AAS 15-795

EXPERIMENTS WITH JULIA FOR ASTRODYNAMICS APPLICATIONS

Nitin Arora[*] and Anastassios Petropoulos[†]

Julia's potential for solving complex astrodynamics problems is studied. Julia is a high-level, new, dynamic programming language with performance approaching C/Fortran and has features like inbuilt parallelism, variable accuracy, integrated numerical libraries and direct C and Fortran interfaces. Two astrodynamics problems are solved in Julia: 1) Lambert's problem, using the vercosine formulation and 2) trajectory integration. Implemented algorithms are compared with C and Fortran based counterparts on key performance parameters (speed, development effort, etc). Using Julia for fast and reliable astrodynamics software development is also discussed.

INTRODUCTION

With the advent of modern computing, complex algorithms (serial or parallel) are increasingly being adopted for solving a variety astrodynamics problems like multiple-gravity-assist impulsive trajectory search, tour design, space situation awareness, parallel high-fidelity trajectory propagation, multi-body low-thrust trajectory design and optimization etc. Dynamically typed languages like MATLAB, Octave, Python (with SciPy and Numpy) provide the programmer with a high-level, interactive, dynamic environment for algorithm development and are increasingly being preferred for rapidly implementing some of these complex algorithms in astrodynamics. One the other hand, statically typed languages like C and Fortran still remain as the prime choice for implementing computationally intensive algorithms in various scientific disciplines, including astrodynamics.

A common strategy to achieve high computational efficiency while still enjoying the benefits of a high-level dynamic programming language, is the two-language approach (a result of Ousterhout's dichotomy[1]). In this approach, a high-level dynamic language (e.g Python) acts as a front end with compute-intensive parts of the algorithm compiled in C or Fortran into functions or libraries which are then glued together using a wrapper layer. Even though this leads to better runtime performance than just using a dynamic language, there is an inherit compromise associated with this approach. Extra computational and personnel resources are required to program and maintain the glue layer (consisting of complex wrappers) between the two languages. This added complexity also results in compatibility issues, hard to detect errors, poor code readability, limited parallelism, complex debugging procedures, and performance degradation due to the wrapper layer itself. Furthermore, as modern astrodynamics algorithms have increased in complexity, it is becoming increasingly difficult

[*]Jet Propulsion Laboratory, California Institute of Technology, Nitin.Arora@jpl.nasa.gov, 818-354-2417
[†]Jet Propulsion Laboratory, California Institute of Technology, anastassios.e.petropoulos@jpl.nasa.gov

to divide them into compute-intensive and non-compute-intensive parts. It is often easier to write more efficient and maintainable software without forcing such a distinction.

To solve the above stated problems, new programming languages, challenging the Ousterhout's dichotomy, have been proposed (like Julia,[2] Google's GO,[3] Apple's Swift,[4] PyPy[5] etc.) and are under rapid development. The most relevant of new programming languages, targeted specifically for numerical and scientific computing, is Julia.[2,6] Julia is a open source, high-level dynamic programming language capable of performance (execution speed) approaching that of compiled C or Fortran code. It has syntax similar to other technical computing environments like MATLAB and enjoys attractive features like inbuilt parallelism capabilities, variable numerical accuracy, easy plotting capabilities, inbuilt numerical libraries (like BLAS, LAPACK etc.) and tightly integrated high performance solvers/optimization packages (JuMP,[7] Sundials,[8] Ipopt,[9] KNITRO[10] etc.). Another major attractive feature is Julia's ability to call Python, C and Fortran functions without any wrapper or glue code. This makes it possible to easily leverage optimized implementations of existing astrodynamics function and libraries written in C or Fortran. More details can be found on the Julia programming language website.[11]

Recognizing that the astrodynamics community stands to benefit from this new development, we evaluate Julia's performance, both from a user and a programmer perspective, on two key astrodynamics algorithms and problems. The first problem which is selected is the Lambert's problem which is solved using the recently proposed universal vercosine formulation.[12] We also evaluate the performance of a custom Julia integrator on an example two-body problem with varying eccentricity. Algorithms implemented in Julia are compared to the Fortran based counterparts. A discussion on using Julia for programming fast, reliable astrodynamics software is also given.

BRIEF OVERVIEW OF LANGUAGE FEATUERS

Figure 1 gives relative performance of various languages (including Julia v 0.3 and Fortran) relative to C. Even though this benchmark is somewhat old, the fact that Julia is able to come close to C and Fortran is quite remarkable. Figure 1 has been taken from the 2014 Julia language reference article.[2] This high performance is even more remarkable considering the fact the Julia is dynamically typed like Python and syntactically math-friendly like MATLAB.

Julia achieves high computational performance by using the concept of type stability and leveraging the just-in-time (JIT) LLVM compiler.[13] A program written in Julia with no variable type information can be as slow as MATLAB or Python, but addition of few type keywords, results in orders-of-magnitude faster execution times.

Julia also provides; a large base library which is mostly written in Julia itself, open source C and Fortran libraries for linear algebra, random-number generation, signal processing, and string processing. Julia's developer community on Git hub is very active and is contributing a variety of numerical packages thanks to the language's built-in package manager. Similar to Ipython,[14] there also exists a web browser based version of Julia known as Ijulia (not tested for this work).

Relevant language features

Table 1 gives a list of language features relevant to astrodynamics along with a brief discretion and possible applications. Some of the planned features in the upcoming Julia v-0.4 release are also mentioned.

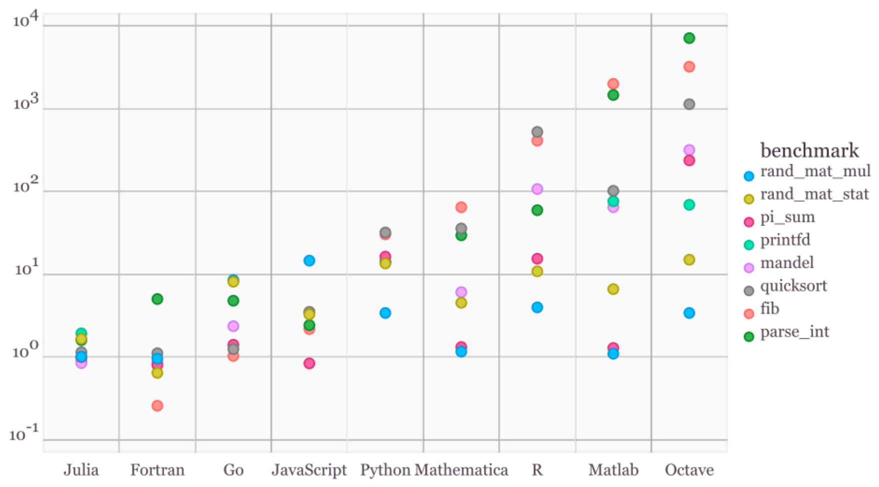

Figure 1. Benchmark times relative to C (smaller is better, C performance = 1.0); plot from article[2]

Table 1. Partial list of Julia features

Feature	Description	Benefits	Application
Fast execution times	via type stability and JIT compiled code	execution speed (∼0.5X Fortran)	all algorithms
Multiple dispatch	function behavior based on argument types	code size, performance	complicated algorithms
Math based syntax	syntax similar to Python / MATLAB	code development, performance	scripting, optimization, driver layer
Packages	inbuilt package manager	code development, ease of use	all algorithms
C/Fortran interface	direct access to C/Fortran functions	code development, ease of use	reuse of existing astrodynamics software
Parallel programming	part of the language	performance	parallel trajectory integration, broad search etc.
Large numerical library	inbuilt linear algebra/optimization packages	performance, ease of use	trajectory optimization, mission design
Rational/Infinite arithmetic	inbuilt rational, infinite precision arithmetic	numerical studies, code debugging	algorithmic comparisons, trajectory integration
Meta programming	ability to auto generate problem specific code	ease of use, code generalization	large mission design tools
Interactive shell	interactive shell like Python, MATLAB	development, code profiling and debugging	All algorithms
Code compilation (in v0.4)	compile binaries	code distribution, ease of use	All algorithms
Unicode support	support characters variables (δ, π etc.)	ease of use	All algorithms
Open source	MIT open source license	free to use	All algorithms

Julia's intuitive syntax

This section is intended as a brief introduction to some aspects of Julia's syntax to give a flavor of how easy it is to get started with Julia. There is much greater depth that is left for the interested reader to explore using Julia's on-line documentation and other on-line resources developed by the vibrant, numerical-programming-minded user community.

Julia was born out of a desire to make easier the numerical programming so often used in the scientific and engineering disciplines. With this pedigree in mind, Julia's fundamental data types include of course integers and real numbers of various ilks (signed, unsigned, 32-bit, 64-bit, etc). User-defined data types, which can be thought of as structures, can also be used, for example to handle quaternions, or to over-ride Julia's default handling of complex numbers. Julia's syntax makes it very easy to use not only scalar variables, but especially numerical arrays and parts thereof. Arrays and their subparts can be operated upon and referenced with a very concise and intuitive syntax, without sacrificing execution speed. For astrodynamics this is crucial, since our bread and butter are vectors (position, velocity, thrust, etc) and matrices (rotation matrices, state transition matrices, etc). Furthermore, Julia uses the concept of multiple dispatch (similar to function overloading), which makes it easy and efficient to have functions or operators that can act on a variety of data types. For example, one can easily list the datatypes and methods associated with the "+" operator, and can then add to them or modify them. User-defined functions can be easily created, both at the

Julia command line and within scripts; specifically, there is no need to keep the functions in a separate file, which is handy for functions that aren't generic enough to warrant creating a package (i.e. library) to contain them. Type declarations are optional for variables; if they are implicitly typed, their type is determined based on what they are assigned. Type declarations are currently possible only in functions, not at the command line (or REPL) since Julia does not yet have constant-type globals. Type "assertions" are, however, possible at the command line and inside the functions; they serve as type checks since an exception is thrown if the expression does not evaluate to the asserted type.

Here we develop a simplistic, short, toy exercise to demonstrate some of the above aspects using the *vis viva* equation for an Earth-orbiting spacecraft.

One can type the following at the Julia command line directly after starting an interactive Julia session. (Alternatively, it can be typed into a file, named say `jscript.jl`, and the file invoked at the command line by typing `include("jscript.jl")`.) The semi-colon character prevents the value from being echoed to the screen.

```
μ       = 368600.5;     # μ= gm, default: 64-bit float:  Float64
a       = 7e3;          # period used as decimal point
r       = 6.9e3;

vvv(μ,a,r) = sqrt( 2*μ ./r - μ ./a ); # vis viva velocity function
# () used for function arguments
# , used to separate arguments
vvv(μ,a,r)
# 7.360944936077629 is printed to the screen

# Let's say we have some uncertainty in the range, r, and represent
# it by taking 100 random samples and storing them in an array:

runc = r + (rand(100)-0.5)*10;
# array created, named ``runc''
# Note: Intelligent element-wise operations:
#       100 random numbers (``zero-mean''), each multiplied by
#       10 and each product added to r

# Compute velocity for the 3 smallest and 3 largest values in runc
runc_s = sort(runc);
vvv(μ, a, runc_s[ [1:3 ; end-2:end] ])'

### the following is printed to the screen:
1x6 Array{Float64,2}:
7.36613  7.36603  7.36603  7.35591  7.35585  7.35583
###
# Note: function vvv accepts array input since all functions and
#           operators used in the definition accept array arguments
#           e.g. ./ means element-wise division.
#       ' means transpose (would otherwise have a column vector)
#       [] used to dereference array elements
#       array indices start at 1, ``end'' denotes last array index
#       : is the range operator (includes upper limit, if possible)
#       1:3 is a column vector with the digits 1,2,3
#        (generally, eg 1:2:6 means from 1, by 2 to no more than 6)
#        ; means vertical concatenation
#        innermost [] creates the array of indices into runc_s
```

A word now on Julia's multiple dispatch features. Julia's official on-line documentation[15] has this to say:

> Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most powerful and central feature of the Julia language. Core operations typically have dozens of methods:
> ...
> Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details, yet generate efficient, specialized code to handle each case at run time.

One should add that out of the box, this specifically applies to numerical algorithms and scientific computing. One can easily extend these capabilities to be tuned to other domains in computing, if desired, but, happily for engineers in the astrodynamics field, the numerical features are largely ready-to-use in a coherent framework. For example, in the above code, the function `vvv()` automatically handles numerical arrays in the expected way (*e.g.* the syntax `2*r` means multiply each element of the vector `r` by `2`).

Multiple dispatch can be demonstrated in our *vis viva* example in the following admittedly frivolous way. Let us add a method to the `vvv()` function, a method which is used only when the third argument is of type `Int64`; this method does not replace the previous method, but is added to the list of available methods for the function. The following is typed directly at the command line:

```
function vvv(μ,a,r::Int64)  # vis viva alternative with a silly twist
μ      = int(μ); # round to nearest integer (not truncation)
a      = int(a); # round to nearest integer (not truncation)
v      = sqrt(2*μ./r-μ ./a); # Int64 promoted to Float64 for operations
result = itrunc(v);    # truncate to integer (remove decimal part)
end
methods(vvv)
# the following is printed to the screen:
# 2 methods for generic function "vvv":
vvv(μ,a,r::Int64) at none:2
vvv(μ,a,r) at none:1
#

vvv(μ,a,r)
# the following is printed to the screen:
7
#

vvv(μ,a,int(runc_s[1:3]))'
# the following is printed to the screen:
1x3 Array{Float64,2}:
7.36621  7.36621  7.36621
#
# this is because
# we have not defined a method for vvv(μ,a,r::Array{Int64})
# so the function call falls through to the generic method.
```

Formatted printing to a file or to the screen is another commonly needed task. There are a variety of functions and built-in macros (denoted by `@`) that can be used for printing, such as `show()`, `print()`, `println()`, `print_with_color()`, `@printf()`, `@sprintf()`. The following screen output provides a brief example of using the `@printf()` macro, including C-style print format specifications, automatic string conversions, unicode characters, and the special nature of built-in constants.

```
julia> @printf("Printing pi %lf\n",pi)
Printing pi ERROR: `decompose` has no...
...method matching decompose(::MathConst{:?})
in isfinite at float.jl:214

julia> @printf("Printing pi, %s\n",pi)
Printing pi, π= 3.1415926535897...

julia> @printf("Printing pi, %s\n",pi*1)
Printing pi, 3.141592653589793

julia> @printf("Printing pi, %lf\n",pi*1)
Printing pi, 3.141593

julia> @printf("Printing pi, %f\n",pi*1)
Printing pi, 3.141593

julia> @printf("Printing e, %.5lf\n",e*1)
Printing e, 2.71828

julia> print("Another ", "printing of pi ", 1*pi)
Another printing of pi, 3.141592653589793

julia> str=sprint(print,"Another ", "printing of pi, ", 1*pi);
julia> str
"Another printing of pi, 3.141592653589793"

julia> print("Yet another ", "printing of $pi")
# using Julia's variable substitution capability
Yet another printing of π = 3.1415926535897...
```

We have provided only the simplest of overviews of some of Julia's syntax. Just with the functions, for example, we omitted any description of multiple return values, variable number of arguments, optional and keyword arguments, anonymous functions. Sometimes, however, these simple things are the hardest to find in the documentation. Perusing the documentation shows the language's breadth as well as the fastidious concern that the authors of Julia have for numerical performance and ease of use.

Numerical packages in Julia

Even though Julia is new, it already enjoys a large number of quality optimization and numerical algorithm packages. The first place to start is the JuliaOPT organization which brings together various optimization related packages in Julia. Packages for Ipopt,[9]NLopt,[16,17] KNITRO,[10] Sundials[8] and other highly respected numerical optimization software are already available. Download, installation and update of these packages is controlled by the inbuilt package manager, thereby eliminating the need for manual installation (which is not always a smooth process). Furthermore, Julia also has mathematical modeling packages for a limited set of optimization problems (e.g JuMP, MathProgBase). Finally, packages for symbolic computation (similar to Maple) and automatic differentiation are available or under rapid development. Julia also has a number of numerical integration packages available with MATLAB like interface. A subset of them is tested in this paper against our own implementation.

Calling other programming languages

A useful feature in Julia is its ability to directly interface with compiled C and Fortran functions and libraries. This feature is especially useful to the astrodynamics community due to a large number of C/Fortran optimized legacy software.

We briefly touch on calling from Julia functions or procedures that are defined in shared object libraries (or, dynamically linked libraries, in Window's parlance), for example in Fortran. The specific name to be used for the procedure will depend on the Fortran compiler, since compilers are free to mangle or adorn names as they see fit in compiling the source code. For example, suppose we have a Fortran library which contains the function `angv` and which has been compiled into a shared object library named `libmymath.so`, wherein the function name has been mangled to `angv_` by the compiler. Then, this function, which returns the angle between two vectors of length `n` may be called as follows from Julia (v 0.3.9):

```
julia> angle = ccall(
(:angv_, ``path/to/libmymath.so''), Cdouble,
(Ptr{Cint}, Ptr{Cdouble}, Ptr{Cdouble}),
&n,Vec1, Vec2
)
```

Note that simpler versions of the function "`ccall`" also exist in the language. Function pointers to "C" function, can also be obtained using the "`cfunction`" in Julia.

Furthermore, the calls to the compiled C and Fortran are inlined by the LLVM compiler (during JIT compilation) and have almost no runtime overhead. The user should be aware of difference, in the syntax when switching from Julia-0.3 to Julia-0.4.

Julia also has the ability to directly call Python and MALTAB functions and scripts. Given the heavy use of MATLAB in the astrodynamics community, having a direct interface in Julia is very useful. The "MATLAB.jl" package[18] provides just that. The authors have successfully used the package for various problems and the interface seems to work fine.

Plotting in Julia

Given that Julia is a modern, dynamic language, it supports inline plotting capabilities in similar ways to MATLAB and Python. There are currently three plotting packages available in Julia:

- Gadlfy
- PyPlot (Matplotlib)
- Winston

Out the three, Gadfly is the most popular due to its ability to follow the "Grammar of Graphics",[19] resulting in beautiful, clean-looking plots. The only draw back to Gadfly is that it is still under development and some critical features like 3D plotting are missing. One the other hand, "PyPlot" which is basically a Julia interface to the widely used Python "Matplotlib" graphics library, has a large number of features and has syntax similar to that of MATLAB. The third package, Winston, is a 2D plotting package for Julia with a minimal interface (simpler than MATLAB in some cases). All plots in this paper are made using Gadfly.

APPLICATION TO ASTRODYNAMICS

In this section we briefly compare performance between Fortran and Julia implementations of solutions to two astrodynamics problems. All algorithms are tested on an Intel Xeon E5-2630 CPU running the GFortran 4.8.3 compiler. The latest [underdevelopment, quasi-stable] version (0.4, August/01/2015) of Julia is used for performance comparisons.

Lambert's Problem

The first problem which is solved in Julia is the zero-revolution Lambert problem. We implement the recently proposed vercosine algorithm in Julia and compare it to the highly optimized Fortran.[12] The vercosine Lambert algorithm takes advantage of geometry-based parameters to simplify the universal formulation of the Lambert time-of-flight equation. This equation, defined as a function of the universal variable, requires only a single transcendental function evaluation and enjoys 40% to 200% reduction in runtime over the popular Gooding's method.[12] Both Julia and Fortran Lambert algorithm implementations are set to a dimensionless tolerance of 1E-13.

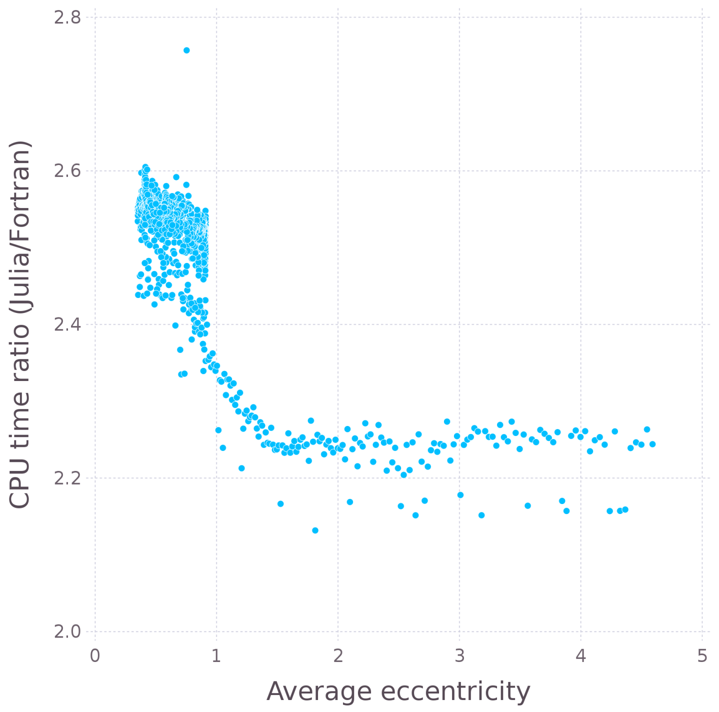

Figure 2. Lambert solver; performance relative to optimized Fortran version

The Fortran version of the algorithm takes ∼1400 lines, one the other hand the Julia version only takes ∼400 (liberally) lines of code. The code development time for the Julia version consisted of: few hours to write the code and a day to test and optimize the implementation. In contrast, the Fortran version was developed over many weeks and is optimized for high performance.

Figure 2 shows the ratio of the CPU time take by the Julia version as compared to the Fortran version for different values of transfer orbit eccentricities. There are in total 1000 points shown in Fig. 2 and each point corresponds to 3000 Lambert transfers, resulting in a total of 3 million Lambert transfers. The Julia version of the vercosine Lambert algorithm is on average 2.5 times slower than its Fortran counterpart. The difference in speed between the hyperbolic and elliptical case is explained by the difference in the underlying algorithm for the two transfer types. The vercosine algorithm relies on an inverse cosine ($\arccos$) function for the elliptical case which is currently more easily optimized in a statically compiled language like Fortran. For the hyperbolic case the algorithm relies on $sqrt$ and $\log$ functions, which enjoy highly optimized implementation in Julia.

It should be noted that only a few hours were spent towards improving the Julia version. Further optimization of the Julia code is likely to improve code performance.

Simple orbit propagation in Julia

The next problem solved is that of a simple two- sbody orbit propagation. The focus here is on the integrators themselves.

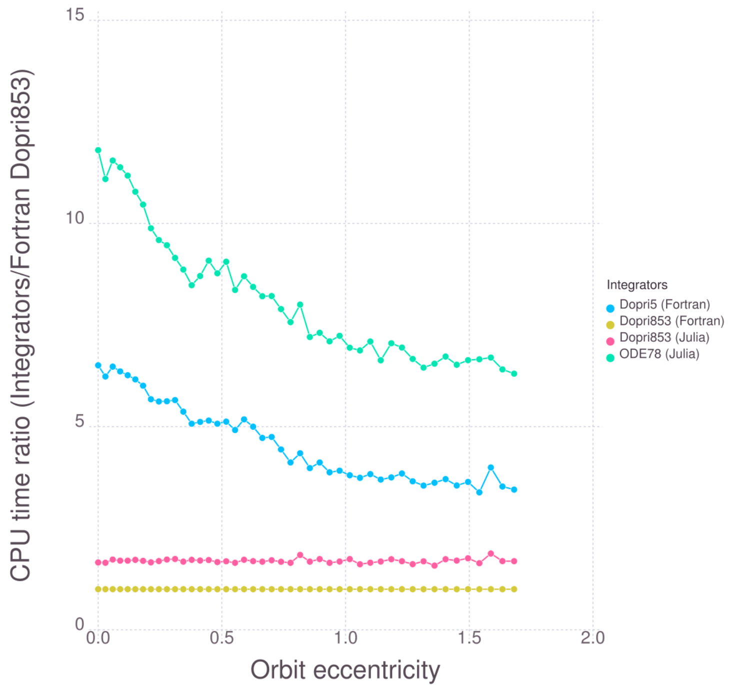

Figure 3. CPU time ratios relative to Dopri 8(5,3) (Fortran version)

We implement a custom integrator in Julia based on the Dopri8(5,3) algorithm developed by

Hairer, Norsett and Wanner.[20] The method is based on a modification of the original Dormand-Prince, Dopri8(7) method[21] (8^{th}-order integration with 7^{th}-order corrections). Instead of a 7th-order correction Dopri8(5,3), uses a 5^{th}-order error estimate and does a 3^{rd}-order correction. This update improves algorithm stability and makes it possible to use with discontinuities. The updated method uses 13 stages and uses in total 12 unique function evaluations per step.

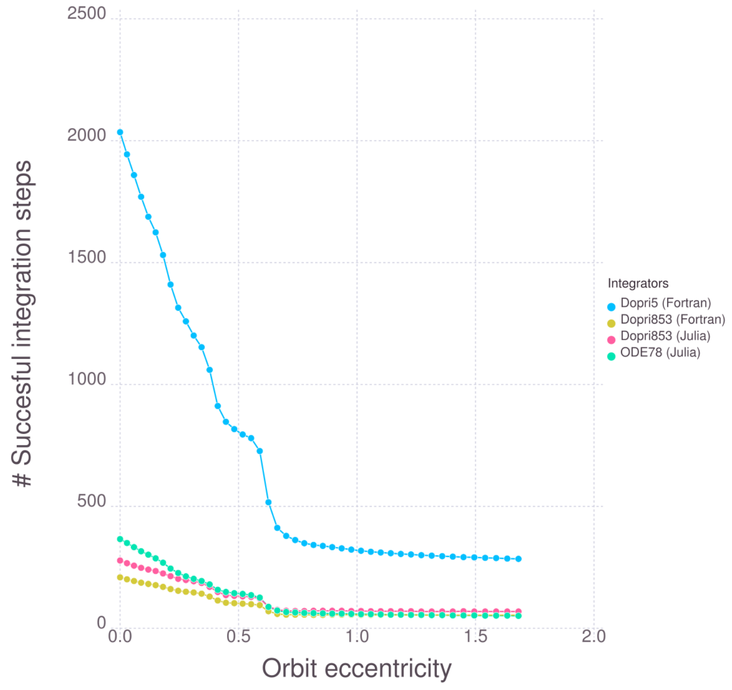

Figure 4. $\#$ of steps relative to Dopri 8(5,3) (Fortran version)

We compare our implementation in Julia (Dopri8[5,3]) to its equivalent Fortran version, available via the package "Dopri.jl". The package provides a simplified interface in Julia to an optimized Fortran implementation of the Dopri8(5,3) and Dopri5 (an explicit Runge-Kutta method of order 5[4] (using Dormand-Prince coefficients) algorithms.[22] Both algorithms have been developed by are available from Hairer and Wanner.[23]

Finally, we also evaluate the Runge-Kutta7(8) (called ODE78) integrator available from the "ODE.jl" package written entirely in Julia. Note, that this package is still under development and integrator is not yet optimized for performance.

Figures 3 and 4 shows performance of these integrators on a two-body orbit propagation problem with varying eccentricity. The CPU time numbers are plotted relative to the Dopri8(5,3) Fortran version, which is expectedly the fastest. Our optimized implementation of the Dopr8(5,3) algorithm is found to be $\sim$1.7 times slower that the optimized Fortran version and it outperforms the Dopri5 (Fortran) and ODE78 implementations. Another thing to note is that even though ODE78 (Julia) takes fewer steps than Dopri5 (Fortran), it requires more function evaluations per step and is also not optimized for performance.

CONCLUSION

Application of Julia, a new, dynamic, high-performance, just-in-time-compiled programming language for solving astrodynamics problems is investigated. Some of the salient features of the language are listed and briefly discussed. Julia-based implementations of the vercosine Lambert's algorithm and the Dopri8(5,3) numerical integrator are developed and compared to their optimized, Fortran-based counterparts. After only a few hours of development time, the Julia-based Lambert solver is found to be approximately 2.5 times slower compared to an optimized Fortran version. The Julia-based Dopri8(5,3) numerical integrator, when compared to its equivalent Fortran version, is found to be $\sim$1.7 slower. It also shown to outperform a 5^{th}-order Dopri integrator implemented in Fortran.

The relatively high performance of Julia in terms of computation time for a scripting language, coupled with Julia's mathematical-programming pedigree, ease-of-use, flexibility and breadth, make Julia an attractive option for solving the numerical problems encountered in astrodynamics.

ACKNOWLEDGMENT

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

REFERENCES

[1] J. Ousterhout, "Scripting: higher level programming for the 21st Century," *Computer*, Vol. 31, Mar 1998, pp. 23–30, 10.1109/2.660187.

[2] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *ArXiv e-prints*, Nov. 2014.

[3] "Google GO programming language," `http://golang.org/`. Accessed: Aug. 2015.

[4] "Apple Swift programming language," `https://developer.apple.com/swift/`. Accessed: Aug. 2015.

[5] "PyPy programming language," `http://pypy.org/`. Accessed: Aug. 2015.

[6] A. Edelman, "Julia: A fresh approach to parallel programming," *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, IEEE, 2015, pp. 517–517.

[7] M. Lubin and I. Dunning, "Computing in Operations Research Using Julia," *INFORMS Journal on Computing*, Vol. 27, No. 2, 2015, pp. 238–248, 10.1287/ijoc.2014.0623.

[8] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Transactions on Mathematical Software (TOMS)*, Vol. 31, No. 3, 2005, pp. 363–396.

[9] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, Vol. 106, No. 1, 2006, pp. 25–57, 10.1007/s10107-004-0559-y.

[10] R. H. Byrd, J. Nocedal, and R. A. Waltz, "KNITRO: An integrated package for nonlinear optimization," *Large-scale nonlinear optimization*, pp. 35–59, Springer, 2006.

[11] "Julia programming language," `http://julialang.org/`. Accessed: Aug. 2015.

[12] N. Arora and R. P. Russell, "A Fast and Robust Multiple Revolution Lambert Algorithm Using a Cosine Transformation," *AAS/AIAA Astrodynamics Specialist Conference, Hilton Head, AAS 13-728*, 2013.

[13] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.

[14] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science & Engineering*, Vol. 9, No. 3, 2007, pp. 21–29.

[15] "Julia's online documentation," `http://julia.readthedocs.org/en/latest/manual/methods/`. Accessed: Aug. 2015.

[16] S. G. Johnson, "The NLopt nonlinear-optimization package," `http://ab-initio.mit.edu/wiki/index.php/NLopt`, 2014.

[17] U. Kumar, S. Soman, *et al.*, "Benchmarking NLopt and state-of-art algorithms for Continuous Global Optimization via Hybrid IACO," *arXiv preprint arXiv:1503.03175*, 2015.

[18] "Julia's MATLAB package," `https://github.com/JuliaLang/MATLAB.jl`. Accessed: Aug. 2015.

[19] L. Wilkinson, *The Grammar of Graphics*. Springer-Verlag New York, 2^{nd} ed., 2005, 10.1007/0-387-28695-0.

[20] E. Hairer and G. Norsett, S.P.and Wanner, *Differential Equations I. Nonstiff Problems*, Vol. 8. Springer-Verlag Berlin Heidelberg, 2nd,springer series in computational mathematics ed., 1993, 10.1007/978-3-540-78862-1.

[21] P. Prince and J. Dormand, "High order embedded Runge-Kutta formulae," *Journal of Computational and Applied Mathematics*, Vol. 7, No. 1, 1981, pp. 67–75.

[22] "Julia's Dopri package," `https://github.com/helgee/Dopri.jl`. Accessed: Aug. 2015.

[23] "Dopri8(5,3) and Dopri5 orginal implementations," `http://www.unige.ch/~hairer/software.html`. Accessed: Aug. 2015.