the same time metrics described below. These metrics can be accessed at any time **in the left bar**, under "**Notation & Hotkeys**".

Unit	Acronym	Measure in Seconds
Seconds	`s`	1
Milliseconds	`ms`	10^{-3}
Microseconds	`µs`	10^{-6}
Nanoseconds	`ns`	10^{-9}

In addition to execution times, all packages provide information about **memory allocations on the heap**, commonly referred to as **allocations**. These allocations can have a significant impact on performance and potentially indicate suboptimal coding practices. Their relevance will become clearer in the upcoming sections.

"TIME TO FIRST PLOT"

The expression "Time to First Plot" refers to a side effect of how the Julia language works: in each session, **the first function call is slower than the subsequent ones**. This occurs because Julia compiles code during its initial run, as we'll explain more thoroughly in the next sections.

Since the penalty will be incurred only once, this overhead isn't a major hindrance when working on a large project. However, it determines that Julia may not be the best option when you simply aim to perform a rapid analysis (e.g., run one regression or draw a graph to quickly unveil a relation).

The speed of the first run varies significantly. While it may be imperceptible in some cases, it can be noticeable in others. For instance, I bet you didn't notice that running `sum(x)` for the first time in your code was slower than the subsequent runs! On the contrary, drawing high-quality plots can take up to 15 seconds, explaining where the term "Time to First Plot" comes from. [1]

> **Warning!**
>
> The time-to-first-plot issue has been reduced significantly since `Julia 1.9`, and each new version is reducing this overhead even further.

@TIME

Julia comes with a built-in macro called `@time` to measure the execution time of a single run. The results provided by this macro, nonetheless, suffer from two significant limitations. First, the time provided is volatile and unreliable, as it only considers a single execution. Furthermore, considering the time-to-first-plot issue, the initial time will incorporate the compilation overhead, and hence it's unrepresentative of the subsequent calls. Taking this into account, you should always execute `@time` at least twice.

The following example illustrates its use, making it clear that first runs include compilation time.

```
x = 1:100

@time sum(x)         # first run                      -> it incorporates compilation time
@time sum(x)         # time without compilation time -> relevant for each subsequent run
```

```
 0.003763 seconds (2.37 k allocations: 145.109 KiB, 99.11% compilation time)
 0.000005 seconds (1 allocation: 16 bytes)
```

PACKAGE "BENCHMARKTOOLS"

A more accurate alternative for measuring execution time is provided by the `BenchmarkTools` package. By performing repeated computations of the operation, this reports more reliable timing results.

The package can be used through either the macro `@btime`, which only reports the minimum time, or through the macro `@benchmark`, which provides a more detailed analysis. Both macros discard the time of the first call, thus excluding compilation time from the results.

```
using BenchmarkTools

x = 1:100
@btime sum($x)        # provides minimum time only
```

```
 1.500 ns (0 allocations: 0 bytes)
```

```
using BenchmarkTools

x = 1:100
@benchmark sum($x)    # provides more statistics than `@btime`
```

Notice that we appended the symbol `$` to the variable `x` in both macros. The notation is necessary to indicate that `x` shouldn't be interpreted as a global variable, as occurs when when a variable is passed to a function. As we'll exclusively benchmark functions, you should always add `$` to each variable—**omitting `$` would lead to inaccurate results**.

The following example demonstrates the consequence of excluding `$`, wrongly reporting higher times than the execution really demands. [2]

```
using BenchmarkTools
x = rand(100)

@btime sum(x)
```

```
 16.132 ns (1 allocation: 16 bytes)
```

```
using BenchmarkTools
x = rand(100)

@btime sum($x)
```

```
 6.600 ns (0 allocations: 0 bytes)
```

PACKAGE "CHAIRMARKS"

A recent new alternative to benchmarking code is given by the package `Chairmarks`. Its notation is quite similar to `BenchmarkTools`, with the macros `@b` and `@be` providing a similar functionality to `@btime` and `@benchmark` respectively. The main benefit of `Chairmarks` is its speed, being orders of magnitude faster than `BenchmarkTools`.

```
using Chairmarks
x = rand(100)

display(@b sum($x))        # provides minimum time only
```

```
6.704 ns
```

```
using Chairmarks
x = rand(100)

display(@be sum($x))       # analogous to `@benchmark` in BenchmarkTools
```

```
Benchmark: 3638 samples with 3720 evaluations
min    6.694 ns
median 6.855 ns
mean   6.990 ns
max    34.462 ns
```

REMARK ON RANDOM NUMBERS FOR BENCHMARKING

The upcoming sections will present alternative approaches for computing the same operation, with the goal of identifying the most efficient method. To compare performance without skewing results, the execution times of each approach will be tested using *the same set of random numbers* as inputs.

Producing identical random numbers in demand can be achieved by using random seeds. By fixing a random seed, we guarantee that each time we generate a set of numbers, the same sequence of random numbers is produced. This creates a consistent pattern of random numbers each time the code is executed.

Rather than generating a predictable sequence, our use case only requires that the same set of random numbers is provided for the operation analyzed. We can ensure this by simply resetting the seed before executing each operation, so that the same set of random numbers is generated during the first call.

Random number generation is provided by the package `Random`. Below, we show how to set a specific seed, for which *any arbitrary number can be used*. For instance, the following examples set the seed `1234` before executing each operation.

```
using Random

Random.seed!(1234)      # 1234 is an arbitrary number, use any number you want
x = rand(100)

Random.seed!(1234)
y = rand(100)           # identical to `x`
```

```julia
using Random

Random.seed!(1234)      # 1234 is an arbitrary number, use any number you want
x = rand(100)


y = rand(100)           # different from `x`
```

To simplify the presentation throughout the website, code snippets based on random numbers will omit the lines that set the random seed. This can be appreciated below, where we illustrate the code that will be displayed and the actual code executed.

```julia
using Random
Random.seed!(123)

x = rand(100)

y = sum(x)
```

```julia
# We omit the lines that seet the seed


x = rand(100)

y = sum(x)
```

BENCHMARKS IN PERSPECTIVE

When evaluating approaches for performing a task, the differences in execution time are often negligible, typically on the order of nanoseconds. However, this should not lead us to believe that the choice of method has no practical implications.

It's true that operations in isolation usually have an insignificant impact on runtime. However, **the relevance of our benchmarks lies in scenarios where these operations are performed repeatedly**. This includes cases where the operation is called in a for-loop, or in iterative procedures (e.g., solving systems of equations or the maximization of a function). In these situations, small differences in timing are amplified, as they're replicated multiple times.

AN EXAMPLE

To illustrate the practical application of benchmarks, let's consider a concrete example. Suppose we want to double each element of a given vector `x`, and then calculate their sum. In the following, we'll compare two different approaches to accomplish this task.

The first method will be based on `sum(2 .* x)`, where `x` enters into the computation as a global variable. As we'll discuss in later sections, this approach is relatively inefficient. A more efficient alternative is given by `sum(a -> 2 * a, x)`, with `x` passed as a function argument. For the purpose

of this comparison, the specific functionality of this function is irrelevant. What matters is that this expression produces the same result as the first method, but employing a different technique.

The runtime performance of each approach is as follows.

```
x      = rand(100_000)

foo()  = sum(2 .* x)
```

```
  65.900 µs (4 allocations: 781.34 KiB)
```

```
x      = rand(100_000)

foo(x) = sum(a -> 2 * a, x)
```

```
  6.060 µs (0 allocations: 0 bytes)
```

The results reveal that the second approach achieves a significant speedup, taking less than 15% of the slower approach. However, even the "slow" approach is still extremely fast, taking less than 0.0001 seconds to execute.

This pattern will be a common theme in our benchmarks, where absolute execution times are often negligible. In such cases, the relevance of our conclusions depends on the specific characteristics of the task at hand. If the operation is only performed once in isolation, readability should be the primary consideration, and the most readable approach should be chosen. On the other hand, if the operation is repeated multiple times, small performance differences might accumulate and become significant, making the faster approach a more suitable choice.

To illustrate this point, let's take the functions from the previous example and call them within a for-loop that iterates 100,000 times.

```
x      = rand(100_000)
foo()  = sum(2 .* x)

function replicate()
   for _ in 1:100_000
      foo()
   end
end
```

```
  22.221 s (400000 allocations: 74.51 GiB)
```

```
x      = rand(100_000)
foo(x) = sum(a -> 2 * a, x)

function replicate(x)
   for _ in 1:100_000
      foo(x)
   end
end
```

```
 650.441 ms (0 allocations: 0 bytes)
```

The underscore symbol `_` is a common programming convention to denote "throwaway" variables. These variables are included solely to satisfy the syntax of an operation, and their value isn't actually used. In our case, it simply reflects that each iteration of the for-loop is replicating the same operation.

The example starkly reveals the consequences of calling the function within a for-loop. Now, the execution time of the slow version balloons to more than 20 seconds, while the fast version completes the task in under one second. Overall, this unveils the relevance of optimizing functions that are called repeatedly, where even slight improvements can have a substantial impact on performance.

FOOTNOTES

[1.] The time indicated is for `Plots`, the standard package for the task. This is a metapackage bundling several plotting libraries. Some of these libraries are relatively fast, requiring less than 5 seconds for rendering the first figure.

[2.] In fact, `sum(x)` doesn't allocate memory, unlike what's reported without `$`.