What is Julia and why is it Gaining Popularity?

ABSTRACT

The paper reviews what the Julia programming language is, and if it has staying power in comparison to other popular programming languages. The main points of the paper go in to detail on two other popular programming languages, C programming and Python programming, to obtain a deeper understanding of what Julia uses from them to make it a more reasonable choice for performance. It also has more details as to where Julia differs from these two languages.

CCS CONCEPTS

KEYWORDS

ACM proceedings, text tagging

ACM Reference format:

G. Gubbiotti, P. Malagò, S. Fin, S. Tacchi, L. Giovannini, D. Bisero, M. Madami, and G. Carlotti. 1997. SIG Proceedings Paper in word Format. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK’97)*, 4 pages.

DOI: 10.1145/123 4

1 INTRODUCTION

According to the NumFocus Project, "Julia is a high-level, high-performance dynamic programming language" developed by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Created in 2012, it was designed to do numerical computing. "Numerical computing is an interconnected combination of computer science and mathematics in which we develop and analyze algorithms for solving important problems in science, engineering, medicine, and business" [2]. Julia also provides "a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library" [1]. "It also integrates open source C and Fortran libraries for many functions such as the use of linear algebra, random number generation, signal processing, and string processing" [1]. It not only borrows from the C programming language, but it also incorporates from other popular dynamic languages like "Python, Perl, Lisp, Lua, and Ruby" [1].

A dynamic programming language are "those languages in which variables must necessarily be defined before they are used. " [6]. 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

[3]

The goal for Julia is to provide "an unprecedented combination of ease-of-use, power, and efficiency in a single language." [3]. The number of advantages that Julia brings is clear evidence as to why it is growing in popularity at such a fast rate.

From what already has been mentioned, "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"

[3]

According to the TIOBE index, it broke the top 50 in September of 2016, and as of November 2017, it currently rests at number 35 in rank right below VBScript, Hack, Fortran, and Rust to name a few [6]. As of September 2017, it has obtained over 2 million downloads and shows spikes of interest in job postings on indeed.com [7]. This brings the question into development, what is Julia and why is it gaining popularity?

2 Inspiration for Julia

As mentioned earlier, Julia incorporates the convenience of dynamic languages like Python, and an integration of open source C. To simply put it, the inspiration for Julia is to have a language that "walks like python, and runs like C", says Junolab [7].

2.1 The history of Python

"Python is an interpreted, interactive, object-oriented programming language that incorporates modules, exceptions, dynamic typing, very high level dynamic types, and classes into it", says Programming Python and Internet Programming with Python [8]. Python is adored by programmers for its remarkable power and a syntax that is crystal clear. The language was released by Guido Van Rossum in February of 1991. It originated from an interpreted language called ABC. Rossum had a goal to fix errors that were in that language and at the same time keep features that were likeable in it. He was at work on an operating system called Amoeba distributed and was wanting a scripted language with syntax like ABC but with the Amoeba system calls, so he made his own. He was driven to do so for the fact that he was a one-man army on the project and was not getting too much attention on it. Thus in 1989 around Christmas time, he created what we now know as Python. [8].

2.2 Python Programming Philosophy

What can be noted from the history of Python, is the fact that Rossum was in a situation where he was in a one-person project with no official budget, so he wanted results quickly in regard to getting his management's attention to support what he was doing. Straight from a blog that he posted back in January of 2009, it states one of the biggest drives was to inherit time saving principles, the most notable being" keeping things simple", and following the UNIX philosophy of "doing one thing well". He had a couple other key elements that led to the success of python by primarily sticking to the core idea of not striving to reach for perfection. These are some of the fallbacks that he learned while gaining experience with ABC that he wanted his language to avoid. If it was not for that lesson he gained, the large success of Python would not be what it is today with such a powerful user base that encouraged its improvements as opposed for him striving to reach it on his own. [9]

2.3 Limitations of Python

There are a few limitations on Python, but the ones that should be paid attention to for the case of understanding Julia, is that it can get slow whenever it tries to execute the code. The reason being is because Python executes through the help of an interpreter instead of the compiler itself, which in return causes it to run slow. Another key limitation to pay attention to is it seems that the database access layer comes across as underdeveloped which can possible be an issue. This means that it can be an issue to obtain data from a database that the user could be trying to use to run a data driven numerical process, for example [10].

2.4 The history of C Programming

C programming is one of the older programming languages being developed back in Bell laboratories in 1972 by Dennis Ritchie. It was inspired by the earlier language entitled B programming and other early ancestors to that, BCPL, and CPL (Combined Programming Language). With the acknowledgment of the limitations that B programming has, Dennis then began development of C. The language ended up being so powerful that the UNIX operating system was completely re-written in C. Prior to that it was written in assembly language. Throughout the 1970's C spread among colleges and universities because of that close tie to UNIX and the availability of C's compilers. Once that became predominant, many companies began incorporating C within them, but doing so caused some issues in compatibility since they were all making their own versions of C. This lead to the creation of the of ANSI standard C, a standard definition of C. This was created by the American National Standards Institute (ANSI) in 1983 [11]. This helped the already popular programming language grow in to what it is now today and is still ranked as the 2nd most popular programming language, right below Java [5].

2.5 C Programming Philosophy

The basic philosophy behind C was to fix the divide between the assembly language (low level) and the high-level language. This allows for the user to have more control of the bits in the machine and to manipulate data in a simpler manner. Because of this behavior is why C programming is still so popular today. With this design philosophy in mind is what allowed it to obtain the speeds that it is so popular for [11].

2.6 Strengths of C programming

There are multiple advantages to the C programming language, thus why it has been so popular for such a long period of time. The biggest ones to focus on if we are talking about Julia, is that it is very structured. This means that it allows for a "complex program to be broken into simpler programs called functions" That basically defines what C is in essence, a big library of functions. These functions can then be used to accomplish whatever task the user may be trying to do. This adds for simplicity. It helps the user think of what is being presented to them as building blocks. The user is presented with them and makes a program with them in whatever way the choose to put these blocks or modules [12].

2.7 Limitations of C Programming

The disadvantage of C is that it does not have a sense of Object Oriented Programming. C++ was developed in order to support this problem. Other notable disadvantages would be there is no runtime checking in the language, meaning it only checks during the time it is compiling the code. This is saying that it cannot be certain until compilation if the correct data type is being used, so it performs automatic type conversion [C3].

3 SYTHESIS OF C AND PYTHON TECHNIQUES

Now that there should be a clearer understanding of C programming and Python, what comparatively is it that Julia differs from these two languages?

3.1 Differences to Python

According to the Julia documentation, the most noteworthy differences from Python are as follows:

* Julia requires end to end a block. Unlike Python, Julia has no pass keyword.
* In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
* Julia's slice indexing includes the last element, unlike in Python. a[2:3] in Julia is a[1:3] in Python.
* Julia does not support negative indexes. In particular, the last element of a list or array is indexed with end in Julia, not -1 as in Python.
* Julia's for, if, while, etc. blocks are terminated by the end keyword. Indentation level is not significant as it is in Python.
* Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
* Julia arrays are column major (Fortran ordered) whereas NumPy arrays are row major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy.
* Julia's updating operators (e.g. +=, -=, ...) are not in-place whereas NumPy's are. This means A = ones(4); B = A; B += 3 doesn't change values in A, it rather rebinds the name B to the result of the right- hand side B = B + 3, which is a new array. Use B[:] += 3, explicit loops, or InplaceOps.jl.
* Julia evaluates default values of function arguments every time the method is invoked, unlike in Python where the default values are evaluated only once when the function is defined. For example, the function f(x=rand()) = x returns a new random number every time it is invoked without argument. On the other hand, the function g(x=[1,2]) = push!(x,3) returns [1,2,3] every time it is called as g().
* In Julia % is the remainder operator, whereas in Python it is the modulus.

[14]

3.2 Differences to C Programming

The Julia documentation takes into account noteworthy differences in both C and C++, which as mentioned earlier, was developed to support Object Oriented Programming. Listed below are these differences in comparison to the Julia Language:

* Julia arrays are indexed with square brackets, and can have more than one dimension A[i,j]. This syntax is not just syntactic sugar for a reference to a pointer or address as in C/C++.
* In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
* Julia arrays are assigned by reference. After A=B, changing elements of B will modify A as well. Updating operators like += do not operate in-place, they are equivalent to A = A + B which rebinds the left-hand side to the result of the right-hand side expression.
* Julia arrays are column major (Fortran ordered) whereas C/C++ arrays are row major ordered by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to C/C++ .
* Julia values are passed and assigned by reference. If a function modifies an array, the changes will be visible in the caller.
* In Julia, whitespace is significant, unlike C/C++, so care must be taken when adding/removing whitespace from a Julia program.
* In Julia, literal numbers without a decimal point (such as 42) create signed integers, of type Int, but literals too large to fit in the machine word size will automatically be promoted to a larger size type, such as Int64 (if Int is Int32), Int128, or the arbitrarily large BigInt type. There are no numeric literal suffixes, such as L, LL, U, UL, ULL to indicate unsigned and/or signed vs. unsigned. Decimal literals are always signed, and hexadecimal literals (which start with 0x like C/C++), are unsigned. Hexadecimal literals also, unlike C/C++/Java and unlike decimal literals in Julia, have a type based on the length of the literal, including leading 0s. For example, 0x0 and 0x00 have type [UInt8](https://docs.julialang.org/en/stable/stdlib/numbers/#Core.UInt8), 0x000 and 0x0000 have type [UInt16](https://docs.julialang.org/en/stable/stdlib/numbers/#Core.UInt16), then literals with 5 to 8 hex digits have type UInt32, 9 to 16 hex digits type UInt64 and 17 to 32 hex digits type UInt128. This needs to be taken into account when defining hexadecimal masks, for example ~0xf == 0xf0 is very different from ~0x000f == 0xfff0. 64 bit Float64 and 32 bit [Float32](https://docs.julialang.org/en/stable/stdlib/numbers/#Core.Float32) bit literals are expressed as 1.0 and 1.0f0 respectively. Floating point literals are rounded (and not promoted to the BigFloat type) if they can not be exactly represented. Floating point literals are closer in behavior to C/C++. Octal (prefixed with 0o) and binary (prefixed with 0b) literals are also treated as unsigned.
* String literals can be delimited with either " or """, """ delimited literals can contain " characters without quoting it like "\"" String literals can have values of other variables or expressions interpolated into them, indicated by $variablename or $(expression), which evaluates the variable name or the expression in the context of the function.
* // indicates a [Rational](https://docs.julialang.org/en/stable/stdlib/numbers/#Base.Rational) number, and not a single-line comment (which is # in Julia)
* #= indicates the start of a multiline comment, and =# ends it.
* Functions in Julia return values from their last expression(s) or the return keyword. Multiple values can be returned from functions and assigned as tuples, e.g. (a, b) = myfunction() or a, b = myfunction(), instead of having to pass pointers to values as one would have to do in C/C++ (i.e. a = myfunction(&b).
* Julia does not require the use of semicolons to end statements. The results of expressions are not automatically printed (except at the interactive prompt, i.e. the REPL), and lines of code do not need to end with semicolons. [println()](https://docs.julialang.org/en/stable/stdlib/io-network/#Base.println) or [@printf()](https://docs.julialang.org/en/stable/stdlib/io-network/#Base.Printf.@printf) can be used to print specific output. In the REPL, ; can be used to suppress output. ; also has a different meaning within [ ], something to watch out for. ; can be used to separate expressions on a single line, but are not strictly necessary in many cases, and are more an aid to readability.
* In Julia, the operator [⊻](https://docs.julialang.org/en/stable/stdlib/math/#Base.xor) ([xor](https://docs.julialang.org/en/stable/stdlib/math/" \l "Base.xor" \t "_blank)) performs the bitwise XOR operation, i.e. [^](https://docs.julialang.org/en/stable/stdlib/strings/#Base.:^-Tuple{AbstractString,Integer}) in C/C++. Also, the bitwise operators do not have the same precedence as C/++, so parenthesis may be required.
* Julia's [^](https://docs.julialang.org/en/stable/stdlib/strings/#Base.:^-Tuple{AbstractString,Integer}) is exponentiation (pow), not bitwise XOR as in C/C++ (use [⊻](https://docs.julialang.org/en/stable/stdlib/math/#Base.xor), or [xor](https://docs.julialang.org/en/stable/stdlib/math/#Base.xor), in Julia)
* Julia has two right-shift operators, >> and >>>. >>> performs an arithmetic shift, >> always performs a logical shift, unlike C/C++, where the meaning of >> depends on the type of the value being shifted.
* Julia's -> creates an anonymous function, it does not access a member via a pointer.
* Julia does not require parentheses when writing if statements or for/while loops: use for i in [1, 2, 3] instead of for (int i=1; i <= 3; i++) and if i == 1 instead of if (i == 1).
* Julia does not treat the numbers 0 and 1 as Booleans. You cannot write if (1) in Julia, because if statements accept only booleans. Instead, you can write if true, if Bool(1), or if 1==1.
* Julia uses end to denote the end of conditional blocks, like if, loop blocks, like while/ for, and functions. In lieu of the one-line if ( cond ) statement, Julia allows statements of the form if cond; statement; end, cond && statement and !cond || statement. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. cond && (x = value), because of the operator precedence.
* Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
* Julia macros operate on parsed expressions, rather than the text of the program, which allows them to perform sophisticated transformations of Julia code. Macro names start with the @ character, and have both a function-like syntax, @mymacro(arg1, arg2, arg3), and a statement-like syntax, @mymacro arg1 arg2 arg3. The forms are interchangable; the function-like form is particularly useful if the macro appears within another expression, and is often clearest. The statement-like form is often used to annotate blocks, as in the parallel for construct: @parallel for i in 1:n; #= body =#; end. Where the end of the macro construct may be unclear, use the function-like form.
* Julia now has an enumeration type, expressed using the macro @enum(name, value1, value2, ...) For example: @enum(Fruit, banana=1, apple, pear)
* By convention, functions that modify their arguments have a ! at the end of the name, for example push!.
* In C++, by default, you have static dispatch, i.e. you need to annotate a function as virtual, in order to have dynamic dispatch. On the other hand, in Julia every method is "virtual" (although it's more general than that since methods are dispatched on every argument type, not only this, using the most-specific-declaration rule).

[14]

4 CONCLUSIONS

In summary, based upon our finding in comparison to some of the most popular programming languages in the market, there is some room for Julia to continue making headway. Determining factors for this language to stay predominant and climb up the ladder of popularity will be maintaining a loyal fanbase. This can be done by continuing to have fluent documentation on how to use the language and documentation showing the user how to reach the performance needs as promised. Another factor would be continuing to hold the promise of having greater performance speeds in comparison to C and Python by using them together to break the boundaries each programming language holds. For scientific computing, Julia may very well take over if it continues to keep its promise of performance, and continues to improve by continuing to release stable builds of the language for everyone to take advantage of.

A HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are di.erent in the appendices. In the appendix environment, the command section is used to indicate the start of each Appendix, with alphabetic order designation (i.e., the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure within an Appendix, start with subsection as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.3 Results and Discussion

A.4 Conclusions

A.5 References

ACKNOWLEDGMENTS

REFERENCES

|  |  |
| --- | --- |
| [1] | https://julialang.org/ |
| [2] | https://cs.nyu.edu/courses/spring12/CSCI-UA.0421-001/index.html |
| [3] | https://docs.julialang.org/en/stable/manual/introduction/ |
| [4] | https://en.wikipedia.org/wiki/Julia\_(programming\_language) |
| [5] | https://www.tiobe.com/tiobe-index/ |
| [6] | https://www.sitepoint.com/typing-versus-dynamic-typing/ |
| [7] | https://juliacomputing.com/blog/2017/02/03/newsletter.html |
| [8] | http://groups.engin.umd.umich.edu/CIS/course.des/cis400/python/python.html |
| [9] | http://python-history.blogspot.com/2009/01/pythons-design-philosophy.html |
| [10] | https://medium.com/@mindfiresolutions.usa/advantages-and-disadvantages-of-python-programming-language-fd0b394f2121 |
| [11] | <http://groups.engin.umd.umich.edu/CIS/course.des/cis400/c/c.html> |
| [12] | https://www.invensis.net/blog/it/benefits-of-c-c-plus-plus-over-other-programming-languages/ |
| [13] | <http://www.techcrashcourse.com/2015/11/Disadvantages-and-limitations-of-C-Programming-languages.html> |
| [14] | https://docs.julialang.org/en/stable/manual/noteworthy-differences/ |
| [15] | https://www.ibm.com/developerworks/community/blogs/jfp/entry/A\_Comparison\_Of\_C\_Julia\_Python\_Numba\_Cython\_Scipy\_and\_BLAS\_on\_LU\_Factorization?lang=en |