



# Introduction

Scientific computing has traditionally required the highest performance, yet domain experts have largely moved to slower dynamic languages for daily work. We believe there are many good reasons to prefer dynamic languages for these applications, and we do not expect their use to diminish. Fortunately, modern language design and compiler techniques make it possible to mostly eliminate the performance trade-off and provide a single environment productive enough for prototyping and efficient enough for deploying performance-intensive applications. The Julia programming language fills this role: it is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages.

Because Julia's compiler is different from the interpreters used for languages like Python or R, you may find that Julia's performance is unintuitive at first. If you find that something is slow, we highly recommend reading through the Performance Tips section before trying anything else. Once you understand how Julia works, it's easy to write code that's nearly as fast as C.

Julia features optional typing, multiple dispatch, and good performance, achieved using type inference and just-in-time (JIT) compilation, implemented using LLVM. It is



multi-paradigm, combining features of imperative, functional, and object-oriented programming. Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby.

The most significant departures of Julia from typical dynamic languages are:

- The core language imposes very little; the standard library is written in Julia itself, including primitive operations like integer arithmetic
- A rich language of types for constructing and describing objects, that can also optionally be used to make type declarations
- The ability to define function behavior across many combinations of argument types via multiple dispatch
- Automatic generation of efficient, specialized code for different argument types
- Good performance, approaching that of statically-compiled languages like C

Although one sometimes speaks of dynamic languages as being "typeless", they are definitely not: every object, whether primitive or user-defined, has a type. The lack of type declarations in most dynamic languages,



however, means that one cannot instruct the compiler about the types of values, and often cannot explicitly talk about types at all. In static languages, on the other hand, while one can – and usually must – annotate types for the compiler, types exist only at compile time and cannot be manipulated or expressed at run time. In Julia, types are themselves run-time objects, and can also be used to convey information to the compiler.

While the casual programmer need not explicitly use types or multiple dispatch, they are the core unifying features of Julia: functions are defined on different combinations of argument types, and applied by dispatching to the most specific matching definition. This model is a good fit for mathematical programming, where it is unnatural for the first argument to "own" an operation as in traditional object-oriented dispatch. Operators are just functions with special notation – to extend addition to new user-defined data types, you define new methods for the + function. Existing code then seamlessly applies to the new data types.

Partly because of run-time type inference (augmented by optional type annotations), and partly because of a strong focus on performance from the inception of the project, Julia's computational efficiency exceeds that of other dynamic languages, and even rivals that of statically-compiled languages. For large scale numerical problems, speed always has been, continues to be, and probably always will be crucial: the amount of data being processed has easily kept pace with Moore's Law over the past decades.

# The Julia Language

stable



Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. In addition to the above, some advantages of Julia over comparable systems include:

- Free and open source (MIT licensed)
- User-defined types are as fast and compact as built-ins
- No need to vectorize code for performance; devectorized code is fast
- Designed for parallelism and distributed computation
- Lightweight "green" threading (coroutines)
- Unobtrusive yet powerful type system
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- Call C functions directly (no wrappers or special APIs needed)
- Powerful shell-like capabilities for managing other processes
- Lisp-like macros and other metaprogramming facilities

Previous: Home Next: Getting Started