AFRL-RI-RS-TR-2019-062

JULIA: A FRESH APPROACH TO TECHNICAL COMPUTING & DATA PROCESSING

MASSACHUSETTES INSTITUTE OF TECHNOLOGY (MIT)

MARCH 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-062 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /	**/ S /**
NANCY ROBERTS Work Unit Manager	TIMOTHY A. FARRELL Acting Chief, Information Intelligence Systems & Analysis Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2019	FINAL TECHNICAL REPORT	SEP 2015 – SEP 2018

4. TITLE AND SUBTITLE
JULIA: A FRESH APPROACH TO TECHNICAL COMPUTING & DATA PROCESSING

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-15-2-0272

5c. PROGRAM ELEMENT NUMBER
62702E

6. AUTHOR(S)
Alan Edelman

5d. PROJECT NUMBER
XDAT

5e. TASK NUMBER
A0

5f. WORK UNIT NUMBER
25

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology (MIT)
77 Massachusetts Ave
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/RIEA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-062

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report summarizes the activities enabled by an XDATA effort which includes a series of papers, the explosive growth of the Julia language, and a remarkable amount of software development. At the time of writing this report, there have been a near 4,000,000 downloads of the Julia Language, several textbooks authored by faculty worldwide based on the Julia language, and any number of classrooms using the Julia language. At the early start of the XDATA effort, Python was extremely popular, (as it remains today), but even large commercial companies such as Google are starting to understand Python's shortcomings. At the same time, libraries written in Julia remain callable from other popular languages such as Python. A remarkable amount of documentation that is directly or indirectly atributable to this work can be found on such pages as: 1. The Julia Lab web page at MIT : https://julia.mit.edu/; 2. The Julia Language web page: https://julialang.org/; and 3. the author's MIT web page: http://math.mit.edu/~edelman/

15. SUBJECT TERMS
abstract programming language, emerging architectures, abstract design tools, dynamic programming languages, design methods, parallel and distributed architectures, programming abstractions, numerical and data-science computing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**NANCY ROBERTS**
U	U	U	UU	19	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

1.0 Summary

Julia combines expertise from the diverse fields of computer science and computational science to create a new approach to numerical computing. The Julia language is designed to be easy and fast. Julia questions notions generally held as "laws of nature" by practitioners of numerical computing:

1. High-level dynamic programs have to be slow,

2. One must prototype in one language and then rewrite in another language for speed or deployment, and

3. There are parts of a system for the programmer, and other parts best left untouched as they are built by the experts.

We introduce the Julia programming language and its design — a dance between specialization and abstraction. Specialization allows for custom treatment. Multiple dispatch, a technique from computer science, picks the right algorithm for the right circumstance. Abstraction, what good computation is really about, recognizes what remains the same after differences are stripped away. Abstractions in mathematics are captured as code through another technique from computer science, generic programming. Julia shows that one can have machine performance without sacrificing human convenience.

2.0 Introduction

The original numerical computing language was Fortran, short for "Formula Translating System", released in 1957. Since those early days, scientists have dreamed of writing high-level, generic formulas and having them translated automatically into low-level, efficient machine code, tailored to the particular data types they need to apply the formulas to. Fortran made historic strides towards realization of this dream, and its dominance in so many areas of high-performance computing is a testament to its remarkable success.

The landscape of computing has changed dramatically over the years. Modern scientific computing environments such as Python [43], R [19], Mathematica [27], Octave [30], Matlab [28], and SciLab [16], to name some, have grown in popularity and fall under the general category known as dynamic languages or dynamically typed languages. In these programming languages, programmers write simple, high-level code without any mention of types like int, float or double that pervade statically typed languages such as C and Fortran.

Many researchers today do their day-to-day work in dynamic languages. Still, C and Fortran remain the gold standard for computationally-intensive problems for performance. In as much as the dynamic language programmer has missed out on performance, the C and Fortran programmer has missed out on productivity. An unfortunate outcome of the currently popular languages is that the most challenging areas of numerical computing have benefited the least from the increased abstraction and

productivity offered by higher-level languages. The consequences have been more serious than many realize.

Julia's innovation is the very combination of productivity and performance. New users want a quick explanation as to why Julia is fast, and whether somehow the same "magic dust" could also be sprinkled on their traditional scientific computing language. Julia is fast because of careful language design and the right combination of the carefully chosen technologies that work very well with each other. This report demonstrates some of these technologies using a number of examples.

Julia's ability to provide expressiveness for technical computing users comes in large part because of its multiple dispatch feature, which is an unconventional object-oriented programming technique which allows users to depart from the limitations of class-based designs and instead leverage the expressive power of generic functions. Loosely speaking, Julia's types are objects that are nouns, and generic functions are verbs. Julia's generic function system with multiple dispatch allows for users to describe different specific algorithms to carry out the same operation upon different objects. For example, there is a clear sense in which a function eigvals can be defined which implements the operation "take the eigenvalues of a matrix M" regardless of the specific properties of the matrix M, e.g. whether it is dense or sparse, local or distributed, symmetric or nonsymmetric. Nevertheless, it is often desirable to comeup with specialized algorithms to take full advantage of knowledge that can be exploited for the special case. For example, taking eigenvalues of a symmetric tridiagonal matrix can take advantage of the special structure of the matrix, avoiding the need to first reduce the matrix to tridiagonal form which would be needed in the general symmetric case, thus resulting in significant computational savings. In Julia, all that would be needed is to define a method for eigvals(M::SymTridiagonal) which implements one algorithm, while defining another method for eigvals(M::Symmetric) which provides a different algorithm taking into account symmetry but also being applicable to general symmetric matrices. By writing similar methods for yet other cases, Julia can expose a variety of algorithms, each of which can take full advantage of simplifications that result from leveraging special matrix properties in each special case. Users are also free to extend the eigvals function by providing yet more methods for their more specific use cases, e.g. for symplectic matrices over quaternions. Julia therefore also breaks down the wall between system library code and user code, dispelling the notion that there are only parts of a system which a programmer can use, with other parts best left untouched as they are built by the experts.

Julia began with a deep understanding of the needs of the scientific programmer, the data scientist, and the needs of the computer in mind. Bridging cultures that have all too often been distant, Julia combines expertise from computer science and computational science creating a new approach to scientific and data-based computing. The result is a programming language that allows users to easily traverse different levels of abstraction, and inspect the relationship between high level code they write and low level code being emitted by the compiler, and allow for rapid tweaking between the different layers.

3.0 Methods, Assumptions and Procedures

Many popular dynamic languages were not designed with the goal of high performance. After all, if you wanted really good performance you would use a static language, or so the popular wisdom would say. Only with the increased need in the day-to-day life of scientific programmers for simultaneous

productivity and performance in a single system has the need for high-performance dynamic languages become pressing. Unfortunately, retrofitting an existing slow dynamic language for high performance is almost impossible specifically in numerical computing ecosystems. This is because numerical computing requires performance-critical numerical libraries, which invariably depend on the details of the internal implementation of the high-level language, thereby locking in those internal implementation details. For example, you can run Python code much faster than the standard CPython implementation using the PyPy just-in-time compiler; but PyPy is currently incompatible with NumPy and the rest of SciPy.

Another important point is that just because a program is available in C or Fortran, it may not run efficiently from the high level language or be easy to "glue" it in. For example when Steven Johnson tried to include his C **erf** function in Python, he reported that Pauli Virtane had to write glue code(to vectorize the erf function over the native structures in Python in order to get good performance. Johnson also had to write similar glue code for Matlab, Octave, and Scilab. The Julia effort was, by contrast, effortless.[2] As another example, randn, Julia's normal random number generator was originally based on calling randmtzig, a C implementation. It turned out later, that a pure Julia implementation of the same code actually ran faster, and is now the default implementation. In some cases, "glue" can often lead to poor performance, even when the underlying libraries being called are high performance. The best path to a fast, high-level system for scientific and numerical computing is to make the system fast enough that all of its libraries can be written in the high-level language in the first place. The JUMP.jl [26] and the Convex.jl [42] packages are great examples of the success of this approach—the entire library is written in Julia and uses many Julia language features described in this paper.

The Two Language Problem: As long as the developers' language is harder than the users' language, numerical computing will always be hindered. This is an essential part of the design philosophy of Julia: all basic functionality must be possible to implement in Julia—never force the programmer to resort to using C or Fortran. Julia solves the two language problem. Basic functionality must be fast: integer arithmetic, for loops, recursion, floating-point operations, calling C functions, manipulating C-like structs. While these are not only important for numerical programs, without them, you certainly cannot write fast numerical code. "Vectorization languages" like Python+NumPy, R, and Matlab hide their for loops and integer operations, but they are still there, inside the C and Fortran, lurking behind the thin veneer. Julia removes this separation entirely, allowing high-level code to "just write a for loop" if that happens to be the best way to solve a problem.

We believe that the Julia programming language fulfills much of the Fortran dream: automatic translation of formulas into efficient executable code. It allows programmers to write clear, high-level, generic and abstract code that closely resembles mathematical formulas, as they have grown accustomed to in dynamic systems, yet produces fast, low-level machine code that has traditionally only been generated by static languages. Julia's ability to combine these levels of performance and productivity in a single language stems from the choice of a number of features that work well with each other:

1. An expressive type system, allowing optional type annotations;
2. Multiple dispatch using these types to select implementations;
3. Metaprogramming for code generation;
4. A dataflow type inference algorithm allowing types of most expressions to be inferred [3, 5];
5. Aggressive code specialization against run-time types [3, 5];

6. Just-In-Time (JIT) compilation [3, 5] using the LLVM compiler framework [23], which is also used by a number of other compilers such as Clang [12] and Apple's Swift [41]; and
7. Julia's carefully written libraries that leverage the language design, i.e., points 1 through 6 above.

4.0 Results and Discussion

4.1 A Taste of Julia

4.1.1 A Brief Tour

```
In[1]: A = rand(3,3) + eye(3) # Familiar Syntax
       inv(A)

Out[1]: 3x3 Array{Float64,2}:

        0.698106 -0.393074 -0.0480912
        -0.223584 0.819635 -0.124946
        -0.344861 0.134927 0.601952
```

The output from the Julia prompt says that A is a two dimensional matrix of size 3 × 3, and contains double precision floating point numbers.

Indexing of arrays is performed with brackets, and is 1-based. It is also possible to compute an entire array expression and then index into it, without assigning the expression to a variable:

```
In[2]: x = A[1,2]
       y = (A+2I)[3,3] # The [3,3] entry of A+2I
Out[2]: 2.601952
```

In Julia, I is a built-in representation of the identity matrix, without explicitly forming the identity matrix as is commonly done using commands such as "eye." ("eye", a homonym of "I", is used in such languages as Matlab, Octave, Go's matrix library, Python's Numpy, and Scilab.)

Julia has symmetric tridiagonal matrices as a special type. For example, we may define Gil Strang's favorite matrix (the second order difference matrix) in a way that uses only O(n) memory.

```
In[3]: strang(n) = SymTridiagonal(2*ones(n),-ones(n-1))
       strang(7)
Out[3]: 7x7 SymTridiagonal{Float64}:
2.0 -1.0 0.0 0.0 0.0 0.0 0.0
-1.0 2.0 -1.0 0.0 0.0 0.0 0.0
0.0 -1.0 2.0 -1.0 0.0 0.0 0.0
0.0 0.0 -1.0 2.0 -1.0 0.0 0.0
0.0 0.0 0.0 -1.0 2.0 -1.0 0.0
```

```
 0.0 0.0 0.0 0.0 -1.0 2.0 -1.0
0.0 0.0 0.0 0.0 0.0 -1.0 2.0
```

A commonly used notation to express the solution x to the equation Ax = b is A\b. If Julia knows that A is a tridiagonal matrix, it uses an efficient O(n) algorithm:

```
In[4]: strang(8)\ones(8)

Out[4]: 8-element Array{Float64,1}:

4.0
7.0
9.0
10.0
10.0
9.0
7.0
4.0
```

Note the Array{ElementType,dims} syntax. In the above example, the elements are 64 bit floats or Float64's. The 1 indicates it is a one dimensional vector. Consider the sorting of complex numbers. Sometimes it is handy to have a sort that generalizes the real sort. This can be done by sorting first by the real part, and where there are ties, sort by the imaginary part. Other times it is handy to use the polar representation, which sorts by radius then angle. By default, complex numbers are incomparable in Julia. If a numerical computing language "hard-wires" its sort to be one or the other, it misses an opportunity. A sorting algorithm need not depend on details of what is being compared or how it is being compared. One can abstract away these details thereby reusing a sorting algorithm for many different situations. One can specialize later. Thus alphabetizing strings, sorting real numbers, or sorting complex numbers in two or more ways all run with the same code. In Julia, one can turn a complex number w into an ordered pair of real numbers (a tuple of length 2) such as the Cartesian form (real(w),imag(w)) or the polar form (abs(w),angle(w)). Tuples are then compared lexicographically in Julia. The sort command takes an optional "less-than" operator, lt, which is used to compare elements when sorting. Note the compact function definition syntax available in Julia used in the example below and is of the form f(x,y,...) = .

```
In[5]: # Cartesian comparison sort of complex numbers
       complex compare1(w,z) = (real(w),imag(w)) < (real(z),imag(z))
       sort([-2,2,-1,im,1], lt = complex compare1 )
Out[5]: 5-element Array{Complex{Int64},1}:
       -2+0im
       -1+0im
        0+1im
       1+0im
       2+0im
```

```
In[6]: # Polar comparison sort of complex numbers
       complex compare2(w,z) = (abs(w),angle(w)) < (abs(z),angle(z))
       sort([-2,2,-1,im,1], lt = complex compare2)
Out[6]: 5-element Array{Complex{Int64},1}:
1+0im
0+1im
-1+0im
2+0im
-2+0im
```

To be sure, experienced computer scientists tend to suspect there is nothing new under the sun. The C function **qsort**() takes a **compa**r function. Nothing really new there. Python also has custom sorting with a key. Matlab's sort is more basic. The real contribution of Julia is that the design of Julia allows custom sorting to be high performance and flexible and comparable with implementations in other dynamic languages.

4.2 Writing programs with and without types

4.2.1 The balance between human and the computer

Graydon Hoare, author of the Rust programming language [35], in an essay on "Interactive Scientific Computing" [17] defined programming languages succinctly:

> Programming languages are mediating devices, interfaces that try to strike a balance between human needs and computer needs. Implicit in that is the assumption that human and computer needs are equally important, or need mediating.

A program consists of data and operations on data. Data is not just the input file, but everything that is held—an array, a list, a graph, a constant—during the life of the program. The more the computer knows about this data, the better it is at executing operations on that data. Types are exactly this metadata. Describing this metadata, the types, takes real effort for the human. Statically typed languages such as C and Fortran are at one extreme, where all types must be defined and are statically checked during the compilation phase. The result is excellent performance. Dynamically typed languages dispense with type definitions, which leads to greater productivity, but lower performance as the compiler and the runtime cannot benefit from the type information that is essential to produce fast code. Can we strike a balance between the human's preference to avoid types and the computer's need to know?

4.2.2 Julia's recognizable types

Many users of Julia may never need to know about types for performance. Julia's type inference system often does the work, giving performance without type declarations.

Julia's design allows for the gradual learning of concepts, where users start in a manner that is familiar to them and over time, learn to structure programs in the "Julian way" — a term that captures well-structured readable high performance Julia code. Julia users coming from other numerical computing environments have a notion that data may be represented as matrices that may be dense, sparse, symmetric, triangular, or of some other kind. They may also, though not always, know that elements in these data structures may be single precision floating point numbers, double precision, or integers of a specific width. In more general cases, the elements within data structures may be other data structures. We introduce Julia's type system using matrices and their number types:

```
In[14]: rand(1,2,1)
Out[14]: 1x2x1 Array{Float64,3}:
        [  :,  :, 1] =
        0.789166 0.652002
In[15]: [1 2; 3 4]
Out[15]: 2x2 Array{Int64,2}:
        1 2
        3 4
In[16]: [true; false]
Out[16]: 2-element Array{Bool,1}:
    true
    false
```

We see a pattern in the examples above. Array{T,ndims} is the general form of the type of a dense array with ndims dimensions, whose elements themselves have a specific type T, which is of type double precision floating point in the first example, a 64-bit signed integer in the second, and a boolean in the third example. Therefore Array{T,1} is a 1-d vector (first class objects in Julia) with element type T and Array{T,2} is the type for 2-d matrices.

It is useful to think of arrays as a generic N-d object that may contain elements of any type T. Thus T is a type parameter for an array that can take on many different values. Similarly, the dimensionality of the array ndims is also a parameter for the array type. This generality makes it possible to create arrays of arrays. For example, Using Julia's array comprehension syntax, we create a 2-element vector containing 2 × 2 identity matrices. In[17]: a = [eye(2) for i=1:2] Out[17]: 2-element Array{Array{Float64,2},1}:

4.2.3 User's own types are first class too

Many dynamic languages for numerical computing have traditionally had an asymmetry, where built-in types have much higher performance than any user-defined types. This is not the case with Julia, where there is no meaningful distinction between user-defined and "built-in" types.

We have mentioned so far a few number types and two matrix types, Array{T,2} the dense array, with element type T and SymTridiagonal{T}, the symmetric tridiagonal with element type T. There are also other matrix types, for other structures including SparseMatrixCSC (Compressed Sparse Columns), Hermitian, Triangular, Bidiagonal, and Diagonal. Julia's sparse matrix type has an added flexibility that it can go beyond storing just numbers as nonzeros, and instead store any other Julia type as well. The indices in SparseMatrixCSC can also be represented as integers of any width (16-bit, 32-bit or 64-bit). All

these different matrix types, although available as built-in types to a user downloading Julia, are implemented completely in Julia, and are in no way any more or less special than any other types one may define in their own program.

For demonstration, we create a symmetric arrow matrix type that contains a diagonal and the first row A[1,2:n].

```
In[18]: # Type Parameter Example (Parameter T)
        # Define a Symmetric Arrow Matrix Type with elements of type T

    type SymArrow{T}
    dv::Vector{T} # diagonal
    ev::Vector{T} # 1st row[2:n]
    end

    # Create your first Symmetric Arrow Matrix
    S = SymArrow([1,2,3,4,5],[6,7,8,9])

Out[18]: SymArrow{Int64}([1,2,3,4,5],[6,7,8,9])
```

The parameter in the array refers to the type of each element of the array. Code can and should be written independently of the type of each element.

Julia's type system allows for abstract types, concrete "bits" types, composite types, and immutable composite types. All of these can have parameters and users may even write programs using unions of these different types.

4.3 Multiple Dispatch

Multiple dispatch is the selection of a function implementation based on the types of each argument of the function. It is not only a nice notation to remove a long list of "case" statements, but it is part of the reason for Julia's speed. It is expressed in Julia by annotating the type of a function argument in a function definition with the following syntax: argument::Type.

Mathematical notations that are often used in print are difficult to employ in programs. For example, we can teach the computer some natural ways to multiply numbers and functions. Suppose that a and t are scalars, and f and g are functions, and we wish to define 1. Number x Function = scale output: a ∗ g is the function that takes x to a ∗ g(x) 2. Function x Number = scale argument : f ∗ t is the function that takes x to f(tx) and 3. Function x Function = composition of functions: f ∗ g is the function that takes x to f(g(x)). If you are a mathematician who does not program, you would not see the fuss. If you thought how you might implement this in your favorite computer language, you might immediately see the benefit. In Julia, multiple dispatch makes all three uses of * easy to express: In[21]: *(a::Number, g::Function)= x->a*g(x) # Scale output *(f::Function,t::Number) = x->f(t*x) # Scale argument *(f::Function,g::Function)= x->f(g(x)) # Function composition Here, multiplication is dispatched by the

type of its first and second arguments. It goes the usual way if both are numbers, but there are three new ways if one, the other, or both are functions. These definitions exist as part of a larger system of generic definitions, which can be reused by later definitions. Consider the case of the mathematician Gauss' preference for sin2 ϕ to refer to sin(sin(ϕ)) and not sin(ϕ) 2 (writing "sin2 (ϕ) is odious to me, even though Laplace made use of it."(Figure 2).) By defining *(f::Function, g::Function)= x->f(g(x)), (f^2)(x) automatically computes f(f(x)) as Gauss wanted. This is a consequence of a generic definition that evaluates x^2 as x*x no matter how x*x is defined. This paradigm is a natural fit for numerical computing, since so many important operations involve interactions among multiple values or entities. Binary arithmetic operators are obvious examples, but many other uses abound. The fact that the compiler can pick the sharpest matching definition of a function based on its input types helps achieve higher performance, by keeping the code execution paths tight and minimal. We have not seen this in the literature but it seems worthwhile to point out four possibilities:

1. Static single dispatch (not done)
2. Static multiple dispatch (frequent in static languages, e.g. C++ overloading)
3. Dynamic single dispatch (Matlab's object oriented system might fall in this category though it has its own special characteristics)
4. Dynamic multiple dispatch (usually just called multiple dispatch).

4.4 High Performance Polynomials and Special Functions with Macros

Julia has a macro system that provides easy custom code generation, bringing a level of performance that is otherwise difficult to achieve. A macro is a function that runs at parse-time, and takes parsed symbolic expressions in and returns transformed symbolic expressions out, which are inserted into the code for later compilation.

For example, a library developer implemented an @evalpoly macro that uses Horner's rule to evaluate polynomials efficiently. Consider

```
In[47]: @evalpoly(10,3,4,5,6)
```

which returns 6543 (the polynomial $3 + 4x + 5x^2 + 6x^3$, evaluated at 10 with Horner's rule). Julia allows us to see the inline generated code with the command

```
In[48]: macroexpand(:@evalpoly(10,3,4,5,6))
```

We reproduce the key lines below

```
Out[48]: #471#t = 10 # Store 10 into a variable named #471#t
Base.Math.+(3,Base.Math.*(#471#t,Base.Math.+(4,Base.Math.*
(#471#t,Base.Math.+(5,Base.Math.*(#471#t,6)))) ))
```

This code-generating macro only needs to produce the correct symbolic structure, and Julia's compiler handles the remaining details of fast native code generation. Since polynomial evaluation is so important for numerical library software, it is critical that users can evaluate polynomials as fast as possible. The overhead of implementing an explicit loop, accessing coefficients in an array, and possibly a subroutine call (if it is not inlined), is substantial compared to just inlining the whole polynomial evaluation.

Steven Johnson reports in his EuroSciPy notebook19

This is precisely how erfinv is implemented in Julia (in single and double precision), and is 3 to 4 times faster than the compiled (Fortran?) code in Matlab, and 2 to 3 times faster than the compiled (Fortran Cephes) code used in SciPy.

The difference (at least in Cephes) seems to be mainly that they have explicit arrays of polynomial coefficients and call a subroutine for Horner's rule, versus inlining it via a macro.

Johnson also used the same trick in his implementation of the digamma special function for complex arguments20 following an idea of Knuth:

> As described in Knuth TAOCP vol. 2, sec. 4.6.4, there is actually an algorithm even better than Horner's rule for evaluating polynomials p(z) at complex arguments (but with real coefficients): you can save almost a factor of two for high degrees. It is so complicated that it is basically only usable via code generation, so it would be especially nice to modify the @horner macro to switch to this for complex arguments.

No sooner than this was proposed, the macro was rewritten to allow for this case giving a factor of four improvement in performance on all real polynomials evaluated at complex arguments.

5.0 Conclusions

This document summarizes the activities enabled by the DARPA XDATA cooperative agreement that includes a series of papers, the explosive growth of the Julia language, and a remarkable amount of software development. At the time of writing this report, there have been a near 4,000,000 downloads of the Julia Language, several textbooks authored by faculty worldwide based on the Julia language, and any number of classrooms using the Julia language. At the early start of the XDATA effort, Python was extremely popular, (as it remains today), but even large commercial companies such as Google are starting to understand Python's shortcomings. At the same time, libraries written in Julia remain callable from other popular languages such as Python. More than just a language, Julia has become a place for programmers, physical scientists, social scientists, computational scientists, mathematicians, and others to pool their collective knowledge in the form of online discussions and in the form of code. A remarkable amount of documentation that is directly or indirectly attributable to this work can be found on the following pages:

1. The Julia Lab web page at MIT: https://julia.mit.edu/

2. The Julia Language web page: https://julialang.org/

3. And the author's MIT web page: http://math.mit.edu/~edelman/

6.0 References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification version 1.0. http://research.sun.com/projects/plrg/fortress.pdf, 2008.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] Jeff Bezanson. Abstraction in Technical Computing. PhD thesis, Massachusetts Institute of Technology, 2015.

[4] Jeff Bezanson, Jiahao Chen, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Array operators using multiple dispatch. ARRAY'14, 2014.

[5] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: a Fast Dynamic Language for Technical Computing. arXiv:1209.5145v1, 2012.

[6] B.L. Chamberlain. A Brief Overview of Chapel. http://chapel.cray.com/papers/ ChapelCUG13.pdf, 2013.

[7] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl., 21(3):291–312, August 2007.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. SIGPLAN Not., 40(10):519–538, October 2005.

[9] Ron Choy and Alan Edelman. Parallel MATLAB: Doing it right. In Proceedings of the IEEE, volume 93, pages 331–341, 2005.

[10] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, and David Cheng. Star-P: High productivity parallel computing. In In 8th Annual Workshop on High-Performance Embedded Computing (HPEC 04), 2004.

[11] Barry A. Cipra. The best of the 20th century: Editors name top 10 algorithms. SIAM News. https://www.siam.org/pdf/news/637.pdf.

[12] The Clang project. http://clang.llvm.org/.

[13] James W. Demmel, Jack J. Dongarra, Beresford N. Parlett, William Kahan, Ming Gu, David S. Bindel, Yozo Hida, Xiaoye S. Li, Osni A. Marques, E. Jason Riedy, Christof Vomel, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, Julie Langou, and Stanimire Tomov. Prospectus for the next LAPACK and ScaLAPACK libraries. Technical Report 181, LAPACK Working Note, February 2007.

[14] Alan Edelman, Parry Husbands, and Steve Leibman. Interactive supercomputing's star-p platform: Parallel matlab and mpi homework classroom study on high level language productivity. In Proceedings of the 10th High Performance Embedded Computing Workshop (HPEC 2006), 2006.

[15] Alan Edelman and Brian Sutton. From Random Matrices to Stochastic Operators. Journal of Statistical Physics, 127:1121–1165, 2007.

[16] Claude Gomez, editor. Engineering and Scientific Computing With Scilab. Birkh¨auser, 1999.

[17] Graydon Hoare. technicalities: interactive scientific computing #1 of 2, pythonic parts. http: //graydon2.dreamwidth.org/3186.html, 2014.

[18] Parry Husbands, Charles L. Isbell, Jr., and Alan Edelman. Interactive supercomputing with mitmatlab, 1998.

[19] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics, 5:299–314, 1996.

[20] Andreas Noack Jensen. Fast and generic linear algebra in Julia. Technical report, MIT, 2015.

[21] William Kahan. How futile are mindless assessments of roundoff in floating-point computation? http://www.cs.berkeley.edu/~wkahan/Mindless.pdf, 2006.

[22] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. Journal of the ACM, 27(1):128–145, January 1980.

[23] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), pages 75–86, Palo Alto, California, Mar 2004.

[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Softw., 5(3):308–323, September 1979.

[25] MIT License. http://opensource.org/licenses/MIT. [26] Miles Lubin and Iain Dunning. Computing in Operations Research using Julia. INFORMS Journal on Computing, 27(2):238–248, 2015.

[27] Mathematica. http://www.mathematica.com.

[28] Matlab. http://www.mathworks.com.

[29] Markus Mohnen. A Graph-Free Approach to Data-flow Analysis. In R. Horspool, editor, Compiler Construction, volume 2304 of Lecture Notes in Computer Science, pages 185–213. Springer Berlin / Heidelberg, 2002.

[30] Malcolm Murphy. Octave: A Free, High-Level Language for Mathematics. Linux J., 1997, July 1997.

[31] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA '08, pages 563–582, New York, NY, USA, 2008. ACM.

[32] The Jupyter Project. http://jupyter.org/.

[33] The MPFR Project. http://www.mpfr.org/.

[34] The X10 project. http://x10-lang.org/.

[35] Rust. http://www.rust-lang.org/.

[36] Helen Shen. Interactive notebooks: Sharing the code, Nature Toolbox, Volume 515, Issue 7525, Nov 2014. http://www.nature.com/news/interactive-notebooks-sharing-the-code-1. 16261.

[37] Guy Steele Jr. Parallel programming and code selection in fortress. In ”PPoPP ’06 Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming”, page 1, 2006.
[38] Gilbert Strang. Introduction to Linear Algebra. Wellesley-Cambridge Press, 2003.

[39] Interactive Supercomputing. Star-p user guide. http://www-math.mit.edu/~edelman/ publications/star-p-user.pdf.

[40] Interactive Supercomputing. Getting started with star-p; taking your first test-drive. http: //www-math.mit.edu/~edelman/publications.php, 2006.

[41] Swift. https://developer.apple.com/swift/.

[42] Madeleine Udell, Karanveer Mohan, David Zeng, Jenny Hong, Steven Diamond, and Stephen Boyd. Convex optimization in Julia. SC14 Workshop on High Performance Technical Computing in Dynamic Languages, 2014.

[43] St´efan van der Walt, S. Chris Colbert, and Ga¨el Varoquaux. The numpy array: a structure for efficient numerical computation. CoRR, abs/1102.1523, 2011.

[44] Hadley Wickham. ggplot2. http://ggplot2.org/.

[45] Leland Wilkinson. The Grammar of Graphics (Statistics and Computing). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

Appendix A - Publications

[1] Alexander Amini, Berthold Horn, and Alan Edelman. Accelerated convolutions for efficient multi-scale time to contact computation in julia. arXiv preprint arXiv:1612.08825, 2016.

[2] Jeff Bezanson, Jake Bolewski, and Jiahao Chen. Fast flexible function dispatch in julia. arXiv preprint arXiv:1808.03370, 2018.

[3] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. Julia: Dynamism and performance reconciled by design. Proc. ACM Program. Lang., 2(OOPSLA):120:1–120:23, October 2018. (http://doi.acm.org/10.1145/3276490).

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. SIAM review, 59(1):65–98, 2017. (https://arxiv.org/abs/1411.1607).

[5] Jeffrey Werner Bezanson. Abstraction in technical computing. PhD thesis, Massachusetts Institute of Technology, 2015. (https://github.com/JeffBezanson/phdthesis/blob/master/main.pdf).

[6] Cy Chan, Vesselin Drensky, Alan Edelman, Raymond Kan, and Plamen Koev. On computing schur functions and series thereof. Journal of Algebraic Combinatorics, Oct 2018. (https://doi.org/10.1007/s10801-018-0846-y).

[7] Alexander Chen, Alan Edelman, Jeremy Kepner, Vijay Gadepally, and Dylan Hutchison. Julia implementation of the dynamic distributed dimensional data model. In High Performance Extreme Computing Conference (HPEC), 2016 IEEE, pages 1–7. IEEE, 2016. (https://arxiv.org/abs/1608.04041).

[8] Jiahao Chen. Linguistic relativity and programming languages. arXiv preprint arXiv:1808.03916, 2018.

[9] Jiahao Chen and Alan Edelman. Parallel prefix polymorphism permits parallelization, presentation & proof. In Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages, pages 47–56. IEEE Press, 2014.

[10] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. arXiv preprint arXiv:1608.04295, 2016.

[11] Alexander Dubbs. Beta-ensembles with covariance. PhD thesis, Massachusetts Institute of Technology, 2014.

[12] Alexander Dubbs and Alan Edelman. Infinite random matrix theory, tridiagonal bordered toeplitz matrices, and the moment problem. Linear Algebra and its Applications, 467:188–201, 2015.

[13] Alan Edelman, Alice Guionnet, S Péché, et al. Beyond universality in random matrix theory. The Annals of Applied Probability, 26(3):1659–1697, 2016.

[14] Alan Edelman and Michael La Croix. The singular values of the gue (less is more). Random Matrices: Theory and Applications, 4(04):1550021, 2015.

[15] Alan Edelman and Yuyang Wang. Random hyperplanes, generalized singular values & "what's my β?". In 2018 IEEE Statistical Signal Processing Workshop, SSP 2018, Freiburg im Breisgau, Germany, June 10-13, 2018, pages 458–462, 2018. (https://doi.org/10.1109/SSP.2018.8450833).

[16] Mike Innes, Stefan Karpinski, Viral Shah, David Barber, PLEPS Saito Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, SimonDanisch, Alan Edelman, et al. On machine learning and programming languages. In SysML 15 Association for Computing Machinery (ACM), 2018. (http://discovery.ucl.ac.uk/10051391/1/37.pdf).

[17] Jeremy Kepner, Ron Brightwell, Alan Edelman, Vijay Gadepally, Hayden Jananthan, Michael Jones, Sam Madden, Peter Michaleas, Hamed Okhravi, Kevin Pedretti, et al. Tabularosa: Tabular operating system architecture for massively parallel heterogeneous compute engines. In 2018 IEEE High Performance extreme Computing Conference (HPEC), pages 1–8. IEEE, 2018.

[18] Oren Mangoubi and Alan Edelman. Integral geometry for markov chain monte carlo: overcoming the curse of search-subspace dimensionality. arXiv preprint arXiv:1503.03626, 2015.

[19] Oren Oren Rami Mangoubi. Integral geometry, Hamiltonian dynamics, and Markov Chain Monte Carlo. PhD thesis, Massachusetts Institute of Technology, 2016.

[20] BH McRae, VB Shah, and Alan Edelman. Circuitscape: modeling landscape connectivity to promote conservation and human health. On web, 2016. (https://www.researchgate.net/profile/Brad_Mcrae2/publication/304835052_Circuitscape_modeling_landscape_connectivity_to_promote_conservation_and_human_health/links/577c318408ae213761caba80.pdf).

[21] Ramis Movassagh and Alan Edelman. Condition numbers of indefinite rank 2 ghost wishart matrices. Linear Algebra and its Applications, 483:342–351, 2015.

[22] Ramis Movassagh and Alan Edelman. Eigenvalue approximation of sums of hermitian matrices from eigenvector localization/delocalization. arXiv preprint arXiv:1710.09400, 2017.

[23] Jeffrey Regier, Kiran Pamnany, Keno Fischer, Andreas Noack, Maximilian Lam, Jarrett Revels, Steve Howard, Ryan Giordano, David Schlegel, Jon McAuliffe, et al. Cataloging the visible universe through bayesian inference at petascale. arXiv preprint arXiv:1801.10277, 2018.

[24] Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. Dynamic automatic differentiation of gpu broadcast kernels. arXiv preprint arXiv:1810.08297, 2018.

[25] Cooper Stokes Sloan. Neural bus networks. PhD thesis, Massachusetts Institute of Technology, 2018.