9d. Slice Views to Decrease Allocations

Martin Alfaro
PhD in Economics

INTRODUCTION

In a previous discussion on Slices and Views, we defined a slice as a subvector derived from a parent vector `x`. Common examples include expressions such as `x[1:2]`, which extracts elements at positions 1 and 2, or `x[x .> 0]`, which selects only those elements that are positive. By default, these operations create a copy of the data and therefore allocate memory. The only exception to this rule occurs when a slice comprises a single object.

In this section, we address the issue of memory allocations associated with slices. To do this, we highlight the role of **views**, which bypass the need for a copy by directly referencing the parent object. The strategy is particularly effective when slices are indexed through ranges. On the contrary, it's not suitable for slices that employ Boolean indexing, like `x[x .> 0]`, where memory allocation will still occur even with views.

Interestingly, we'll show scenarios where **copying data could actually be faster than using views**, despite the additional memory allocation involved. This apparent paradox emerges because copied data is stored in a contiguous block of memory, which facilitates more efficient access patterns. In contrast, views might reference non-contiguous memory locations, potentially leading to slower access times, despite avoiding the initial allocation cost.

VIEWS OF SLICES

We begin by showing that views don't allocate memory *when a slice is indexed by a range*. This behavior can yield performance improvements over regular slices, which create a copy of the data by default.

SLICE AS A COPY

```
x = [1, 2, 3]

foo(x) = sum(x[1:2])          # it allocates ONE vector -> the slice 'x[1:2]'
```

```
julia> @btime foo($x)
  15.015 ns (1 allocation: 80 bytes)
```

SLICE AS A VIEW

```
x = [1, 2, 3]

foo(x) = sum(@view(x[1:2]))    # it doesn't allocate
```

```
julia> @btime foo($x)
  1.200 ns (0 allocations: 0 bytes)
```

However, **views created through Boolean indexing neither reduce memory allocations nor are more performant**. Therefore, you shouldn't rely on views of these objects to speed up computations. This fact is illustrated below.

BOOLEAN INDEX (COPY)

```
x = rand(1_000)

foo(x) = sum(x[x .> 0.5])
```

```
julia> @btime foo($x)
  662.500 ns (4 allocations: 8.34 KiB)
```

BOOLEAN INDEX (VIEW)

```
x = rand(1_000)

foo(x) = @views sum(x[x .> 0.5])
```

```
julia> @btime foo($x)
  759.770 ns (4 allocations: 8.34 KiB)
```

COPYING DATA MAY BE FASTER

Although views can reduce memory allocations, there are scenarios where copying data can result in faster performance. A detailed comparison of copies versus views will be provided in Part II. Here, we simply remark on this possibility.

Essentially, the choice between copies and views reflects a fundamental trade-off between memory allocation and data access patterns. On the one hand, newly created vectors store data in contiguous blocks of memory, enabling more efficient CPU access and allowing for certain optimizations. On the other hand, views avoid allocation, but may also require accessing data scattered across non-contiguous memory regions.

Below, we illustrate a scenario in which the overhead of creating a copy is outweighed by the benefits of contiguous memory access, making copying the more efficient choice.

COPY

```
x = rand(100_000)

foo(x) = max.(x[1:2:length(x)], 0.5)
```

```
julia> @btime foo($x)
  30.100 μs (4 allocations: 781.34 KiB)
```

VIEW

```
x = rand(100_000)

foo(x) = max.(@view(x[1:2:length(x)]), 0.5)
```

```
julia> @btime foo($x)
  151.700 μs (2 allocations: 390.67 KiB)
```