Visualizing the Julia set with Julia

Philip Nelson

***Abstract*—This paper evaluates the strengths and weaknesses of the Julia language by analyzing the design characteristics of the Julia language and their impact on readability, write-ability, cost, and performance. The domain where the Julia language is best suited will also be discussed.**

Introduction

In 2009, four men, Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman, came together to create a language that was high level and took many aspects from Ruby, Lisp, Matlab, Python, R and Perl but was as performant as C [1]. Today Julia's designers describes Julia as "a high-level, high-performance dynamic programming language for numerical computing. It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library.

Julias Base library, largely written in Julia itself, also integrates mature, best-of-breed open source C and FORTRAN libraries for linear algebra, random number generation, signal processing, and string processing. In addition, the Julia developer community is contributing a number of external packages through Julias built-in package manager at a rapid pace. IJulia, a collaboration between the Jupyter and Julia communities, provides a powerful browser-based graphical notebook interface to Julia. [2]" From this description, found on Julia's home page, it can be seen that Julia is a modern language which it's authors hoped it would be. After having written a program in Julia that visualizes the Julia set fractal, I will comment on Julia as a programming language.

The Julia REPL

As I began learning Julia, having access to a read eval print loop was fantastic. When I wanted to experiment with a new language feature, I could easily open the repl and test it out. Having access to the repl removed the need to recompile and run the program repeatedly just to lean how language feature worked. The repl improves write-ability and development cost because it allows the developer to interact with their code easily and test it interactively. I found that it helped me find bugs more easily because I could load the file I was writing in the repl and run and test individual functions.

Variable Names

Julia allows flexibility in naming variables. Variable names are case-sensitive have no semantic meaning, and must begin with a letter, underscore, or a subset of Unicode code points greater than 00A0. Compared to C++ and similar languages, this convention is very standard. Since Julia follows a widely adopted standard for naming variables, it positively affects readability, write-ability and cost.

Something that Julia encourages, however, is using Unicode symbols in variable names. The Julia repl and other Julia editors support typing a backslash, then a LaTeX symbol name followed by tab to produce Unicode symbols. For example, `\alpha<tab>` produces the α character and `\alpha<tab>\hat<tab>\_2` creates the $\hat{\alpha}_2$. Although this feature might seem insignificant at first, I believe that encouraging developers to use Unicode symbols for variable names can improve readability and write-ability. Remember that Julia was developed for scientific computing. In the scientific community, Greek letters dominate variable names. Having the ability and encouraging developers to use the same names would make the transition from equations on paper to code much easier. Although using Unicode is not unique to Julia, because the community encourages using Unicode in variable names, Unicode characters end up being used more often in Julia code.

Numbers

Julia supports many primitive types. Specifically, signed and unsigned integers from 8 to 128 bits and signed and unsigned floating point numbers from 16 to 64 bits. It also has boolean, character, string. After these there are some composite types such as big integer and big float, which provide arbitrary precision, rational numbers and complex numbers. In taking on the task of visualizing the Julia set, having a built in complex number made the computation easier to write and much easier to read afterwards. `z = z^2 + c` where `z` and `c` are complex numbers is very obviously correct. This helps readability and write-ability a lot when dealing with these special types.

Julia also provides a function `eps(x)` which provides the distance between `x` and the next representable floating-point value for the given type. This functionality can be very valuable for comparing floating point numbers to determine if they are within an "epsilon" of each other. Similarly, `zero(x)` and `one(x)` functions are provided to return a zero or one of the specified `x` type to avoid the overhead of unnecessary type conversions. These two functions improve Julia's performance.

Beyond these few functions, Julia includes a host of mathematical functions including all the trigonometric functions, powers, logs, roots, sign, absolute value, greatest common denominator, and many more. A language is more than its syntax; it is the sum of all its standard libraries too. Julia was created for high performance computing, therefore having all these functions easily accessible makes Julia a tool fit for the job of scientific computing. This lowers the cost of writing in Julia while increasing its write-ability since a developer does not have to take time to define many of these basic functions himself.

Operators

Julia defines the usual binary operators: +, -, *, /, and % as well as their updating versions. Julia however, defines two more operators, the / (inverse divide) and ^ (power). Although I do not know a good use for /, I found it very useful to have a ^ operator for powers. This add to the readability and write-ability of Julia because the symbol ^ is already widely recognized as the power operator. Especially for the Julia set, which relies on squaring complex numbers, I found it convenient, and easier to write, given a power operator.

The comparison operators behave as they would be expected compared to C++ with one additional behavior I found to be favorable. Julia allows for the arbitrary chaining of comparison operators which are evaluated identically to mathematical comparison operators. For example if `x = 3` then `1 ≤ x < 5` evaluates as true. Julia's desire to translate mathematical equations closely to their paper representation shows here and I believe that this aids in readability and write-ability by removing the need to write `1 ≤ x && x < 5`.

Strings

The first things about strings in Julia that caught my attention was interpolation. I have not come across this feature in languages thus far and so I found it quite helpful. It is so easy to place a variable or expression in a string with out the need for string concatenation. For example, if `color` is a struct representing an rgb color, then printing the color to a ppm file is as easy as `write(fout, "$(color.r) $(color.g) $(color.b)")`. I found that interpolation increases write-ability and readability. Chains of string concatenation or multiple print/write statements can be verbose to write and confusing to read. Interpolation allows developers to quickly and easily insert variables into strings. I can, however, see where this functionality could be abused. Interpolation allows for expressions to be evaluated, for example `"1+1=$(1+1)"` would result in the string `"1+1=2"`. This could allow for complex expressions to be evaluated and interpolated creating, possibly, a more confusing statement overall. Despite this, I still believe interpolation is a positive feature.

Julia takes an interesting approach when it comes to string concatenation. As is explained in the Julia documentation, "In mathematics, + usually denotes a commutative operation, where the order of the operands does not matter ... In contrast, $\star$ typically denotes a noncommutative operation, where the order of the operands does matter ... As such, $\star$ is a more natural choice for an infix string concatenation operator, consistent with common mathematical use.

More precisely, the set of all finite-length strings S together with the string concatenation operator $\star$ forms a free monoid $(S, *)$. The identity element of this set is the empty string, "". Whenever a free monoid is not commutative, the operation is typically represented as $\cdot$, $\star$, or a similar symbol, rather than +, which as stated usually implies commutativity. [3]" I think that there is merit in the decision that Julia's authors made when determining the string concatation operator. I liked that they chose a noncommutative operator to use for string concatenation. Following from the $\star$ operator, the ^ operator is used for string repetition. Since Julia strives to be a high level language, I believe that infix operators are a positive for readability, write-ability, and cost.

Functions

in Julia, functions are first class citizens. Functions "can be assigned to variables and called using the standard function call syntax from the variable they have been assigned to. They can be used as arguments, and they can be returned as values. They can also be created anonymously, without being given a name. [4]" The last line of a function is automatically returned, however, the `return` keyword can be used to return from a function. I do not know the value of this functionality, I think that I hurts readability and the ability to reason about the code. I think it is a strength of languages that enforce explicit returning.

There was another feature of Julia, however, that I thought was a strength. Julia allows for automatic construction and deconstruction of tuples which give the feeling of returning multiple values from functions.

```
function foo(a,b)
    return a+b, a*b
end

x, y = foo(2,3)
```

This results in `x` having the value `5` and `y` the value `6`. This could remove the need for creating structures for the sole purpose of returning multiple values which would increase readability and write-ability

Julia also supports keyword arguments. This functionality, although not required, would be a great strength when dealing with functions that have many default parameters, or a complex interface. Keyword parameters increase readability greatly. When looking at code it is easy to know what each parameters of a function call means.

Packages

The final feature of Julia that I would like to mention is packages. Since its inception, Julia's community has grown immensely and contributed widely to Julia. All of the packages that have been written for Julia by its community are available through Julia's own package manager. I was very impressed with this aspect because it standardizes the way packages are maintained, distributed and used in Julia programs. I believe that having a package system that encourages and allows code to be shared and reused is very important. This greatly improves the cost of development when a package already exists that helps simplify or solve your problem. Write-ability is also takes advantage of this package system. Downloading and using packages is simple and can be accomplished in two lines of code. One drawback to packages, however, is that you have to trust the author. Packages are not held to as high of a standard as standard libraries so you must be careful.

CONCLUSION

The language Julia has many strength and some weaknesses. Julia is advertised as a high level, high performance language, and from what I can see of Julia's features, I believe that it returns on its promises. It is apparent that Julia's authors were inspired by many different languages. I felt as thought I was writing in a language that was the child of C++ and Python. It felt high level like Python but concurrently performant as C++. I believe the "syntactic sugar" of operators like the power operator (^) made writing and reasoning about code much more straightforward. Equations are representable in a form more similar to their paper counterparts which feel natural.

It is clear that Julia excels in the scientific computing field. Just a few months ago, in September of 2017, Julia entered the "Petaflop Club" when a project at UC Berkley, Celeste, "achieved peak performance of 1.54 petaflops using 1.3 million threads on 9,300 Knights Landing (KNL) nodes of the Cori supercomputer at NERSC... Celeste is written entirely in Julia, and the Celeste team loaded an aggregate of 178 terabytes of image data to produce the most accurate catalog of 188 million astronomical objects in just 14.6 minutes with state-of-the-art point and uncertainty estimates. [5]" Julia is clearly being adopted and used in large scale projects and performing well. The melding of high level and high performance makes reading and writing code in Julia much easier with out compromising performance. This lowers the cost of developing with Julia and this is what is making Julia a successful language.

REFERENCES

[1] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al., *Why We Created Julia*, 14-Feb-2012. [Online]. Available: https://julialang.org/blog/2012/02/why-we-created-julia. [Accessed: 30-Nov-2017].

[2] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al., *A Summary of Features, The Julia Language.* [Online]. Available: https://julialang.org/. [Accessed: 30-Nov-2017]

[3] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al., *"Strings" The Julia Language.* [Online]. Available: https://docs.julialang.org/en/stable/manual/strings/#String-Basics-1. [Accessed: 02-Dec-2017].

[4] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al., *"Functions" The Julia Language*. [Online]. Available: https://docs.julialang.org/en/stable/manual/functions/. [Accessed: 02-Dec-2017].

[5] Andrew Claster, *Julia Joins Petaflop Club,* 12-Sep-2017. [Online]. Available: https://juliacomputing.com/press/2017/09/12/julia-joins-petaflop-club.html. [Accessed: 03-Dec-2017].

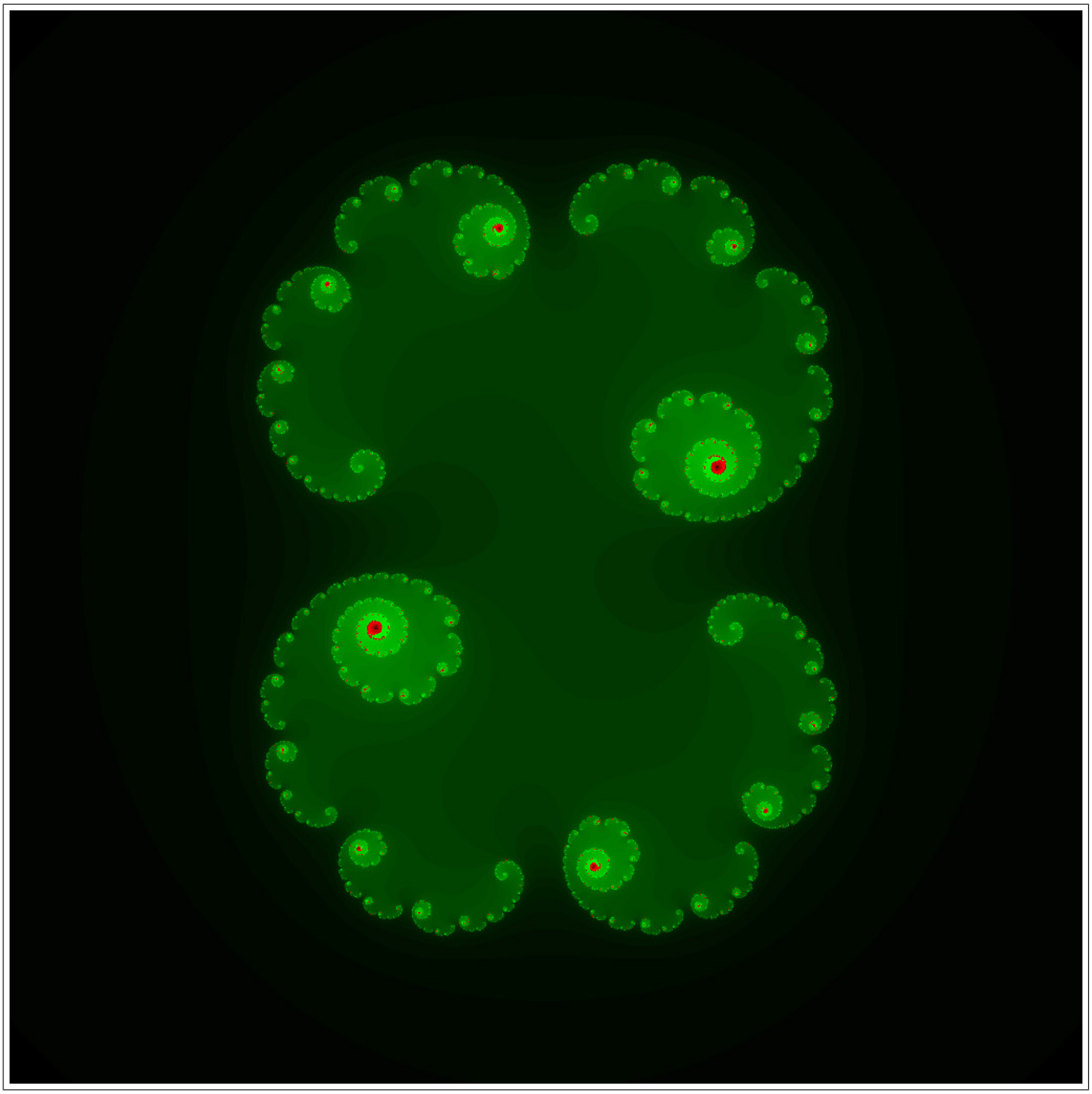

Fig. 1. The Julia Set $c = .3 - .01i$