7c. Benchmarking Execution Time

Martin Alfaro
PhD in Economics

INTRODUCTION

This section introduces standard tools for benchmarking code performance. Our website 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 compared with `BenchmarkTools`. I recommend adopting `Chairmarks` once it's achieved sufficient stability and adoption within the community.

To set the stage, we'll start by addressing some key points for interpreting benchmark results. We'll also look at Julia's built-in `@time` macro, whose limitations explain why `BenchmarkTools` and `Chairmarks` should be used instead.

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}

Alongside execution times, each package also reports the amount of **memory allocated on the heap**, typically referred to simply as **allocations**. These allocations can play a major role in overall performance, and usually indicate suboptimal coding practices. As we'll explore in later sections, monitoring allocations tends to be crucial for achieving high performance.

"TIME TO FIRST PLOT"

The expression "Time to First Plot" refers to a side effect of how Julia operates, where the first execution in any new session takes longer than subsequent ones. This latency isn't a bug. Rather, it's a direct consequence of the language's design, which relies on a just-in-time (JIT) compiler: Julia compiles the code for executing functions in their first run, translating them into highly optimized machine code on the fly. This compilation process will be thoroughly covered in upcoming sections.

The first time you run any function, Julia generates low-level machine instructions to carry out the function's operations. This process of translating human-readable code into machine-executable instructions is called **compilation**. Unlike other programming languages, Julia relies on a just-in-time (JIT) compiler, where this code is compiled on-the-fly when a function is first run. This compilation process will be thoroughly covered in upcoming sections.

In each new session, this compilation penalty is incurred only once per function and set of argument types. Once a function is compiled, its machine code is cached, making all subsequent calls faster. The consequence is that the resulting overhead isn't a major hindrance for large projects, where startup costs are quickly amortized. However, it does mean that Julia may not be the best option for quick one-off analyses, such as running a simple regression or producing a quick exploratory plot.

The latency caused by this feature varies significantly across functions, making it difficult to generalize its impact. While it may be imperceptible for simple functions like `sum(x)`, it can be noticeable for rendering a high-quality plot. Indeed, drawing a first plot during a session can take several 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 subsequent release is reducing this overhead even further.

@TIME

Julia comes with a built-in macro called `@time`, allowing you to get a quick sense of an operation's execution time. The results provided by this macro, nonetheless, suffer from several limitations that make it unsuitable for rigorous benchmarking.

First, a measurement based on just a single execution is often unreliable, as runtimes can fluctuate significantly due to background processes on your computer. Additionally, if that run is a function's first call, the measurement will include compilation overhead. The extra time Julia spends generating machine code inflates the reported runtime, making it unrepresentative of subsequent calls.

While running `@time` multiple times can address these issues, its most significant flaw arises when benchmarking functions. This is because `@time` mischaracterizes function arguments as global variables. We'll show in upcoming sections that global variables have a marked detrimental effect on performance. Consequently, the time reported doesn't accurately reflect how the function would perform in practice.

The following example illustrates the use of `@time`, highlighting the difference in execution time 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`, which addresses the shortcomings of `@time` in several ways.

First, it reduces result variability by running operations multiple times and then computing summary statistics. It also measures the execution time of functions without compilation latency, since the package discards the first run for the reported timing. Additionally, the package allows users to explicitly control variable scope: by prefixing function arguments with the `$` symbol, they're treated as local variables during a function call.

The package offers two macros, depending on the level of detail required: `@btime`, which only reports the minimum time, and `@benchmark`, which provides detailed statistics. Below, we demonstrate their use.

```
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`
```

In later sections, we'll exclusively benchmark functions. This means that you should always prefix the function arguments with `$`. **Omitting `$` will lead to inaccurate results**, including incorrect reports of memory allocations.

The following example demonstrates the consequence of excluding `$`, where the runtimes reported are higher than the actual runtime.

```julia
using BenchmarkTools
x = rand(100)

@btime sum(x)
```
```
 14.465 ns (1 allocation: 16 bytes)
```

```julia
using BenchmarkTools
x = rand(100)

@btime sum($x)
```
```
 6.546 ns (0 allocations: 0 bytes)
```

PACKAGE "CHAIRMARKS"

A new alternative for benchmarking code is the `Chairmarks` package. Its notation closely resembles that of `BenchmarkTools`, with the macros `@b` and `@be` providing a similar functionality to `@btime` and `@benchmark` respectively. The main benefit of `Chairmarks` is its speed, as it can be orders of magnitude faster than `BenchmarkTools`.

As with `BenchmarkTools`, measuring the execution time of functions requires prepending function arguments with `$`.

```julia
using Chairmarks
x = rand(100)

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

```julia
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

When comparing the performance of different methods, we must ensure that our measurements aren't skewed by variations in the input data. This implies each approach must be tested using *the exact same set of values*. This guarantees that differences in execution time can be attributed solely to the efficiency of the method itself, rather than to a change in the inputs.

Such consistency can be achieved by using random number generators. They rely on a **random seed**, which is an arbitrary starting point that dictates the entire sequence of values they produce. By setting the same seed before each test, we can generate identical deterministic sequences of random numbers across multiple runs. Importantly, **any arbitrary number can be used for the seed**. The only requirement is that the same number is employed, so that you replicate the exact same set of random numbers.

Random number generation is provided by the package `Random`. Below, we demonstrate its use by setting the seed `1234` before executing each operation. Note, though, that any other number could be used.

```
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`
```

```
using Random

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


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

For presentation purposes, code snippets on this website will omit the lines dedicated to setting the random seed. While adding these code lines is essential for ensuring reproducibility, their inclusion in every example would create unnecessary clutter. Below, we illustrate the code that will be displayed throughout the website, along with the actual code executed.

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

x = rand(100)

y = sum(x)
```

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


x = rand(100)

y = sum(x)
```

BENCHMARKS IN PERSPECTIVE

When evaluating approaches for performing a task, execution times are often negligible, typically on the order of nanoseconds. Yet, this doesn't mean that the choice of method is without practical consequence.

While it's true that operations in isolation may have an insignificant impact on a program's overall runtime, **the relevance of benchmarks emerges when 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 maximizing functions). In these situations, small differences in timing are amplified as they are replicated hundreds, thousands, or even millions of times.

AN EXAMPLE

To illustrate this matter, let's consider a concrete example. Suppose we want to double each element of a 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)`, with `x` treated as a global variable. As we'll discuss in later sections, this approach is relatively inefficient. A more performant alternative is given by `sum(a -> 2 * a, x)`, where `x` is passed as a function argument. While we haven't explained why this implementation is better, it's sufficient to note that both methods produce the same result. The measured runtimes of each approach are 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, requiring less than 15% of the time taken by the slower method. However, even the "slow" approach is remarkably fast, taking less than 0.0001 seconds to execute.

This pattern will be recurring in our benchmarks, where absolute execution times are often negligible. In such cases, the relevance of our conclusions heavily depends on the context. If the operation is only performed once in isolation, readability should be the primary consideration for choosing a method. On the other hand, if the operation is executed repeatedly, small differences in performance might accumulate and become meaningful, making the faster approach a more suitable choice.

To make this point concrete, let's revisit the functions from the previous example and call them inside a for-loop that runs 100,000 times. Since our sole goal is to repeat the operation, the iteration variable itself plays no role. In such cases, it's common practice to employ a **throwaway variable**: placeholder that exists only to satisfy the loop's syntax, without ever being referenced. This convention signals to other programmers that the variable's value can be safely ignored. In our example, `_` serves this purpose, simply reflecting that each iteration performs exactly the same operation.

```
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 example starkly reveals the consequences of calling these functions within a for-loop. The execution time of the slow version now jumps to more than 20 seconds, while the fast version finishes in under one second. Such a stark contrast underscores the importance of optimizing functions that are executed repeatedly: even seemingly minor improvements can accumulate into pronounced performance gains.