9d. Slice Views to Reduce 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 satisfy a specific condition. 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.

Next, we address the memory allocations associated with slices. We do this by highlighting the role of **views**, which bypass the need for a copy by directly referencing the parent object. This method is particularly effective when slices are indexed through ranges. However, it's not suitable for slices that employ Boolean indexing, like `x[x .> 0]`, where memory allocation will still occur despite using a view.

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 start showing that views don't allocate memory *if the slice is indexed by a range*. This property can lead to performance improvements over regular slices, which create a copy 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 under Boolean indexing won't reduce memory allocations or be more performant**. Therefore, don'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 be the faster approach. This is due to an inherent 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. On the other hand, while views avoid allocation, they require accessing data scattered throughout memory.

In certain cases, the overhead of creating a copy may be outweighed by the benefits of contiguous memory access, making copying the more efficient choice. This possibility is illustrated below.

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)
```