A COMPARATIVE STUDY OF PROGRAMMING LANGUAGES FOR NEXT-GENERATION ASTRODYNAMICS SYSTEMS

Helge Eichhorn, Reiner Anderl

Technische Universität Darmstadt
Dept. of Computer Integrated Design
Darmstadt, Germany

Juan Luis Cano

Universidad Politécnica de Madrid
Madrid, Spain

Frazer McLean

CS GmbH
Darmstadt, Germany

ABSTRACT

Due to the computationally intensive nature of astrodynamics tasks, astrodynamicists have relied on compiled programming languages such as Fortran for the development of astrodynamics software. Interpreted languages such as Python on the other hand offer higher flexibility and development speed thereby increasing the productivity of the programmer. While interpreted languages are generally slower than compiled languages recent developments such as JIT (just-in-time) compilers or transpilers have been able to close this speed gap significantly. Another important factor for the usefulness of a programming language is its wider ecosystem which consists of the available open-source packages and development tools such as integrated development environments or debuggers.

The study compares three compiled languages and three interpreted languages which were selected based on their popularity within the scientific programming community and technical merit. The three compiled candidate languages are Fortran, C++, and Java. Python, Matlab, and Julia were selected as the interpreted candidate languages. All six languages are assessed and compared to each other based on their features, performance, and ease-of-use through the implementation of idiomatic solutions to classical astrodynamics problems.

We show that compiled languages still provide the best performance for astrodynamics applications but JIT-compiled dynamic languages have reached a competitive level of speed and offer an attractive compromise between numerical performance and programmer productivity.

Index Terms— scientific programming, computational astrodynamics, open-source software

1 INTRODUCTION

The choice of programming language for a project is usually a highly subjective or political matter. Every person has their own personal preferences and introducing a new programming language into an existing environment will inevitably cause disruptions. While Fortran 77 has been the gold standard in astrodynamics for many decades we argue that the introduction of more modern programming languages is a sensible investment despite the migration costs. With this study we aim to provide scientifically sound evidence for the merit of different programming languages and give organizational leaders and project managers the opportunity to make an informed decision.

While the features of the programming language itself are important its so-called ecosystem, which includes toolchains, programming tools, libraries, and also online communities, should also be considered. The study is therefore structured into two main parts: a general comparison of the candidate languages and implementations and benchmarks of classical astrodynamics problems.

Our main argument is that the conventional wisdom that a programming language can either be statically compiled and fast or dynamically interpreted and easy to work with is no longer true. We have therefore included three languages from either category in this study.

- Compiled languages:
  - Fortran
  - C++
  - Java
- Interpreted languages:
  - Matlab
  - Python
  - Julia

1.1 METHODOLOGY

To assess the suitability of programming languages for astrodynamics applications we have indentified the following requirements:

Numerical Performance
The iterative nature of many astrodynamics algorithms makes high numerical performance a necessity.

Linear Algebra Capabilities
Vector- and matrix-based calculations are equally important. A programming language for astrodynamics thus needs to provide convenient access to linear algebra routines.

Concise Syntax
Usability research by Prechtelt suggests that programmers produce the same number of code units (e.g. lines) per unit time regardless of the programming language they use [1]. Hence a concise language improves programmer productivity and reduces the size of the overall codebase which eases maintenance.

Interfacing with Legacy Code
While it might be tempting to start on a green field it is economically unsound to reimplement everything from scratch. To preserve the vast organizational knowledge contained in legacy libraries the candidate languages shall be able to call code written Fortran 77 and C. Should procedures be reimplemented in a new language the capability to call legacy code is also useful for verifying and validating the new implementation.

Programming Environment
In our experience the development of algorithms or exploratory data analyses are greatly simplified through interactive environments such as an Read-Eval-Print-Loop (REPL). Other useful tools include debuggers, integrated development environments (IDE), or visualization libraries.

The core astrodynmical calculations are often just a small part of a larger astrodynamics software system. Most codes also need to read and write input and output files and conduct standard tasks such as searching or sorting data. It is therefore beneficial if a programming language distribution provides a standard library which covers these functions.

Availability of Open-Source Libraries
Not only does the availability of open-source libraries save the programmer from "reinventing the wheel" it is also an indicator for the popularity of language. The more popular a programming language is the easier it becomes to find help in online communities and the higher the probability that students and young professionals already have experience with the language.

To test the aforementioned functional requirements under realistic conditions we have implemented the following astrodynamics problems in the six target languages:

- Calculation of the classical Keplerian elements from the Cartesian state vector (see 3.1).
- Solving Kepler's equation (see 3.2).
- Solution of Lambert's problem (see 3.3).
- Runge-Kutta orbit propagation (see 3.4).

2 CANDIDATE LANGUAGES

Fortran
Fortran probably does not need an introduction since it has been a staple of computational astrodynamics from the very beginning. The Fortran 77 standard has been especially successful and many organizations still use and maintain large codebases in this version of the language. In fact the core linear algebra routines such as BLAS or LAPACK that most of the other languages discussed in this study use are based on hand-optimized Fortran77 and assembly. But the language has evolved considerably during the last decades. Fortran 90/95 moved from punchcard-based fixed-form source code to free-form code and simplified array operations. Object-oriented features were introduced in Fortran 2003 and the latest standard Fortran 2008 added constructs for array-based parallelism [2].

C++
C++ is a high-performance, multi-paradigm programming language with a mature toolset and many options for compilers, IDEs, and other tools. C++ is usually critized for its complexity because there are generally multiple ways to solve a problem in the language. The standard reference written by the language's creator which includes more than 1300 pages seems to confirm this [3]. While this fact makes the language very powerful it also makes the language hard to master and complicates the composition of software libraries that use different "dialects" of C++. We conclude that an organization that introduces C++ should generally determine beforehand which subset of the language should be used to keep the complexity of the resulting software manageable.

Java
Java was originally developed by Sun Microsystems with the goal of being portable, secure and easy to learn. The first version was released in 1995 and Sun Microsystems has since been acquired by Oracle. Although Java has been extended with generic and functional programming features its main programming paradigm is class-based object-orientation. The Java compiler does not emit native machine code but Java bytecode which is then executed on the Java Virtual Machine (JVM). In the past Java had the reputation to be slow but this is no longer the case [4] and it has become an important part of the scientific programming toolset, especially in the realm of so-called "big data" analyses [5].

Matlab
Matlab is not only a programming language but also a full featured technical computing environment. The Matlab IDE

posses powerful interactive computing, visualization, and debugging features. It also ships with an extensive library of useful functions and MathWorks provides an online platform for sharing open-source Matlab code. The biggest point of criticism is its price. This is made worse by the fact that many features and libraries are sold separately as so-called toolboxes.

Python
While Python is a general purpose language we consider it a suitable language for scientific computing due to the existence of well established libraries like NumPy and SciPy for mathematical calculations. Not only do these libraries make use of compiled code from existing accelerated libraries like BLAS and LAPACK, but it is also possible in Python to interface with compiled code to speed up specific algorithms where required. This allows a user to define a problem in Python, while benefiting from the speed of compiled languages for computationally expensive algorithms.

Julia
Julia is the youngest language in this study. Its development began as a research project at the Massachusetts Institute of Technology (MIT) in 2009 and the first public version was released in 2012. It uses a JIT-compiler based on LLVM to combine Matlab-like syntax with performance close to C and Fortran [6].

Due to its young age the language is still in flux and some essential, e.g. a debugger, are still under development.

3 ASTRODYNAMICS APPLICATIONS

In this chapter we compare the implementations of the aforementioned astrodynamics applications within the six candidate languages. We will only list sections of the source code that are relevant to the discussion here. Please refer to the complete source code published online at GitHub (`https://github.com/helgee/icatt-2016`) for further information.

3.1 Calculating the Keplerian Elements

In this simple example we calculate the classical Keplerian orbital elements from the Cartesian state vector and compare the syntax of the candidate languages for expressing vector equations. We will discuss the syntax differences based on the derivation of the eccentricity vector as shown in the following equation.

$$\mathbf{e} = \frac{\left(\|\mathbf{v}\|^2 - \frac{\mu}{\|\mathbf{r}\|}\right)\mathbf{r} - (\mathbf{r}\cdot\mathbf{v})\mathbf{v}}{\mu}$$

Fortran
The calculation is straight-foward in Fortran. The exponentiation can be conveniently expressed through the `**` operator and the inner product through the `dot_product` intrinsic function.

```
e = ((v_mag**2 - mu/r_mag)*r -
 ↪  dot_product(r,v)*v)/mu
```

C++
C++ is lacking an exponentiation operator which makes a call to `std::pow` necessary. The inner product is an instance method of the `VectorXd` template class of the Eigen library.

```
e = ((std::pow(v_mag,2)-mu/r_mag)*r -
 ↪  r.dot(v)*v)/mu;
```

Java
In Java the mathematical expression is turned into a chain of method calls. As in C++ an exponentiation operator is not available. Code like this can be fluidly written with the support of an IDE but in our opinion it is harder to read and obscures the underlying mathematics.

```
e = r.scalarMultiply(Math.pow(v_mag,2) -
 ↪  mu/r_mag).subtract(r.dotProduct(v),
 ↪  v).scalarMultiply(1/mu);
```

Matlab
Matlab is foremost a convenient environment for vector- and matrix-based calculations. It does not come as a surprise that the calculation of eccentricity vector can be expressed through very concise Matlab code.

```
e = ((v_mag^2 - mu/r_mag)*r -
 ↪  dot(r,v)*v)/mu;
```

Python
Python uses the same exponentiation operator as Fortran. The inner product is implemented in the NumPy library which is by convention imported as `import numpy as np` and thereby made available through the `np` identifier.

```
e = ((v_mag**2 - mu/r_mag)*r - np.dot(r,
 ↪  v)*r)/mu
```

Julia
The ASCII version of the Julia code listed below is identical to the Matlab version. Julia also supports UTF-8 identifiers thus the greek letter μ can be used as a variable name for the gravitational parameter.

```
# ASCII version
e = ((v_mag^2 - mu/r_mag)*r -
 ↪  dot(r,v)*v)/mu
# UTF8 version
e = ((v_mag^2 - μ/r_mag)*r - (r·v)*v)/μ
```

3.2 Solving Kepler's Equation

In this example the task is to implement a generic Newton-Raphson solver which is then used to solve Kepler's equation. We test several properties of the candidate languages which we need within this example to express the problem as concisely as possible:

1. How does the language handle runtime errors? The solver shall report an error if the solution has not converged after a certain number iterations.

2. How can default arguments to functions be supplied? The user shall be able to control the convergence tolerance and the maximum number of iterations but shall also be able to use default values.

3. Can new functions be constructed ad-hoc? The Kepler equation and its derivative shall be passed as arguments to the generic solver. Since they do not only depend on the eccentric anomaly but also the mean anomaly and the eccentricity they need to be constructed on-the-fly by the Kepler solver and need to be able to access variables from the solver. In computer science terms these features are called higher-order functions (functions that create functions) and closures (functions are able to access variables from their enclosing scope).

Fortran
Fortran does not support higher-order functions or closures. An implementation that fulfills the requirements outlined above is therefore necessarily more involved. We have implemented the example by using Fortran 2003 features but the result is too verbose to list it here.

In production code it might make more sense to copy-paste the Newton-Raphson solver into the subroutine that solves the Kepler equation. Although this violates the "Do Not Repeat Yourself" rule [7]. Another possibility would be to extend the interface of the Newton-Raphson solver to accept an additional array of double-valued parameters.

Unlike all other examined languages Fortran does not provide exceptions or any other mechanism or convention for error handling. We use the `stop` statement in the example code to terminate the program when an error occurs. This is not possible in a production environment because the code calling the routine might be able to recover from the error and a single problem should not bring down the whole system. In real-world code a convention for signaling errors through parameters must be defined and enforced through coding standards.

Default values for parameters can be set by using the `optional` keyword in the subroutine signature and later checking for the presence of the value via the `present` intrinsic function.

C++
In C++ default values for parameters are defined directly in the function signature.

```
double newton(
        double p0,
        std::function<double(double)>
 ↪  const &func,
        std::function<double(double)>
 ↪  const &deriv,
        int maxiter = 50,
        double tol = 1e-8
) {
    for (auto i = 1; i < maxiter; i++) {
        auto p = p0 - func(p0) /
 ↪  deriv(p0);
        if (std::fabs(p - p0) < tol) {
            return p;
        }
        p0 = p;
    }
    throw runtime_error("Not
 ↪  converged.");
}
```

The C++11 standard introduced lambda expressions and closures. Through a lambda expression an anonymous function can be created with the following syntax:

```
[CLOSURE](SIGNATURE) -> RETURN_TYPE {
 ↪  FUNCTION_BODY}
```

It is a peculiarity of C++ that the variables contained in the closure must be explicitly defined. The C++ Kepler solver can then be expressed as shown below:

```
double mean2ecc(double M, double ecc) {
    auto E = newton(M,
    [ecc, M](double E) -> double {
        return E - ecc * sin(E) - M;
    },
    [ecc](double E) -> double {
        return 1 - ecc * cos(E);
    });
    return E;
}
```

Java
To supply default values for parameters in Java a method needs to be overloaded with another method which accepts fewer parameters. Therefore the method `getRoot` is defined twice in the Java class below.

```
public class Newton {
    public static double getRoot(
```

```
            double p0,
 ↪  Function<Double,Double> func,
 ↪  Function<Double,Double> deriv, int
 ↪  maxiter, double tol) {
        Double result = Double.NaN;
        for (int i=0; i < maxiter; i++)
 ↪  {
            double p = p0 -
 ↪  func.apply(p0) / deriv.apply(p0);
            if (Math.abs(p - p0) < tol)
 ↪  {
                result = p;
                break;
            }
            p0 = p;
        }
        if (result.isNaN()) {
            throw new
 ↪  RuntimeException("Not converged.");
        } else {
            return result;
        }
    }
    public static double getRoot(double
 ↪  x0, Function<Double,Double> func,
 ↪  Function<Double,Double> deriv) {
        return getRoot(x0, func, deriv,
 ↪  50, 1e-8);
    }
}
```

Java 8 also introduced lambda expressions and closures which are used in the class listed below. A Java 7 implementation which requires the use of an interface and an inner class is available in the online repository.

```
public class Kepler {
    public static double
 ↪  meanToEcc(double M, double ecc) {
        Function<Double,Double> keplerEq
 ↪  = E -> E - ecc * Math.sin(E) - M;
        Function<Double,Double>
 ↪  keplerDeriv = E -> 1 - ecc *
 ↪  Math.cos(E);
        return
 ↪  NewtonFunctional.getRoot(M,
 ↪  keplerEq, keplerDeriv);
    }
}
```

Matlab

In Matlab default values for parameters can be handled through defining a function with a varying number of parameters. The disadvantage of the implementation shown below is that not all combinations of input arguments are allowed, e.g. if the tolerance needs to be set manually the maximum number of iterations needs to be defined manually as well and cannot use the default value.

```
function p = newton(x0, func, deriv,
 ↪  varargin)
    switch nargin
        case 3
            maxiter = 50;
            tol = 1e-8;
        case 4
            maxiter = varargin{1};
            tol = 1e-8;
        case 5
            maxiter = varargin{1};
            tol = varargin{2};
    end

    p0 = x0;
    for ii = 1:maxiter
        p = p0 - func(p0)/deriv(p0);
        if abs(p - p0) < tol
            return
        end
        p0 = p;
    end
    error('Not converged.');
end
```

Matlab supports closures through the use of nested functions or the definition of an anonymous function through the `@(x) x^2;` syntax.

```
function E = mean2ecc(M, ecc)
    keplereq = @(x) x - ecc*sin(x) - M;
    keplerderiv = @(x) 1 - ecc*cos(x);
    E = newton(M, keplereq,
 ↪  keplerderiv);
end
```

Python

Like C++ Python defines default values for parameters in the function signature. Unique to Python is the fact that it uses indentation to delimit code blocks and therefore does not need an `end` statement or curly braces.

```
def newton(x0, func, derivative,
 ↪  maxiter=50, tol=1e-8):
    p0 = x0
    for _ in range(maxiter):
        p = p0 - func(p0)/derivative(p0)
        if np.abs(p - p0) < tol:
            return p
        p0 = p
    raise RuntimeError("Not converged.")
```

Higher-order functions and closures are supported in pure Python but JIT-compilation of this code with Numba is currently not possible.

```
def mean2ecc(M, ecc):
    def keplereq(E):
        return E - ecc*np.sin(E) - M
    def keplerderiv(E):
        return 1 - ecc*np.cos(E)
    return newton(M, keplereq,
 ↪  keplerderiv)
```

Julia

The Julia code is again very similar to the Matlab code. The first notable difference is that Julia does not define the return value in the function signature but rather through the `return RETURN_VALUE` statement. An important Julia feature are the type annotations in the function signature (e.g. `x0::Float64`). These help the JIT compiler to infer the types of the variables within the function and are used during method dispatch. Default values are defined in the function signature.

```
function newton(x0::Float64,
 ↪  func::Function,
 ↪  derivative::Function,
 ↪  maxiter::Int=50, tol::Float64=1e-8)

    p0 = x0
    for i = 1:maxiter
        p = p0 - func(p0)/derivative(p0)
        if abs(p - p0) < tol
            return p
        end
        p0 = p
    end
    error("Not converged.")
end
```

Julia supports higher-order functions and closures. A shorthand syntax for defining new functions in the form of `f(x) = x^2` is available.

```
function mean2ecc(M::Float64,
 ↪  ecc::Float64)
    kepler(E::Float64) = E - ecc*sin(E)
 ↪  - M
    kepler_der(E::Float64) = 1 -
 ↪  ecc*cos(E)
    return newton(M, kepler, kepler_der)
end
```

3.3 Solution of Lambert's Problem

In this example we have implemented the algorithm for solving Lambert's problem which was proposed by Bate, Mueller, and White [8] with adaptions by Vallado [9]. Apart from the differences in mathematical notation outlined above the implementations are very similar and the example serves mainly as a performance benchmark for iterative algorithms. Please refer to section 4 for the results.

3.4 Runge-Kutta Orbit Propagation

The final example tests how well legacy code written in Fortran 77 can be integrated into the candidate languages. We chose the DOP853 8th-order Runge-Kutta integrator with Dormand-Prince coefficients that was described by Hairer [10].

The example is quite complex since the function that represents the right-hand side of the differential equation, which is written in one of the other candidate languages, must be passed to and called by the Fortran code.

We use the code to solve Newton's equation for a uniform gravitational field.

Fortran

While it is obviously possible to directly call Fortran 77 code from modern Fortran it is advisable to define an explicit interface for the legacy code. This allows the compiler to check the type of the arguments and prevent errors that would otherwise cause memory corruption, e.g. by erroneously passing a scalar argument instead of an array argument.

We have also implemented a thin wrapper that uses the `ISO_C_BINDING` module to make the Fortran routine callable from C. This wrapper is also used by all other implementations. For C++ and Java a C/C++ header file is also required.

C++

Together with the Fortran 90 wrapper and the header file the DOP853 integrator can be called like any other C++ function. Since Fortran uses pass-by-reference pointer arguments need to be used when calling Fortran from C++.

Java

The Java version uses the Java Native Interface (JNI) to call the Fortran code through the wrapper. In this case 50+ additional lines of glue code in C that handle the conversion between Java and C/Fortran data types are required. These data type conversions also impose a performance penalty due to the necessary additional memory allocations (see section 4).

Matlab

Matlab can call C and Fortran code through its MEX interface. Like in the Java example significant amounts of glue code are required for data conversions. The development of MEX extensions requires great care because the environment

is rather unforgiving and programming errors can crash the Matlab IDE.

A MEX implementation could not be completed in time for the publication of this paper but the results will be published to the online repository at a later time.

Python

The reference implementation of Python is written in C thus a multitude of options for extending Python with native code exist. The Python extension could also not be completed in time and will be published online.

Julia

Similarly to C++ the DOP853 can be directly called from Julia via the `ccall` function.

One additional step is necessary to convert a Julia function reference to a C-compatible function pointer like shown below.

```
cfcn = cfunction(gravity!, Void,
 ↪  (Ptr{Cint}, Ptr{Cdouble},
 ↪  Ptr{Cdouble}, Ptr{Cdouble},
 ↪  Ptr{Cdouble}, Ptr{Cint}))
```

4 BENCHMARK

The benchmark was conducted on a machine equipped with an Intel Core i5-2557M CPU with 1.70GHz. The following software versions of compilers and runtime environments were used:

- Fortran: Intel Fortran 16.0.1 with flag `-O3`
- C++: Clang 7.0.2 with flag `-O3`
- Java: JDK 1.8.0-74
- Matlab: Release 2015a
- Python: 3.5 with Numba 0.23.1
- Julia: 0.5-dev

All code were executed and timed 100,000 times and the average runtime was compared to the Fortran version as shown in table 1 and figure 1. Please note that the y-axis is a log scale.

Fortran and C++ defend their reputation as high performance languages. It is notable that C++ seems to be significantly faster in the elements. We suspect that this might be a measurement error at the expense of Fortran because the resolution of Fortran's timing functions is not high enough. Java is generally the slowest of the compiled languages but still orders of magnitude faster than the interpreted languages Matlab and (pure) Python. The Julia language and just-in-time compiled Python with Numba provide excellent results.

Table 1. Average Runtime Relative to Fortran out of 100000 Runs

Problem	Elements	Kepler	Lambert	Dopri
C++	0.42	0.2	1.46	1.28
Java	7.54	0.7	3.2	19.22
Julia	2.29	0.71	1.1	2.97
Python	251.41	25.85	133.23	N/A
Python+Numba	5.57	N/A	1.56	N/A
Matlab	186.41	282.96	196.89	N/A

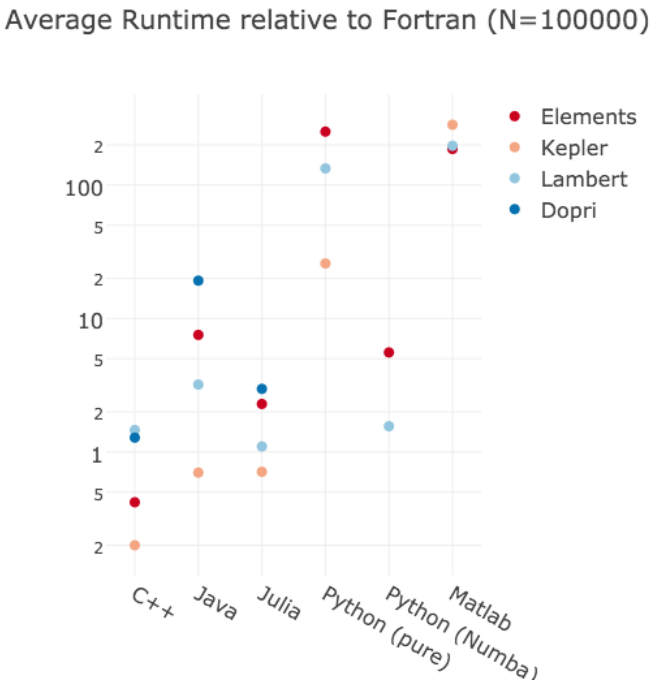

Fig. 1. Benchmark Results

With the slowest Julia example being a factor 3 slower and the slowest Python+Numba example being a factor 6 slower they offer a viable alternative if only very good performance and not the "best" performance is required.

5 CONCLUSION

The recent additions to classical compiled languages such as lambda expressions and closures in C++ and Java and object-oriented features in Fortran have greatly improved their expressivity and made them easier to work with. Dynamically interpreted languages on the other hand have offered these features for long time but were severly lacking in numerical performance. We have compared Fortran, C++, Java, Matlab, Python, and Julia through the implementation and benchmarking of astrodynamics problems. Statically compiled lan-

guages still offer best-in-class performance but to utilize it the user must deal with their inherent complexities such as setting up build systems. While purely interpreted languages such as Matlab and pure Python are still several orders of magnitude slower than compiled languages, JIT-compiled dynamic languages such as Python with Numba or the Julia language have reached a competitive level of performance while still offering the advantages of lower complexity and better programmer productivity.

6 References

[1] Lutz Prechtelt, "Two Comparisons of Programming Languages," in *Making Software: what really works, and why we believe it*, Andy Oram and Greg Wilson, Eds., pp. 239–258. O'Reilly, Sebastopol, California, 1st edition, 2011.

[2] Michael Metcalf, John Reid, and Malcolm Cohen, *Modern Fortran explained*, Numerical mathematics and scientific computation. Oxford University Press, Oxford, New York, 2011.

[3] Bjarne Stroustrup, *The C++ programming language*, Addison-Wesley, Upper Saddle River, NJ, fourth edition edition, 2013.

[4] Benjamin J. Evans, *Java in a Nutshell*, O'Reilly, Sebastopol, California, sixth edition, 2015.

[5] Nathan Marz and James Warren, *Big data: principles and best practices of scalable real-time data systems*, Manning, Shelter Island, New York, 2015.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah, "Julia: A Fresh Approach to Numerical Computing," Nov. 2014, arXiv: 1411.1607.

[7] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson, "Best Practices for Scientific Computing," *PLoS Biology*, vol. 12, no. 1, pp. e1001745, Jan. 2014.

[8] Roger R. Bate, Donald D. Mueller, and Jerry E. White, *Fundamentals of Astrodynamics*, Dover Publications, New York, 1971.

[9] David A. Vallado and Wayne D. McClain, *Fundamentals of Astrodynamics and Applications*, Microcosm Press, Hawthorne, Calif., 2013.

[10] E. Hairer, S. P. Nørsett, and Gerhard Wanner, *Solving ordinary differential equations I: nonstiff problems*, Number 8 in Springer series in computational mathematics. Springer, Heidelberg, London, 2nd rev. edition, 2009.