7c. Benchmarking Execution Time

[Martin Alfaro](#)
PhD in Economics

INTRODUCTION

This section introduces standard tools for benchmarking code performance. In particular, our website currently reports results based on the `BenchmarkTools` package, which is currently the most mature and reliable option in the Julia ecosystem. That said, the newer `Chairmarks` package has demonstrated notable improvements in execution speed when compared to `BenchmarkTools`. I recommend adopting `Chairmarks` as the default benchmarking tool once it has achieved sufficient stability and adoption within the community.

TIME METRICS

Julia uses the same time metrics described below, regardless of whether you use `BenchmarkTools` or `Chairmarks`. For quick reference, these metrics can be accessed at any point **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 report information about **memory allocations on the heap**, commonly referred to as just **allocations**. They can have a significant impact on performance and potentially indicate suboptimal coding practices. As we'll see in later sections, keeping track of allocations is crucial for high performance.

"TIME TO FIRST PLOT"

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

Since the penalty is incurred only once per session, this overhead isn't a major hindrance for large projects, where startup costs are quickly amortized. However, it implies that Julia may not be the best option for quick one-off analyses (e.g., running a regression or drawing a graph to get a sense of data).

It's hard to provide general statements about the inconvenience caused by this feature, since the time of the first run varies significantly across functions—while it may be imperceptible in some cases, it can be noticeable in others. For instance, the impact for simple operations like `sum(x)` won't be noticeable at all. Instead, rendering a high-quality plot for the first time can take up to fifteen seconds, explaining the origin of the term "Time to First Plot".

> **Warning!**
>
> The Time-to-First-Plot issue has been significantly mitigated since `Julia 1.9`, thanks to improvements in precompilation. Each new version is reducing this overhead even further.

@TIME

Julia comes with a built-in macro called `@time` for measuring the execution time of a single run. The results provided by this macro, nonetheless, suffer from two significant limitations. First, the time provided is based on just a single run, which can be highly variable and therefore unreliable. Additionally, considering the time-to-first-plot issue, the first measurement will always incorporate compilation overhead, making it unrepresentative of subsequent calls. This is why you should always execute `@time` at least twice.

The following example illustrates the use of `@time`, which also demonstrates the time difference between the first and subsequent runs.

```
x = 1:100

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

```
 0.002747 seconds (3.56 k allocations: 157.859 KiB, 99.36% compilation time)
 0.000003 seconds (1 allocation: 16 bytes)
```

PACKAGE "BENCHMARKTOOLS"

A more reliable alternative for measuring execution time is provided by `BenchmarkTools`. This package improves its accuracy by performing repeated computations of an operation. It then generates summary statistics based on these runs. Depending on the level of detailed required, the

package offers two macros: `@btime`, which only reports the minimum time, and `@benchmark`, which provides detailed statistics. 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
```

```
 2.314 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. [1]

```
using BenchmarkTools
x = rand(100)

@btime sum(x)
```

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

```
using BenchmarkTools
x = rand(100)

@btime sum($x)
```

```
 6.546 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
```

```
6.550 ns
```

```
using Chairmarks
x = rand(100)

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

```
Benchmark: 3856 samples with 3661 evaluations
 min    6.679 ns
 median 6.815 ns
 mean   6.785 ns
 max    14.539 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)
```

```
  35.519 μs (5 allocations: 781.37 KiB)
```

```
x      = rand(100_000)

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

```
  6.393 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
```

```
  5.697 s (500000 allocations: 74.52 GiB)
```

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

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

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