Why is Julia so fast?

For a more technical explanation, see here.

Why is C fast?

Languages like C, C++ and Fortran are generally the fastest in terms of execution speed, and as a result are the most popular languages for applications where performance is critical, in particular for intensive calculations on supercomputers, where C and Fortran rule as kings.

These three languages share a certain number of characteristics in their design, that are at the source of their performance. However, C++ is significantly more sophisticated than the other two.

- **Compilation** The source code of a program is turned into machine code by a special program called the *compiler*, which usually performs a certain number of transformations to optimize the resulting machine code.
- **Static Typing** Every expression has a well-defined type at compile time. This property is important, as the compiler uses type information for its optimizations.
- **Manual memory management** Simple expressions aside, the programmer must manage himself the allocation of memory in his programs. This avoids the overhead of using a *garbage collector* to manage memory automatically, which can be costly during execution. This is a double-edged sword, however, as manually managing memory is notoriously complicated and a source of bugs and software vulnerabilities.

At high level, we can describe the strategy of these three languages as offloading as much of the evaluation of the program to the compiler.

The counterpart of these performances is that these languages can be complicated to use and leave many opportunities for the programmer to make mistakes that he will only discover during or after compilation.

Why are Python and Matlab so slow?

Unlike C and Fortran, languages like Python, Matlab or R are rather slow, but are a lot more flexible and easy to use, which allows the programmer to save time when writing programs.

The characteristics of these languages are

- **Interpretation** Rather than going through a compiler, source code is read by a program called an *interpreter* which evaluates the program line by line and evaluates them progressively. A consequence of this is less room for optimizing the program.
- **Dynamic Typing** The type of an expression is only known at runtime, which again leaves less opportunities for optimization.
- **Automatic memory management** Contrary to C and Fortran, Python and Matlab automatically allocate and free memory during the execution of the program. This is done in part by a program called *garbage collector*, which watches the memory space used by the program and frees the unused parts. The garbage collector runs alongside the user program, which requires additional computing power.

Traditionally, using Python or Matlab in spite of the loss of performance is a calculated choice. The hours spent by the programmer writing and debugging code are almost universally more expensive than the time it takes to run the program. Furthermore, not having the compilation step allows for rapid prototyping, evaluation and correction of the program. This shorter feedback loop generates a non-trivial productivity gain for the programmer.

When performance is really needed, Python and Matlab use functions that rest on implementation in a fast language (This is why Python and Matlab encourage using vectorized functions instead of loops). This approach is limited however, and in extreme cases, the only solution is to completely rewrite the program in a more performant language.

The importance of types

Up until now, we have talked about the distinction between static and dynamic typing, and suggested that static typing is inherently faster than dynamic typing. The situation is more nuanced however, as other properties of the type system need to be taken into account.

In particular, another aspect of type systems is that they allow verifying that a piece of program is correct (free of syntax errors) and prove assertions about it.

This last point is not an analogy. To perform its optimizations, the compiler needs to be able to prove that they don't modify the behavior of the program. The types of a program then serve as propositions in a formal language that allow the compiler to prove assertions about it. For certain languages with a very sophisticated type system like Coq, the compiler is essentially a theorem prover.

A well designed type system is therefore important, as it enables the compiler to

- Prove more properties of the program, and hence perform more optimizations
- Prove/verify that the program is correct, which allows avoiding errors and vulnerabilities. Furthermore, when an error arises at runtime, the information about the types of the piece of code that generated the error is often useful to debug it.

Why is Julia fast? A cake in three ingredients

At first glance, Julia's performance makes no sense. It is used like an interpreted language, and it is not necessary to write down the type of every variable (which means Julia is dynamically typed). And yet, Julia regularly displays performances comparable to C or C++, despite using a garbage collector, and is at least an order of magnitude faster than Python or Matlab. The secret behind this sorcery lies in Julia's design and has at least three aspects.

JIT Compilation, the flour

In spite of appearances, Julia is technically a *compiled* language. However, it uses a particular evaluation strategy called *Just In Time Compilation* (JIT). Its principle is the following

When a Julia function is called, the Julia runtime checks if that function has already been called with arguments of the same type. If it hasn't, the Julia compiler is called to generate an optimized version of the function for these types of arguments. This optimized function (which we'll call

method from now on) is then stored for later use when the function is called with the same argument types again.

The advantage of this strategy is that it allows benefitting from compiler optimizations without having to compile the whole program in advance. Only the necessary parts of the program are compiled. The disadvantage is that compilation can take some time when starting the program. This problem is known in the Julia community as the *Time to first plot*.

Note that the Julia compiler is only called to optimize functions. This point is important, and it is always recommended using as many functions as possible in Julia programs.

The type system, the butter

While JIT compiling is already responsible for a good chunk of Julia's performance, it doesn't explain everything. Indeed, both Python and Matlab have JIT implementations, yet are still much slower than Julia. The reason being that their types systems weren't designed for performance.

Type systems are designed to facilitate and augment the compiler. To make this clear, we will review the main aspects of Julia's type system.

Primitive Types

As any language that is even remotely performant, Julia possesses the usual low level types such as integers and floating point numbers (in 32 and 64 bits), characters, booleans, ... These types are called *primitive* because they're directly associated to a representation taking a certain amount of bits in memory.

Arrays

To represent an array of objects, such as a vector or matrix, Julia uses the type `Array`, which as in C, corresponds to a block of memory where the elements of the array are lined up in order. For example, the type of a vector of 32 bits integers in Julia is `Vector{Int32}`. In general, the type of an array of dimension `N` with element type `T` is denoted `Array{T,N}` (`Vector{T}` is actually just a shorthand for `Array{T,1}`). The fact that all elements are of the same type, and are arranged contiguously in memory is important for performance, as it means every element takes up the same number of bits, and when accessing the elements of the array sequentially, the processor can load elements close to one another in advance.

Composite Types

To represent a composite type, which C programmers know as a *struct*, Julia does the same thing as C or C++. The different fields of the composite type are arranged sequentially in a single block of memory. For instance, a type `ComplexF64` containing two fields of type `Float64` to represent the real and imaginary parts of a complex number will be represented as a block of memory where the real part and imaginary part are stored next to one another. The advantages of this representation are the same as for arrays.

Parametric Types

The previous example with `ComplexF64` illustrates a recurrent problem in C. If, for some reason, the programmer wanted to have a complex number type with 32 bits floats or integers, he would have to rewrite a structure exactly similar to that of `ComplexF64`, but with the other type instead of `Float64`. Not only that, he would have to rewrite all the functions he had written for `ComplexF64` for the new type, even though the code would essentially be the same.

Ideally, we would like to write our functions only once by abstracting over the exact type used for the real and imaginary parts. For this, Julia uses *parametric types* (also known as generic types). A parametric type is of the form `MyType{T}`, where `T` is a type parameter (it can be thought of as a variable whose value is a type). For example, the parametric form of the complex number type in Julia is `Complex{T}`. W can now write our functions for `Complex{T}`, and they will automatically work for `Complex{Float64}` or `Complex{Int32}`!

To finish this section, let us note that the type `Array{T,N}` seen previously is an example of a parametric type. It is however a bit special because the parameter `N` is not a type, but an integer.

Abstract types and type hierarchy

Until now, we have only seen concrete types, directly linked to a memory representation. Most of the time though, Julia programmers manipulate *abstract types*. An abstract type is not tied to a memory representation, and thus can't be instantiated at runtime.

What are these types used for? To answer this, we need to introduce the other element which is a *partial order relation* on types. Julia uses the notation `S <: T` to say that type `S` is a *subtype* of type `T`. A type can have an arbitrary amount of subtypes and supertypes, except concrete types which cannot have any subtypes. Finally, there is a type called `Any`, which is a supertype of every type (and has no other supertype other than itself).

This order structure on types can be visualized as a tree, like in the following figure, which shows the subtypes and supertypes of the type `AbstractFloat`, which is the abstract type for all floating point numbers. On the bottom, we see the concrete types `Float64`, `Float16`, `BigFloat` and `Float32`, and on the top, we see that `AbstractFloat` is a subtype of `Real`, which is a subtype of `Number`, which is itself a subtype of `Any`.

As one might guess, `Real` is the abstract type for representing real numbers (integers, rationals, ...), while `Number` is the abstract type for representing generic numbers (e.g. complex numbers, but also more exotic structure like tropical numbers)

The order relation between types is used by the compiler to choose which function to apply for a given call. Indeed, Julia applies the *Liskov substitution principle*, which states that every instance of a subtype `S` of type `T` should be usable where any object of type `T` is used, without changing the behavior of the program. In other words, any function defined for type `T` is automatically defined for every subtype of `T`.

Liskov's principle therefore allows us to define high level functions on abstract types, which will work for a large number of concrete types, without having to reimplement it for each type.

The other advantage is that the hierarchy of abstract types allow us to distinguish the axioms we

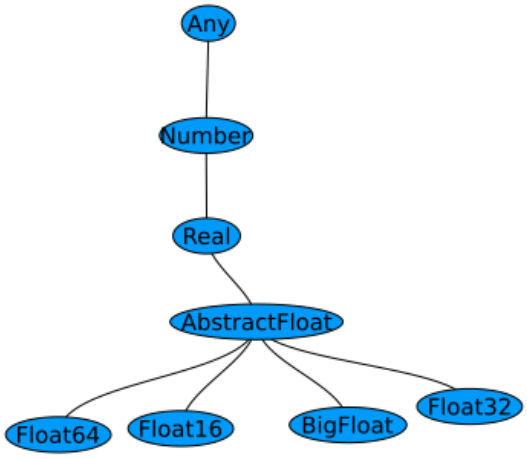

Figure 1: Subtypes and supertypes of `AbstractFloat`

make on our types. For instance, the type `Number` is used to represent objects with algebraic composition laws (addition and multiplication) that verify certain well-known axioms (associativity, distributivity, neutral element, invertibility, ...). The programmer is then encouraged to think about the minimal axioms needed for his algorithms, and implement them at the most general level possible.

Let us note however that these axioms are not formally verified by the Julia compiler, but rather depend on the programmer. In spite of this, there are many examples where functions written at a general level by one programmer worked out of the box on another programmer's types, without modification!

Multiple Dispatch and Type Stability, the eggs

After this lengthy description of Julia's type system, we can now cover two properties that allow the Julia compiler to exploit information about types to optimize programs written by the developer. These are *Multiple Dispatch* and *Type Stability*.

Multiple dispatch is at the heart of JIT compiling. When compiling a function call for a particular combination of input types, the compiler seeks the most specific method for that particular combination of types.

For example, suppose we want to compute the determinant of a square matrix. The Julia standard library defines a general matrix type, as well as specialized types such as `Diagonal` or `Triangular`, all being subtypes of `AbstractMatrix`. Upon evaluating the determinant of a matrix `A` with `det(A)`, if `A` is of type `Diagonal` or `Triangular`, its determinant will be

computed using the appropriate algorithm for that matrix type, otherwise, the default method inherited from `AbstractMatrix` will be used.

Multiple dispatch therefore allows us to define a generic implementation on abstract types, while letting the programmer define specialized versions of this function for more specific types.

The other property, type stability, is more complicated to explain, but essentially boils down to the idea that the compiler must be able to prove properties about the program it is optimizing. Formally, a function is said to be *type stable* if the return type of that function can be inferred from the types of its arguments.

Type stability is extremely important for the performance of compiled code. Indeed, if the return type of a function cannot be determined by the compiler, it has to be determined at runtime, which has a disastrous effect on the performance of the function. Even worse, if a function is type unstable, the problem can spread to other functions that call it and so on, recursively.

Just like the axioms on abstract types, type stability is not a formal constraint. It is perfectly possible to write type unstable code, although it is not recommended (there are some situations where type instability cannot be avoided however). Thankfully, Julia's design makes it so that outside of these special cases, writing type unstable code is harder than writing type stable code.

Semantic sugar, and the cherry on the cake...

The elements we have just discussed, JIT compilation, the type system, multiple dispatch and type stability are the source of most of Julia's performance. There are however a few additional features that allow the programmer to maximize the performance of his programs.

Indeed, Julia provides several tools to evaluate a program's performance. For instance, the macro `@time`, which allows to measure the execution time and memory allocated during a function call. It is also possible for the programmer to create his own tools. For instance, the `BenchmarkTools` package provides macros to rigorously evaluate code performance.

There are also macros for inspecting the compiled code for a function call. In particular, the macro `@code_warntype` displays the compiled code in an intermediate language with inferred types and highlights type instabilities.

Finally, it is also possible, again with macros, to squeeze a lit bit more speed by sacrificing some safety. For example, the macro `@inbounds` intervenes when indexing an array, and disables bounds checking.

We have just discussed objects called *macros* without defining them. The term is familiar to C programmers, where it refers to substitution rules defined by the programmer, which are applied during compilation. Julia macros are far more sophisticated that this. They're actually special functions that take Julia expressions as arguments and return another expression.

To make things clearer, it is useful to introduce the more general notion of *first class citizens*. In a given programming language, a first class citizen is a class of objects that

1. Can be used as function arguments
2. Can be used the result of a function
3. Can be assigned to variables

4. Can be compared with an equality operator

We can add a fifth property of being able to create these objects in an expression without assigning them to a variable (one may speak of anonymous variables).

In C and Fortran, only instances of primitive or composite types are first class citizens. Neither arrays, nor functions satisfy all these properties, although some of them are through the use of pointers (which are notoriously complex to use).

In contrast, Julia has many first class citizens. Arrays and any instance of a type are trivially first class, but so are functions (via the `Function` type; 1. and 2. are satisfied by so-called *higher order functions*, 5. by *lambda expressions*), types (there is a `Type` type whose instances are types) and Julia expression (via the `Expr` type and macros)

Having these three classes of objects as first class citizens confers considerable expressivity to Julia, as it allows the programmer and the compiler to reason about them. The macro system in particular is extremely powerful, to the point of allowing the programmer to directly modify the compiler. Very few languages allow such introspection, and there already multiple Julia packages that make use of macros to inspect code execution, measure performance in detail or even completely change the way code is interpreted.

Finally, let's close this section with one of the most beautiful consequences of Julia's design. In modern scientific computing, it is extremely common to use parallel or massively parallel processors, be it through multicore processors, clusters or GPUs. Writing and debugging parallel programs is notoriously complex, not only on a conceptual level, but even more so for languages like C or Fortran, which were designed before the rise of parallel computing.

For these reasons, a new generation of programming languages has appeared in the last 20 years (e.g. D, Go, Rust, Nim), that all aim to correct for the flaws of C and C++, while keeping their speed, most notably by providing native support for concurrent and parallel programming.

In many ways, Julia is part of that new generation, and cleverly exploits its design to make parallel computing almost trivial. Thus, the Julia standard library provides the `@threads` macro for executing a loop in parallel on multiple threads in the following way

```
@threads for i in 1:length(xs)
    xs[i] = foo(xs[i])
end
```

Similarly, the `@distributed` macro lets us do the same on a cluster

```
@distributed for i in length(xs)
    xs[i] = foo(xs[i]) # NB. Restrictions apply
end
```

This last case is facilitated by the `DistributedArrays` package, which provides a type for arrays distributed over the cluster.

Finally, the `CUDA` package provides the `CuArray` type, which represents an array stored in a GPU (Nvidia only). This type works exactly like a normal array, which makes it possible to automatically use most user functions directly on the GPU without modification. This is the case for example

with `DifferentialEquations`, where all it takes to solve an ODE on the GPU is to put the initial condition in a `CuArray`.

Conclusion

Over the course of this document, we have gone in great detail over what makes the Julia programming language as fast as it is, namely JIT compiling, supported by a particularly well-designed type system and multiple dispatch. This is of course, only half of the equation, as the language provides flexibility and modularity, as well as fantastic tools to assess the performance of a program.

The end result is that not only is Julia fast, it also makes using complicated features such as parallelism as trivial and painless as one could possibly imagine. While it's still possible to write slow code in Julia, the design of the language makes it harder than writing efficient code.