Julia Programming Language Benchmark Using a Flight Simulation

Ray Sells
DESE Research, Inc./Jacobs Space Exploration Group
Guidance, Navigation, and Mission Analysis
NASA-MSFC EV 42
harold.r.sells@nasa.gov

***Abstract*—Julia's goal to provide scripting language ease-of-coding with compiled language speed is explored. The runtime speed of the relatively new Julia programming language is assessed against other commonly used languages including Python, Java, and C++. An industry-standard missile and rocket simulation, coded in multiple languages, was used as a test bench for runtime speed. All language versions of the simulation, including Julia, were coded to a highly-developed object-oriented simulation architecture tailored specifically for time-domain flight simulation. A "speed-of-coding" second-dimension is plotted against runtime for each language to portray a space that characterizes Julia's scripting language efficiencies in the context of the other languages. With caveats, Julia runtime speed was found to be in the class of compiled or semi-compiled languages. However, some factors that affect runtime speed at the cost of ease-of-coding are shown. Julia's built-in functionality for multi-core processing is briefly examined as a means for obtaining even faster runtime speed. The major contribution of this research to the extensive language benchmarking body-of-work is comparing Julia to other mainstream languages using a complex flight simulation as opposed to benchmarking with single algorithms.**

TABLE OF CONTENTS

1. INTRODUCTION

Julia [1] is a relatively new computer language that aims to reduce the challenge for math-modelers to develop fast computer tools and simulations. It potentially combines the ease-of-coding feature of scripting languages (like Python) with the performance of compiled languages (like C++). Normally, these two features are mutually exclusive with existing, conventional programming languages. Julia's ease-of-coding principally derives from its dynamic typing feature, where data are inferred "on-the-fly" without the necessity (code and complexity) for static data declaration. The dynamic feature is most commonly associated with scripting languages. Syntactically, Julia somewhat resembles Python, but is not meant to be a "compiled Python." A key question for Julia application to the simulation domain is, "Can Julia, with its obvious coding simplicity, attain runtime speeds comparable to conventional compiled languages for flight simulation?"

A wide body of literature exists for Julia benchmarking. Benchmarking is not as objective as many would accept it to be. For instance, benchmarks between different languages can be masked or misleading due to inconsistent code architecture or different styles of coding. Also, it is a coder's prerogative on how to utilize a language's features for a particular problem. The literature contains many excellent language benchmarking sources that recognize the subjectivity of benchmarking and attempt to address it in a fair manner. One of the original and most prominent benchmarks was performed by Bagley [2]. A wide variety of languages (but not Julia) were benchmarked across several code tests. The code tests were comprised of algorithms spanning a wide domain of operation from sorting, string manipulation, floating point computations, and hashes, among many others. Notably, the source code is included for all tests (in all languages). Following in this pattern, *The Computer Language Benchmarks Game* [3] is a more recent and similar presentation that includes Julia. Notably, multiple code submissions for a benchmark in the same language from different contributors are catalogued in a fair attempt to capture code with the fastest speed. Kouatchou (the "NASA Modeling Guru") [4] maintains a comprehensive set of benchmark results where the algorithm tests tend more exclusively toward a scientific interest. This is reflected in the languages tested including Python, Julia, Matlab, and R. *R Beats Python!* [5] is an example of how benchmarks can be interpreted differently. Some "unfair timing comparisons" by "the Julia group" are contested because the "R code was not vectorized." A takeaway from this body-of-work review, fairness not withstanding, is that most of the benchmarks were characterized by single algorithms being executed in tight loops. While these benchmarks are certainly informative for flight simulation coding, extrapolation of their results to large scale flight simulation execution speed is not straightforward.

Beyond basic algorithmic benchmarks, the literature contains several favorable Julia benchmarks for problem domains easily vectorized, like "field" problems for numerical solution of partial differential equations. These type of computations are typically found in the high speed computing domain. Eadline [6] makes the case for Julia as an ideal solution for this type of computing. Gutierrez [7] emphasizes Julia's built-in primitives for parallel computing including vectorization. However, flight simulations typically involve the execution of sequentially-dependent calculations inside a state-propagation loop. This type of computation does not lend itself well to vectorization.

Consequently, the literature review for Julia benchmarking revealed a gap in addressing Julia for time-domain, lumped-element dynamic computations typical for flight simulation. This paper begins describing how a unique combination of existing elements have been employed to address this relatively sparse area of the Julia benchmark research. An extensively documented object-oriented simulation architecture, its implementation in an industry standard rocket flight simulation, and separate versions (C++, Java, and Python) provide the setting for a comparative evaluation of Julia runtime speed (and ease of coding). This combination of existing elements avoids a common pitfall of benchmarks: Benchmarks between different languages can be masked or misleading due to inconsistent code architecture or different styles of coding. Thus, a common architecture and flight application provides for an accurate and fair comparison.

Execution speeds for a Julia-version of the rocket simulation are presented for a variety of runtime scenarios including single-run and various increments of "batch" runs comprised of several executions. These are compared against their C++, Java, and Python simulation counterparts. Recognizing runtime speed must be examined in the larger context of ease-of-coding, metrics for the relationship between these two criteria are shown for each of the language implementations. A comparison of lines-of-code is included, for instance. Also, key Julia benchmark caveats are noted that reflect a usability and runtime speed tradeoff.

The paper concludes with a brief experiment addressing Julia runtime speed in a multi-core, parallel computing context. Although not in the original scope of this research, Julia makes parallel computation very accessible to simulation practitioners without requiring a high degree of computer expertise. In keeping with the objective to achieve high runtime speed complemented with easy coding, Julia parallel computation was briefly explored with independent rocket simulations executing on multiple cores.

2. BENCHMARK DESCRIPTION

Aerospace flight simulation is a large domain within the digital simulation world and a large potential audience for Julia application. Within this domain, it was recognized that a simulation for experimentation was available. The Mini-Rocket simulation [8] is coded in several language versions which are in active use. Mini-Rocket is open-source [9].

Mini-Rocket (MR) is a very easy-to-configure, multiple degree-of-freedom missile and rocket fly-out model that accurately generates trajectories in three-dimensional space, including maneuver characteristics. It features a unique algorithm that accurately models missile dynamics at a fraction of the computational cost of conventional six degree-of-freedom simulations while maintaining a significant amount of the fidelity. The program is ideal for those analyses requiring trajectory modeling without the necessity of detailed modeling of the onboard missile subsystems. Some of its more detailed features are summarized in Table 1.

Table 1. Mini-Rocket Features

Feature
• Osculating-plane formulation [10] reduces compute-time overhead of full six degree-of-freedom equations-of-motion • Motion in three-dimensional space • Two independent channels (pitch and yaw) for steering and guidance • 1, 2, and 3-dimensional table lookups to model aerodynamics and propulsion characteristics • Capability to model angle-of-attack variations in lift and drag • Constraints on lateral acceleration based on angle-of-attack and closed-loop airframe response time • Detailed models for control and guidance subsystems not required

Key to this benchmark, all language versions of MR have been coded to the same object-oriented architecture. An *object-oriented simulation kernel* (OSK) [11] successively executes sequences of model objects inside a differential equation (DE) engine. A long, traceable heritage of comparisons exists to establish the accuracy of the MR model and coding mechanizations [8, Section 5] in its different language instantiations.

3. PRIOR FINDINGS

The C++, Java, and Python MR simulations have been previously benchmarked [12]. It is informative to review those findings here before proceeding. The OSK architecture was used to build simulation engines in C++, Java, and Python owing to unique user and stakeholder requirements that had a language preference. Also, these languages span a range of syntax simplicity versus runtime speed and they are all object-oriented. Key features of each language are shown in Table 2. This was a good opportunity to explore different facets of the languages as their functionality was expressed in different ways to build MR.

Table 2. Key Characteristics of Benchmark Languages

Language	Characteristic
C++	• Fast Execution • Most syntactically complex • Object-Oriented
Java	• Simpler syntax • Compiles to slower byte-code, but still fast • Object-oriented
Python	• Most syntactically concise, easiest to code • Interpreted – no need for compilation but slower execution • Dynamic-typing reduces lines of code • Object-oriented

Figure 1 illustrates the trade between these languages for MR. Identical MR models, in the sense of having the same math models and coding architecture, were executed to generate a representative trajectory. Runtime speed results were collected. A metric for the other trade parameter, ease-of-coding, was obtained by subjectively estimating how quickly the simulation could be coded (after becoming proficient in each language). For instance, the Java simulation could be coded twice as quickly as the C++ version and the Python version four times as quickly.

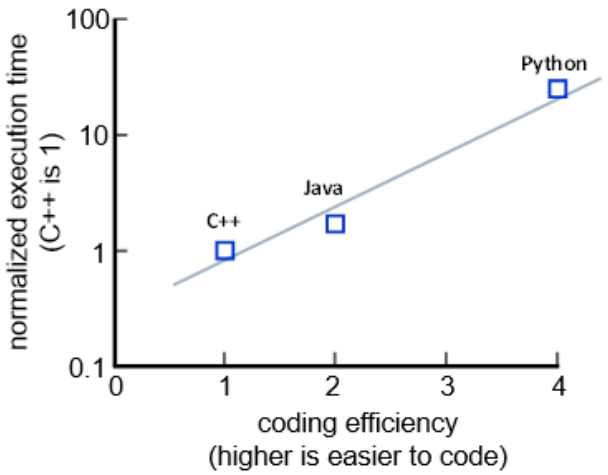

Figure 1. Benchmark Runtime and Coding Efficiency Comparison for Different Computer Languages

This prior work provides a unique opportunity for Julia runtime speed evaluation that leverages man-years of previous work. Additional Julia benchmarks can be performed in a "real" simulation application setting with relatively little additional effort.

4. JULIA EXPERIMENTATION

Proficiency in Julia was a mandatory step for a fair benchmark comparison. Of course, Julia's home page is a good starting point [1]. Besides the official documentation, the author found several useful learning and reference aids. The "wikibook" *Introducing Julia* [13] has a convenient hyper-linked table of contents. Although the online book *Julia Tutorial* [14] is intended as an introductory tutorial for university students, it succinctly shows advanced aspects of the language. Finally, the author found the Julia "cheat sheet" [15] ever useful. These were the first steps to become familiar with the language and to proficiently utilize its features and idioms.

Once coding begins, two distinctly different approaches can be taken to build a Julia MR simulation: translation versus refactoring. The translation approach is it sounds - an attempt at a line-to-line conversion. Baker [16] is an early example of this approach directed at Fortran programmers learning C. C programming macros are used to replicate Fortran indexing (starting with 1 instead of 0) and looping in C in an attempt to preserve Fortran programming idioms in C. The more well-known Bell Laboratories f2c [17] mechanizes the translation. The translation approach was NOT adopted to create the Julia MR code for benchmarking. This build process would, in essence, handicap the Julia code since the Julia language features and programming constructs would not be fully used (if at all). Thus effort to become proficient in Julia programming was critical to the integrity of a fair benchmark.

A refactoring approach was implemented to build a Julia MR from "the-ground-up." Julia is not strictly object-oriented but provides mechanisms for an object-oriented approach [1, Composite Types]. For instance, Julia functions are not bundled with the data they operate on - a common object-oriented characteristic. But, on the other hand, multiple dispatch capability is provided so that functions with the same name can be chosen based on the number-of and types of the calling arguments. Julia also provides extensive support for composite types (*records* and *structs* in other languages). Both these attributes are commonly associated with object-oriented languages. Coding focused on utilization of the organic Julia constructs to build a "native" Julia simulation as opposed to a translation.

Before building the Julia MR itself, three major infrastructure parts were built and tested independently, partly for familiarization, but mostly to build a solid simulation infrastructure foundation with respect to speed and functionality. The three parts were the DE engine, table look-up utilities, and vector/matrix manipulations. These parts are integral to all flight simulations. Some code excerpts are shown to highlight the ability to construct very concise, readable Julia code for these functional pieces.

The DE engine is the critical piece since it is the simulation executive. The DE engine is the code that orchestrates

execution of the integrating time loop, successively processing and propagating state vectors defined by the flight model objects. The same DE engine structure used in the C++, Java, and Python MR simulations was used as the pattern for the Julia code. A code excerpt illustrating DE engine usage is shown in Figure 2. Notably, the 4th order Runge-Kutta integrator functions and states are encapsulated as integrator objects.

```
#### SIMULATION
mr = build_rocket( "rocket.dat", "**** rocket ****")
integrators = [ mr.myclock, mr.mass, mr.vmx, mr.vmy, mr.vmz,
                mr.pmx, mr.pmy, mr.pmz]
for ii in 1:100 # MC LOOP
  clock = Clock1( 0.0, 0.01)
  init( mr)
  println( "Launch!!!")
  while clock.t <= 100.0 + 1e-6
    #### UPDATE
    update( mr, clock)
    #### REPORT
    report( mr, clock)
    #### Propagate
    [ propagate( state, clock) for state in integrators]
    update( clock)
  end
end
```

integrator objects

use integrators to propagate states

Figure 2. Julia MR Differential Equation Engine Representation in Main Program

Interacting with the DE engine, all simulation entities are a hierarchy of objects including the clock, integrators, tables, rocket stages, and the full-up rocket itself. Figure 3 shows a portion of the simulation object composition, starting with creation of an atmosphere object. Three stage objects are established, where their underlying physical characteristics are read from data files at object creation. The stage objects are then collected into a rocket object where they are executed sequentially by the DE engine.

```
# CREATE ATMOSPHERE OBJECT
mr.atm = Atm62()

# CREATE STAGE OBJECTS
stage1 = stage_read( fname, "**** stage 1 ****")
stage2 = stage_read( fname, "**** stage 2 ****")
stage3 = stage_read( fname, "**** stage 3 ****")
mr.stages = [ stage1, stage2, stage3]
```

Figure 3. MR Simulation Object Composition

Table look-ups and associated utilities were the second simulation infrastructure piece studied for development. Tables of the physical data are called throughout the simulation and are a major influence on runtime efficiency. An object-oriented paradigm was used to create a very usable programming interface for the simulation coder as shown by the code excerpt in Figure 4. Note multiple dispatch in interpolating the thrust, *txv*, and axial drag coefficient, *ca_off*. The same *interp* function is used, but the call is dispatched to the correct function based on the table dimension. Features like this ease the programming burden.

```
# CREATE TABLE OBJECT WITHIN STAGE OBJECT
txv_table = table1_read( fname, "txv_table", line0)
ca_off_table = table2_read( fname, "ca_off", line0)
...
# ACCESS TABLE OBJECTS TO INTERPOLATE
txv = interp( txv_table, clock.t)
ca_off = interp( ca_off_table, amach, alph)
...
```

Figure 4. Object-Oriented Paradigm to Create and Access Tabular Data

The third critical infrastructure piece, vector and matrix support, needed no code development. Julia has excellent native linear algebra functionality. Vectors (and matrices) are manipulated in an intuitive fashion within Julia in an object-oriented manner. In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations that are supported with LAPACK [18] for its more advanced functions.

Extensive experimentation with coding techniques and structures to leverage Julia features was conducted with emphasis on code readability and then timing. Benchmark timing studies were conducted with the DE engine and table elements to understand optimum Julia coding practice for speed. For instance, multiple independent 2nd order transfer functions were run in parallel to prototype different DE engine Julia coding patterns. Large-scale table lookup benchmarks were conducted to optimize speed. Very importantly, readability was never sacrificed for speed in the final versions of these critical simulation elements. If a choice had to be made between speed and readability, the latter was always favored.

This familiarization process and incremental simulation build-up was invaluable for its discoveries. After much experimentation, the final Julia code could be made to execute approximately half as fast as equivalent C++ code for DE engine and table look-up benchmarks. Speed tips for Julia are well documented [19]. Some of the most important tips for speed, verified by experimentation, were no globals, liberal use of type declarations, and to avoid changing the type of a variable. All of these tips were observed to provide significant speed-up. One critical speed factor that was not cited in the Julia documentation was to use as "flat", or in-line, of an architecture as possible. In other words, underlying functions were nested as shallowly as possible without sacrificing modularity. For instance, the integrator objects for the MR states were defined and used in the main program instead of, perhaps more logically, embedding them in the models. Moving the integrator objects to the main DE engine loop and propagating them there (see Figure 2) significantly reduced runtime. A previous version involved calling each of the model objects and propagating their integrators there and was much slower. Thus, Julia's speed was found to be highly dependent on knowledge of detailed operational aspects of the language. This was observed in these pre-MR simulation development experiments and later confirmed

with the full-up simulation. Thus, Julia's ease-of-coding might be somewhat offset by required knowledge of particular coding constructs and practices.

A Julia MR, conforming to an OSK architecture, was built using the "lessons-learned" from building and benchmarking the infrastructure parts. The final preparation step was to ensure that the Julia simulation agreed with previous ones. The same trajectory as before was used. Trajectory details are summarized in Table 3. This trajectory was selected to exercise the missile dynamics in all channels (pitch and yaw, as well as axial) in order to "touch" all the objects' math models code.

Table 3. Benchmark Trajectory Details for a Hypothetical Rocket

Trajectory Description
• 3-stage hypothetical rocket • vertical launch with pitch-over • rotating earth • pre-programmed maneuver in pitch and yaw channels (case chosen to fully exercise steering code in pitch and yaw) • flight time = 100 sec • stage splits = 40, 75 at 2708, 6790 m/sec • final velocity = 6574 m/sec at t = 100 sec

Numerically, all the C++, Java, Python, and Julia results were very nearly identical. A trajectory overlay from all four MR simulations is shown in Figure 5. No difference is discernible.

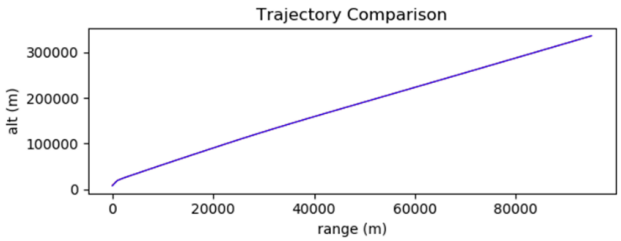

Figure 5. Trajectories For All Four Benchmark Simulations

Thus the stage was set for a fair benchmark comparison.

5. BENCHMARK RESULTS

Before assessing timing results, it was informative to collect lines-of-code (LOC) and compare them since all the models were coded conforming to the same OSK architecture. Table 4 decomposes the LOC count to the simulation infrastructure functions described earlier. In a sense, LOC could be one metric to judge coding efficiency. Note that no code had to be written for the Julia simulation's vector utilities since this functionality was built-in. On a similar note, no external vector libraries were used for the other languages so code had to be written for these. For instance, *numpy* [20] could have been used in-lieu of the natively-coded vector utilities in Python. For this reason, and to facilitate a more fair comparison, lines-of-code totals are shown with and without the vector code for C++, Java, and Python. Along these lines, it is important to note that the speed of the C++, Java, and Python MR versions might have benefitted from using external libraries. External libraries were not used since they are not a native part of these languages. Another biasing factor in favor of Julia was that the DE engine's integrators were propagated in the main program (as discussed earlier) which slightly altered the code architecture of the DE engine. In a similar manner as-before, LOC are shown without the DE engine to reflect this alteration in the Julia code. A related metric would be characters-of-code. Julia characters-of-code could have been greatly reduced without specifying types (at cost of great speed penalty and code readability as described earlier). These differences and evaluation criteria are indications of the difficult nature of benchmarking.

Table 4. Lines-of-Code Comparison

Simulation Function	C++	Java	Python	Julia
DE Engine	360	310	227	98
rocket model	830	753	596	613
vector utilities	533	524	351	0
table utilities	650	384	252	165
misc. utilities	104	153	64	82
Total (all)	2477	2124	1490	958
Total (less vector utilities)	1944	1600	1139	958
Total (less vector utilities& DE engine)	1584	1290	912	860

Table 5 provides details on the computer used for the benchmark. A relatively powerful computer (speed and memory) was used to limit the possibility that performance for any particular simulation would be influenced by hardware limits. The cores were not utilized. No special code was written in any of the MR versions to take advantage of the underlying cores to avoid the particulars of any of the languages' core utilization affecting the results.

Table 5. Benchmark Computational Hardware

Category	Description
Operating System	• Windows 10
Model	• Dell Precision 7820, Wintel High Performance Dual Socket Engineering Workstation
CPU	• Intel Xeon Gold 6130 CPU @ 2.10GHz (2 processors, 32 cores, 64 threads)*
Processing	• single-thread processing used for these runs

*each CPU has 16 cores, each core has 2 hyper-threads

Table 6 documents the language versions. Again, only the standard distributions of these compilers (and interpreter) were used; no numeric or vector libraries were added.

Table 6. Language Version

Language	Version
C++	Borland 5.5
Java	12.0.1
Python	2.7.16
Julia	1.1.1

Table 7 summarizes the principle results of the benchmark experiment. For convenience-of-interpretation, the times are normalized to the C++ (fastest) MR version. Initial data file reading and preprocessing were not included in the timing loop; timing profile code was wrapped only around the integrating time loop within the DE engine (see code in Figure 2). The very small amount of screen output was redirected to a buffered file (solid state drive) and no explicit compiler optimization flags were used for C++ and Java.

Table 7. Benchmark Results*

	1 run	10 runs	100 runs
C++	0.125 (1.00)	0.126 (1.01)	0.127 (1.02)
Java	0.169 (1.35)	0.139 (1.11)	0.099 (0.79)
Python	3.765 (30.12)	3.752 (30.02)	3.762 (30.10)
Julia	1.470 (11.76)	0.378 (3.02)	0.269 (2.15)

* normalized times in parentheses

Since Java and Julia use a Just-In-Time Compiler (JIT), sets of 10 and 100 runs were conducted to successively diminish the effect that the JIT compile time might have on the benchmark. As expected, Java and Julia got better with more runs (Julia more so than Python). It is speculated that both the Java and Julia JIT compilers got better at optimizing with more runs (or at least the compile time's share of total runtime was diminished with more runs). It is interesting to note that Java speed approached, or even exceeded, C++ for a large number of runs. The C++ and Python results (relative to each other) were the same as those found previously (see Fig. 1).

A language's utility for simulation cannot be best evaluated considering only runtime in isolation. As with the previous benchmark presentation (see Fig. 1), "coding efficiency" was considered adding a second dimension in the language evaluation metric. Recognizing the subjectivity of this metric, the question was asked, "How easy was it to code a working simulation?" The prominent role of the up-front experimentation provided experience to fairly address this question.

Figure 6 portrays the data in Table 7 against a "speed-of-coding" axis. Again, the coding efficiency was highly subjective based on prior experience in the general time required to stand-up a simulation. While the scripting language characteristics of Julia definitely expedited coding speed, it was definitely more difficult than Python owing to the speed-critical coding knowledge described earlier. Consequently, the Julia speed-of-coding metric was judged to be less than Python, but still greater than Java.

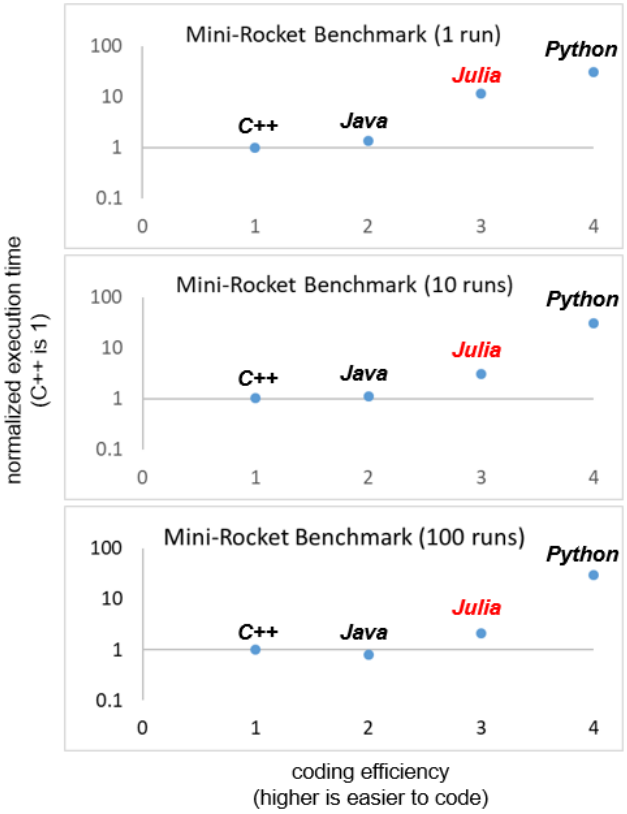

Figure 6. Benchmark Results With Coding Efficiency Dimension Added

Although not within the scope of this benchmark study, it was noted how easy it is to extend Julia execution for parallel processing on multiple threads. The Julia Distributed package [1, Distributed Computing] adds functionality to extend the single execution thread used up-until-now to multiple threads. The benchmark computer had 64 threads (Table 5). Kaminski and Szufel [21] provide very simple code that can be executed interactively to execute Julia scripts concurrently. This interactive command set was used to execute 65 MR runs concurrently (a master process and its 64 worker processes). Runtime results are shown in Table 8. It should be noted that easily-accessible multi-thread computing is not exclusive to Julia; the author has used the Python multiprocessing package [22] with equal success. Along the lines of parallel computing, Julia provides simplified access to any Graphical Processing Units (GPUs) that are available [23]. However, GPUs seem best suited for simple repetitive computation (over grids, for example) and do not seem suitable for the flight simulation-type calculations studied here.

Table 8. Multi-Thread Benchmark Results

Configuration	# concurrent runs	Time (sec)	Time/run (sec)
1 run/thread	65	3.00	0.0460
10 runs/ thread	650	7.63	0.0117
100 runs/ thread	6500	53	0.00815
1000 runs/ thread	65000	525	0.00808

6. CONCLUSIONS

Challenges still exist in providing the tools and environment for quickly and efficiently constructing dynamical system simulations that address every step in the missile simulation life cycle. The potential contribution of Julia is to give "non-expert" coders (scripters) the ability to build high performance simulations. Julia was well suited to coding the object-oriented structure in MR with an exceptional economy-of-code.

Julia execution speed was much faster than Python but still slower than C++ and Java. Julia speed was found to be highly dependent on knowledge of detailed operational aspects of the language. This was observed in the pre-simulation development experiments and confirmed with the full-up simulation. Thus Julia's ease-of-coding might be somewhat offset by required knowledge of particular coding constructs and practices. Also, Julia's JIT compiler becomes more efficient with multiple run execution. This is why the multiple run experiment design was an important benchmarking activity.

The economy of Julia to express complex programming constructs makes it attractive as a simulation experiment "testbed" for prototyping any future simulation applications. Although only touched upon in these results, parallel computing capability and its application to time-domain dynamic system simulations is especially compelling for further flight simulation research.

REFERENCES

[1] Julia language website: Julia.org.

[2] D. Bagley, "The Great Win32 Computer Language Shootout," http://dada.perl.it/shootout/

[3] "The Computer Language Benchmarks Game": https://benchmarksgame-team.pages.debian.net/benchmarksgame

[4] J. Kouatchou, "NASA modeling guru," https://modelingguru.nasa.gov/docs/DOC-2676

[5] Mad Data Scientist: https://matloff.wordpress.com/2014/05/21/r-beats-python-r-beats-julia-anyone-else-wanna-challenge-r/comment-page-1/

[6] D. Eadline, "Dirt Simple HPC," https://www.nextplatform.com/2016/01/26/dirt-simple-hpc-making-the-case-for-julia/

[7] D. Gutierrez, "Julia: A High-Level Language for Supercomputing," https://insidebigdata.com/2017/09/06/julia-high-level-language-supercomputing-big-data/

[8] Sells, R. and Sanders, G., "Mini-Rocket User Guide," U.S. Army Technical Report AMR-SS-07-27, [online archive] URL: http://www.dtic.mil/dtic/tr/fulltext/u2/a472173.pdf

[9] Open-Source: A Mechanism for Advancing Simulation State-of-the-Art Principles," Technology Alabama Magazine, Fall 2005, p. 11.

[10] R.C. Hibbeler, *Engineering Mechanics: Statics and Dynamics*, Macmillan Publishing Co., Inc., New York, pg. 473,1974.

[11] Sells, H.R., Sanders, G., and Saylor, R., "An Object-Oriented Simulation Kernel for a Large Spectrum of Simulations," Summer Computer Simulation Conference, Society for Computer Simulation, 2006, Calgary, Alberta.

[12] Sells, R., "A Code Architecture to Streamline the Missile Simulation Life Cycle," SciTech 2107 – Modeling and Simulation Technologies Conference, AIAA, Grapevine, TX, January 12, 2017.

[13] *Introducing Julia*, on-line book: https://en.wikibooks.org/wiki/Introducing_Julia

[14] J. Fernández-Villaverde, *Julia Tutorial*, https://www.sas.upenn.edu/~jesusfv/Chapter_HPC_8_Julia.pdf

[15] "The Fast Track to Julia," https://juliadocs.github.io/Julia-Cheat-Sheet/

[16] L. Baker, *C Tools for Scientists and Engineers*, McGraw Hill, New York, 1989.

[17] f2c wiki: https://en.wikipedia.org/wiki/F2c

[18] LAPACK website: http://www.netlib.org/lapack/
[19] "Julia Performance Tips," https://docs.julialang.org/en/v1/manual/performance-tips/index.html
[20] Numpy: numpy.org
[21] Kaminski, B. and Szufel, P., *Julia 1.0 Programming Cookbook*, Packt, Birmingham, 2018, pg. 35.
[22] "multiprocessing — Process-based "threading" interface": https://docs.python.org/2/library/multiprocessing.html
[23] S. Danisch," An Introduction to GPU Programming in Julia," https://nextjournal.com/sdanisch/julia-gpu-programming.

BIOGRAPHY

Ray Sells *received B.S. and M.S. degrees in Mechanical Engineering from Tennessee Technological University in 1980 and 1981. He currently is Vice President of Advanced Technology at DESE Research, Inc. where he is principally engaged in next-generation tactical missile system technology exploration and development. He has authored numerous publications in the diverse areas of thermal analysis, large-scale shock isolation systems, missile autopilots and guidance laws, simulation architectures, embedded software, data visualization, genetic algorithms, and missile defense system operations research. He is currently supporting the NASA Space Launch System Flight Dynamics team at MSFC.*