(Preprint) AAS 21-303

MODERN NUMERICAL PROGRAMMING WITH JULIA FOR ASTRODYNAMIC TRAJECTORY DESIGN

Dan Padilha*, Diogene A. Dei Tos†, Nicola Baresi‡, and Junichiro Kawaguchi†

A programming toolkit is developed to leverage Julia, a high-performance numerical programming language, in the generation, optimisation, and analysis of orbital trajectories. Julia combines high-level abstraction with the computational efficiency of dynamic compilation, enabling highly composable and extensible programs and state-of-the-art performance in differential systems, statistical analysis, and machine learning. This paper outlines the motivations and consequences of Julia's multiple dispatch, meta-programming, and other capabilities, and demonstrates techniques enabled for future development of astrodynamics toolkits. The resulting `OrbitalTrajectories.jl` toolkit's composability, extensibility, and performance is compared to JAXA's in-house *jTOP* trajectory propagation and optimisation tool, outperforming it by up to an order of magnitude in ephemeris-based restricted N-body trajectory propagation.

INTRODUCTION

Modern astrodynamics research exploits the chaotic dynamics of the solar system, through simulation and dynamical systems theory, to deliver lower-cost, higher-fidelity, and more ambitious spacecraft trajectories. The complexities involved in computational astrodynamics often require a two-language programming approach: a dynamic language (such as Python or MATLAB) provides a high-level abstraction to an underlying high-performance code (in a low-level static language such as C++ or Fortran), used for the computationally-intensive parts of a trajectory simulation or analysis. In practice, such an approach introduces numerous inefficiencies and complexities not only computationally, but also in the development, maintenance, and testing of new algorithms and tools.[1]

A more recent approach to avoid such issues is that taken by Julia, a modern, open-source, high-performance numerical programming language leading the state-of-the-art in differential systems, statistical analysis, and machine learning applications.[2–5] Julia avoids the two-language problem by combining the high-level semantics of dynamic languages with the speed and efficiency of Just-In-Time (JIT) compilation, providing high-performance without compromising on code maintainability and development efficiency. Its careful design lead inherently to highly composable, extensible, and performant algorithm designs.[6–8]

In this paper, we introduce `OrbitalTrajectories.jl`,* an astrodynamics toolkit for generating, optimising, and analysing spacecraft trajectories, and being developed as a demonstration of the Julia language and its potential benefits to computational astrodynamics software. We expand on prior related works, discussed in the next section, by motivating and applying some of Julia's inherent computational capabilities within the context of astrodynamic trajectory design. The preliminary development of our toolkit is described, in which we demonstrate several procedures with comparatively simple, composable, and immediately performant code implementations for the

*Department of Aeronautics and Astronautics, The University of Tokyo, Tokyo, Japan.
†Institute of Space and Astronautical Science (ISAS), JAXA, Sagamihara, Japan.
‡Surrey Space Centre (SSC), University of Surrey, Guildford, United Kingdom.
*Toolkit pending open-source release under a permissive license.

design and computation of astrodynamical models and their propagation functions, state transition matrices, and differential correctors for the generation of periodic orbits. Our toolkit's composability, extensibility, and computational performance, are briefly compared against JAXA's in-house *jTOP* trajectory propagation and optimisation tool.[9] Finally, we discuss current and potential future developments for Julia-based astrodynamics tools, and provide recommendations towards supporting more complex applications in space mission design and analysis.

Related Work

JPL were perhaps the first to identify the features of Julia likely to benefit future astrodynamics research (see Table 1 in the reference).[1] The authors demonstrated implementations for solving Lambert's problem, i.e. a time-of-flight solver, and for 2-body propagation; these were not only substantially simpler both in lines of code and in development time, but also were found to be only 1.7∼2.5 times slower than reference Fortran implementations. In the case of 2-body propagation, their performance was likely hampered by a custom implementation of an Ordinary Differential Equations (ODE) solver using the early `ODE.jl` package of the time, nowadays deprecated in favour of the best-in-class ODE solvers provided by `DifferentialEquations.jl`.[10] Despite this, the authors concluded that 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."

In a comparative study of programming languages for astrodynamics systems, Julia was found to be a "viable alternative to less productive compiled languages" thanks to its features, performance, and ease-of-use.[11] Here, the authors implemented and demonstrated 4 problems: calculating Keplerian elements from a Cartesian state vector; solving Kepler's equation; solving Lambert's problem; and 2-body Runge-Kutta orbit propagation. In every problem, Julia outperformed MATLAB, Java, and JIT-compiled Python by up to 1∼2 orders of magnitude, and approximately matched the performance of C++ and Fortran. In addition, the Julia implementations also required the fewest lines of code, fewer even than Python or MATLAB.

Perhaps the most comprehensive astrodynamics package available in Julia is `SatelliteToolbox.jl` from the Brazilian National Institute for Space Research (INPE), which drives the simulation core of the institute's FOrPlan Satellite Simulator and has been used for mission analysis.[12,13] This package is primarily focused on near-Earth applications for satellites, providing Earth atmospheric and geomagnetic models, two-body propagators with J2/J4/SGP4 perturbations (utilising the ODE solvers of `DifferentialEquations.jl`), and reference frame converters. It is provided alongside `Astrodynamics.jl` (an in-development trajectory design toolkit), as well as Julia bindings for GMAT and Orekit.*

Finally, in the more general field of astronomical and astrophysical computation, several relevant examples exist. An important example is that of the `Celeste` project for astronomical imaging surveys, developed by UC Berkeley together with Julia Computing and Intel, which achieved "the largest-scale application of Bayesian inference reported to date" and concluded that Julia is suitable for high-performance computing applications.[2] More recently, JPL and Embry-Riddle Aeronautical University described a Julia-based ephemeris reader including support for gravitational models of small bodies and asteroids.[14] Similarly, the independently developed `SPICE.jl` provides a nearly complete Julia interface to the `C` implementation of the NAIF `SPICE` Toolkit,[15] and is used by our implementation of `OrbitalTrajectories.jl`. Separately, `JPLEphemeris.jl` provides a pure-Julia ephemeris reader for computing the position and velocities of solar system bodies.†

Our toolkit differs from the above examples by implementing 4 separate propagation models with increasing fidelity, making use of automatic code generation via symbolic computation for high-performance propagation, and demonstrating the use of meta-programming and auto-differentiation

*`SatelliteToolkbox.jl` and `Astrodynamics.jl` are maintained by JuliaSpace (`https://juliaastrodynamics.github.io/`).
†Both `SPICE.jl` and `JPLEphemeris.jl` are maintained by the JuliaAstro organisation (`https://juliaastro.github.io/`).

features of Julia to determine orbit stabilities and generate periodic orbits. In addition, we utilise version 1.5.3 of Julia, released in Nov 2020, and its up-to-date package ecosystem, providing functionality not available to previous authors. Similarly to `SatelliteToolbox.jl` and `Astrodynamics.jl`, we demonstrate integration with the `SPICE` Toolkit, including reading ephemerides and performing dynamic reference frame conversions.

DESIGN OF MODERN NUMERICAL PROGRAMMING LANGUAGES

We begin by introducing two key problems inherent in numerical programming languages, which motivate the design philosophies behind modern languages. The consequences of several design choices in the Julia language are explained – namely of multiple dispatch, Just-In-Time (JIT) compilation, and meta-programming – and later demonstrated in the context of astrodynamics models.

The Two-Language Problem

A common design pattern in scientific research is to use a dynamic numerical language (such as Python or MATLAB) to define the high-level abstractions of an analysis, and combine them with a static or compiled language (such as C or Fortran) to define the performance-critical lower-level components. In combining two languages, researchers gain the best of both worlds: greater productivity from dynamic abstractions, which help to more easily reason about and iteratively develop algorithms; and greater performance from lower-level languages, which tend to be more complicated but provide much finer control of the underlying computational hardware. However, as most practitioners would likely attest to, such an approach has many non-trivial costs, including: the increased complexity and development costs of a coupled design; the separate environments requiring disparate syntax, tools, and development processes; and the non-trivial overheads of wrapping and binding data between high- and low-level components. This trade-off in language specialisations is known as the Two-Language Problem (see Figure 1), and having a single language that can do both roles simultaneously was a key motivation for the design of the Julia language.[6]

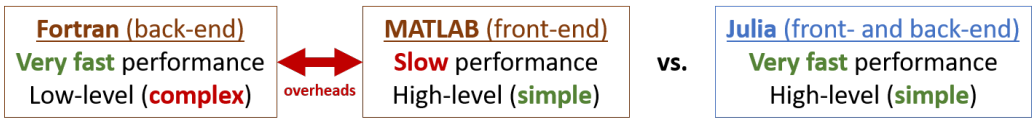

Figure 1: Example of the Two-Language Problem, arising from the trade-off in specialisations between low- and high-level programming languages. The Julia language is specifically designed to avoid this.

The jTOP trajectory optimisation and mission design tool in use at JAXA is a prime example of this, being implemented primarily in MATLAB with propagation of equations of motion performed by a Fortran 90 library.[9] Other examples include those of ULTIMAT (an ultra-low thrust mission analysis tool in MATLAB and C-`MEX`) and DIRETTO (a spacecraft optimal control tool in MATLAB and C++).[16] While the two-language approach helps greatly improve the performance of a tool, its development suffers qualitatively by being more difficult to modify or improve, as well as potential issues due to proprietary or closed-source components (as in MATLAB's standard library and toolboxes, third-party solvers and optimisers, and so on).

Julia avoids the Two-Language Problem by providing high-level syntax and dynamic behaviour similar to Python/MATLAB, and JIT compilation for all Julia functions to provide high-performance compiled execution specialised to the data types in use at run-time.[6] The primary drawback of Julia's design is known colloquially as the "time to first plot" problem,* in which each Julia function must be compiled the first time it is called with any new combination of types. This also means that the Julia run-time is relatively costly and may not be suitable in low-power embedded devices, although both issues may be somewhat mitigated by careful pre-compilation of Julia programs.

*Although it applies more generally than just to plotting.

We note that even in cases where other dynamic languages achieve high performance, they are likely taking a two-language approach to do so. For example, MATLAB scripts can be augmented with `MEX` files – a subset of MATLAB code that is compiled from C/C++/Fortran – however "[the] development of MEX extensions requires great care, because the environment is rather unforgiving and programming errors can crash the Matlab IDE."[11] Similarly, Cython performs static compilation of Python code into C, at the cost of a mixed syntax with explicit type annotations and limited speed-up for native Python structures, while Numba offers JIT compilation at the cost of supporting only a subset of Python code. Popular scientific packages such as Numpy and Tensorflow consist of pre-compiled C/C++ cores, making the lower-level components more difficult to understand and modify for the average user. Although any of these approaches can be highly productive for the modern researcher, they are akin to bolting on performance improvements, rather than addressing the underlying design principles of the languages themselves.

The Expression Problem

From a mathematical perspective, the purpose of code is simply to perform operations on data. In this sense, functions (or methods) are the building blocks that define operations from input to output data types. When maintaining or improving some piece of code such as a library or package, one would seek to extend along two possible directions: adding new functions/methods, new data types, or both. The Expression Problem states that it is difficult to easily support extensions across both directions simultaneously, and is a common drawback of traditional Object-Oriented Programming (OOP) languages (such as Python, C++, or Java).[17]

The practical consequences of the Expression Problem are difficult to grasp without a more concrete motivating example. Suppose that we wish to represent different types of orbits in code, such as `CircularOrbit` and `EllipticOrbit`, each deriving from some abstract supertype `Orbit`, and that we require a function to compute a `transferBetweenOrbits(`A`, `B`)` from `Orbit` A to B. In a traditional OOP language this could be implemented as a superclass `Orbit` with a default method `transferBetweenOrbits(`$self$`, `$other$`)`; each subclass `CircularOrbit` and `EllipticOrbit` would inherit `Orbit`'s default behaviour, and perhaps overload its `transferBetweenOrbits` method to something more appropriate. A user of this `OrbitsPackage` could create their own orbit types, say a `HyperbolicOrbit`, by simply inheriting behaviours from these existing `Orbit` types – for example, as a subclass of the `EllipticOrbit` with an eccentricity $e > 1$.* However, modifying the existing transfer functions to add specific support for `HyperbolicOrbit`s is much more difficult; either the `OrbitsPackage` developer has to pre-empt and design for such a situation, or the user must modify the package source manually or via dynamic code introspection.

Julia solves the Expression Problem by utilising the concept of multiple dispatch.[6] Each function in Julia is implemented by multiple individual methods, each defined based on the combination of types used for its input arguments; an illustrated example of the `OrbitsPackage` in Figure 2 shows how the `transferBetweenOrbits()` function defines separate transfer methods depending on the orbits involved. In essence, multiple dispatch selects the most appropriate (and, thanks to its JIT compilation, the most performant) method for each given set of inputs,† and allows for easily adding either new types (extending existing types), or new methods (extending existing functions). In practice, this inherently encourages a more functional approach to programming that avoids hard-coded implementation-specific details, leading to highly extensible and composable packages based on more generic data types and functionality.[6,7]

*In Julia, a concrete type cannot inherit from (be a subclass of) another concrete type. Although this may feel like a significant limitation, it results in an inherent preference for composition-over-inheritance, a commonly recommended design pattern in OOP programming to avoid the aptly-named Circle-Ellipse Problem.[8]

†It is important to note that multiple dispatch differs from function overloading (as present in C++ and Java), in that it dispatches on run-time dynamic types of the inputs rather than compile-time static types.

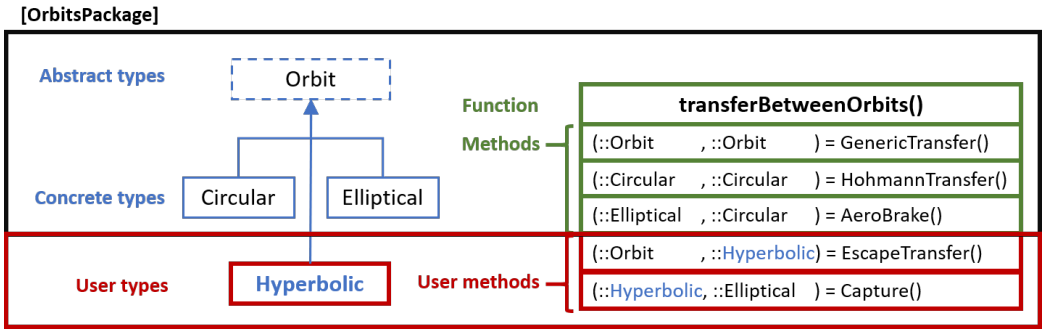

Figure 2: An illustrative example of Julia's approach to solving the Expression Problem. Multiple dispatch allows multiple individual methods to be uniquely defined for a function based on the combination of types used in their arguments. This makes it simple for a user to extend a package both with new types and new methods.

Meta-Programming & Code Generation

Finally, we briefly introduce the concept of meta-programming, which refers to code that can reason about and manipulate other code. Julia code is itself represented in tree-like data structures of `Expr` types, and can be manipulated using methods defined either generically or specifically for code expressions. In practice, this means that Julia code can modify or generate more Julia code, including intermediate representations used by the compiler, to provide functionality such as automatic compilation to GPU devices[18] or automatic differentiation of any pure-Julia function.[3]

COMPILED SYMBOLIC ASTRODYNAMICAL MODELS

This section outlines how the previously described features of Julia help to simplify the development of composable dynamical models for astrodynamics systems, utilising only the expressions of their equations of motion in their canonical literature forms. Our `OrbitalTrajectories.jl` toolkit demonstrates Julia's powerful code generation capabilities (meta-programming) by making use of `ModelingToolkit.jl` symbolic expression tools, transforming symbolic equations of motion to: build 1st-order propagation functions for use with ODE solvers; optimise memory allocations for in-place, out-of-place, or sparse arrays; and even compute the Jacobian of the dynamical system. From these, the JIT compiler generates highly-specialised functions, leaving the developer with easy-to-maintain and mathematically-simple code, and high-performance compiled implementations of each model.

Elliptic Restricted 3-Body Problem (ER3BP) Model

The 2nd-order differential equations of motion for the ER3BP model[19] in Equation (1) can be expressed succinctly in Julia code as in Listing 1:*

$$\begin{aligned} x'' &= 2y' + \omega_{/x}, \qquad y'' = -2x' + \omega_{/y}, \qquad z'' = \omega_{/z}. \\ \omega &= \frac{\Omega^{(3)} - \frac{1}{2}z^2 e\cos f}{1 + e\cos f} \qquad \Omega^{(3)} = \frac{1}{2}(x^2+y^2) + \frac{1-\mu}{r_1} + \frac{\mu}{r_2} + \frac{1}{2}\mu(1-\mu) \end{aligned} \tag{1}$$

where ω and $\Omega^{(3)}$ are the elliptic and circular potential functions respectively and dependent on the normalised position $(x, y, z)^T$, distance to the primary and secondary body r_1 and r_2 respectively, true anomaly f, eccentricity e, and mass fraction μ of the system.

Lines 1 and 2 of Listing 1 define an `_ER3BP` concrete sub-type of an abstract `AstrodynamicalModel` type. Lines 3 onwards define a constructor for an Ordinary Differential Equation (ODE) system

*Note that snippets of Julia code in this paper are intended to illustrate the ideas presented, and may differ slightly from actual implementations, in particular due to the rapidly evolving Julia ecosystem.

```
1  abstract type AstrodynamicalModel end     # Super-type of all astrodynamical models
2  struct _ER3BP <: AstrodynamicalModel end  # Concrete representation of ER3BP equations
3  function ModelingToolkit.ODESystem(::Type{_ER3BP})
4      @parameters  μ e f                    # Mass fraction, eccentricity, true anomaly
5      @variables   x(f) y(f) z(f)           # Position variables as functions of true anomaly
6      @derivatives D'~f D2''~f              # Derivatives with respect to true anomaly
7      @derivatives Dx'~x Dy'~y Dz'~z        # Derivatives with respect to position
8      _ω = ω(μ, e, f, (x, y, z))            # Call math functions using symbolic variables
9      return ODESystem([
10             D2(x) ~ +2D(y) + Dx(_ω),
11             D2(y) ~ -2D(x) + Dy(_ω),      # Equations of motion (as per Equation (1))
12             D2(z) ~        + Dz(_ω)
13         ], f, [x, y, z], [μ, e])          # Independent variable, state variables, parameters
14 end
15 r(μ, pos)             = @. norm((pos - [-μ, 0., 0.], pos - [1 - μ, 0., 0.]))
16 Ω_3(μ, (x,y,z))       = 0.5*(x^2+y^2) + sum(((1-μ), μ) ./ r(μ,(x,y,z))) + 0.5*μ*(1-μ)
17 ω(μ, e, f, (x, y, z)) = (Ω_3(μ,(x,y,z)) - 0.5*z^2*e*cos(f)) / (1 + e*cos(f))
```

Listing 1: Example of Julia code for trajectory propagation in the developed `OrbitalTrajectories.jl` toolkit. The equations of motion for the ER3BP model are defined as symbolic expressions that can be manipulated directly.

(`ODESystem`) type, dispatched on the `_ER3BP` type. This is a simple example of Julia's solution to the expression problem, in which we may easily extend the functionality of the `ModelingToolkit.jl` library's existing `ODESystem` type with our own methods and types.

Lines 15 to 17 are ordinary Julia functions defining the mathematical expressions for (r_1, r_2), $\Omega^{(3)}$, and ω respectively for any arbitrary input types. Each mathematical operation in Julia is defined as a function in the `Base` library, including addition `Base.:(+)(a, b)`, multiplication `Base.:(*)(a, b)`, norms `LinearAlgebra.norm(x)`, and trigonometric functions `Base.cos(x)`. Multiple dispatch allows for the `ModelingToolkit.jl` library to extend the functionality of these to its own symbolic data types, thus defining a symbolic expression algebra.* Lines 4 to 7 use macros – meta-programming functions that operate on Julia expressions to generate code – to define parameters, variables, and derivatives as symbolic data types. These symbolic types are thus traced through the generic mathematical expressions as input arguments of the ω function on Line 8. With this, the resulting `ODESystem` (Lines 9 to 13) contains a set of symbolic expressions representing exactly the equations of Equation (1).

Generic Trajectory Propagation Functions

The `ODESystem` of a model can be passed to a generic `AstrodynamicalModel` type constructor in Listing 2, which uses the expression manipulation functions of `expand_derivatives` and `ode_order_lowering` from `ModelingToolkit.jl` to expand and lower the system to 1^{st}-order equations respectively. The system is then built into an `ODEFunction` object in Line 7, using meta-programming to generate Julia code (as a structured tree of `Expr` objects) that defines propagation functions for the equations of motion of the ER3BP system. For example, the in-place propagation function (stored as literal code in the `ODEFunction`'s `f.f_iip.body` property) accepts an initial state, system parameters, and timestep, and modifies a cached output array in-place with the 1^{st}-order differential values for that timestep. These functions can be built dynamically at run-time, or as part of the `OrbitalTrajectories.jl` package pre-compilation process.

The `ODEFunction` object can be supplied together with a desired ODE solver to the generic `solve` function provided by the `DifferentialEquations.jl` library,[10] as in Line 4 of Listing 3. Julia's dynamic JIT compilation automatically specialises the call to `solve` to the ODE solver and to the propagation

*The authors have been informed that a paper on `ModelingToolkit.jl` is in preparation as of Jan. 2021.

```julia
function (T::Type{<:AstrodynamicalModel})(args...; kwargs...)
    # Expand and lower model to a 1st-order system of differential equations
    _ODE        = ODESystem(T, args...; kwargs...)
    _eqs        = expand_derivatives.(equations(_ODE))
    eqs, states = ode_order_lowering(_eqs, independent_variable(_ODE), states(_ODE))
    ODE         = ODESystem(simplify.(eqs), independent_variable(_ODE), states, parameters(_ODE))
    return ODEFunction(ODE)  # Generate integration functions automatically
end
```

Listing 2: Example Julia code for constructing any arbitrary astrodynamical model, allowing the Julia JIT compiler to automatically generate specialised, performant versions of its propagation function directly from high-order differential equations.

functions,* leading to high-performance compiled integration of the differential equations for the model. An example of the resulting trajectories for several models, as plotted by Line 5 (a function from the `Plots.jl` library extended using a custom plotting recipe), can be seen in Figure 3. The trajectories computed by ODE solvers from `DifferentialEquations.jl` automatically include high-order interpolation and, if desired, also support generic numerical types including arbitrarily-precise numbers, distributed arrays, and physical units of measurement.[10]

```julia
state = State(ER3BP("Earth", "Moon"),          # State wrapper for pre-compiled model of ODE system
    [0.76710535, 0., 0., 0., 0.47262724, 0.],  # Normalised initial state (position, velocity)
    (1.05π, 3π))                               # Propagation timespan (e.g. true anomaly f)
trajectory = solve(state)                      # Propagate in Vern7 ODE solver, 1 × 10^-10 tolerance
plot(trajectory, RotatingFrame())              # Plot in a normalised rotating frame
```

Listing 3: Example Julia code for propagating a spacecraft trajectory. The `State` type wraps an astrodynamical model, in this case `ER3BP`, an initial spacecraft state (position and velocity), and an integration timespan. A dispatched call to `DifferentialEquations.solve(::State)` solves the initial value ODE problem, returning the propagated spacecraft state trajectory.

Composable Model Derivation: Circular Restricted 3-Body Problem (CR3BP)

The CR3BP model can be defined by simply specifying an ER3BP model with an eccentricity of $e = 0$, as per Listing 4. Again, the resulting `ODESystem` can be passed directly to the generic constructor in Listing 2. Thanks to meta-programming and the JIT compiler, the computations are automatically simplified, and the resulting propagation functions compiled into highly performant versions of the CR3BP model. In this same way, we envision future versions of `OrbitalTrajectories.jl` being able to compose dynamical models together, such as adding custom perturbations dynamically.

```julia
struct _CR3BP <: AstrodynamicalModel end
function ModelingToolkit.ODESystem(::Type{_CR3BP})
    ER3BP = ODESystem(_ER3BP)   # Build directly from the ER3BP equations (setting eccentricity => 0)
    (μ, e) = parameters(ER3BP)
    eqs = [eq.lhs ~ simplify(substitute(eq.rhs, e => 0)) for eq in equations(ER3BP)]
    return ODESystem(eqs, independent_variable(ER3BP), states(ER3BP), [μ])
end
```

Listing 4: Symbolic ODE system defining the equations of motion for the CR3BP model by simply composing an ER3BP model and substituting into it an eccentricity value of $e = 0$.

*The `ODEFunction` type is parametrised by the types of its propagation functions, and therefore the compiler can compile specifically high-performance code for any given `ODEFunction`.

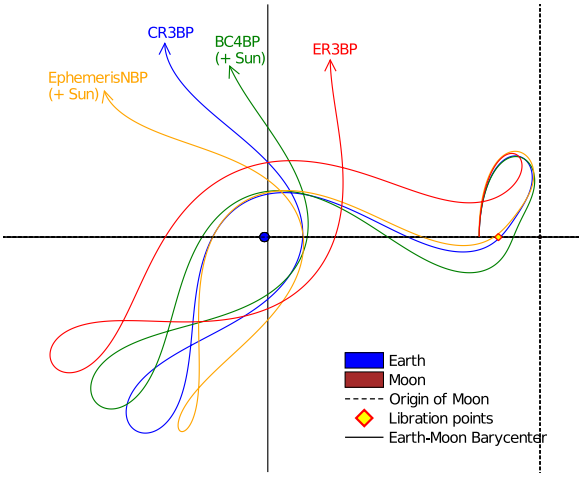

Figure 3: Example trajectories in the Earth-Moon(-Sun) system, computed using propagation functions generated automatically from symbolic astrodynamical models, and plotted in a normalised rotating frame, as per Listing 3 for the respective models.

Bi-Circular Restricted 4-Body Problem (BC4BP) Model

The 2nd-order differential equations of motion and code for the BC4BP model are omitted here, however their code largely resembles that of the ER3BP model defined in Listing 1. The code is greatly simplified by using Julia `Vector`s for the spacecraft position $\mathbf{p}$, third-body (Sun) position $\mathbf{p}_S$, and unit vector $\mathbf{r}$, transcribing precisely the form of Equation (21) of (Reference 19), and expressing the resulting system as `ODESystem(@.(D2(p) ~ RHS), ...)`, using the broadcasting macro `@.` to automatically distribute the right-hand-side terms across the spacecraft acceleration vector $\mathbf{p}''$.

Ephemeris Restricted N-Body Problem (EphemerisNBP) Model

The higher-fidelity EphemerisNBP model is constructed from a restricted set of N-body forces, in which each body's position is computed directly from planetary ephemerides. The 2nd-order differential equations of motion for the EphemerisNBP model[20] in Equation (2) are expressed in Julia code as in Listing 5.

$$\mathbf{p}'' = -\mu_1 \frac{\mathbf{p}}{||\mathbf{p}||^3} + \sum_{j=2}^{N} \mu_j \left(\frac{\mathbf{p}_j - \mathbf{p}}{||\mathbf{p}_j - \mathbf{p}||^3} - \frac{\mathbf{p}_j}{||\mathbf{p}_j||^3} \right) \tag{2}$$

where $\mathbf{p} = (x, y, z)^T$ is the inertial position of the spacecraft relative to a chosen central body, $\mu_j = Gm_j$ is the jth body's mass parameter (where G is the universal constant of gravitation, m_j is the mass of body j, and $j = 1$ refers to the central body), and $\mathbf{p}_j = (x, y, z)_j^T$ is the instantaneous position of the jth body relative to the central body (with $\mathbf{p}_1 = (0, 0, 0)$) as given by the JPL ephemeris data from DE430 (computed by the `SPICE.jl` wrapper to the C SPICE Toolkit).

Again, the `_EphemerisNBP` type is constructed using the generic constructor of Listing 2. However, in this case, the equations of motion contain a call to the C SPICE Toolkit's `spkpos` function (to compute the position of a target body relative to some reference).[15] As Julia can not trace symbolic variables through a call to an externally compiled C function, we must register it with `ModelingToolkit.jl` as a fundamentally symbolic expression in Lines 1 to 3 of Listing 5. This leads to two unfortunate side-effects. Firstly, the symbolic expressions do not know that `p_j_term` is a vector; this is fixed by explicitly rewriting `p_j` as a vector of its terms. Secondly, they also do not know that the `spkpos` function is mathematically pure (always returns the same result for the same inputs); this leads to a significant performance penalty as SPICE is called each time the `p_j_term` appears in the

```julia
get_pos(tgt, t, ref) = SPICE.spkpos(tgt, t, "ECLIPJ2000", "none", ref)[1]
get_pos(tgt, t::ModelingToolkit.Num, ref) = get_pos(Num(tgt), t, Num(ref))
ModelingToolkit.@register get_pos(tgt, t, ref)            # Force `spkpos` to be symbolic
struct _EphemerisNBP <: AstrodynamicalModel end
function ModelingToolkit.ODESystem(::Type{_EphemerisNBP}, center, bodies, μ_list)
    @parameters  t                                        # Time parameter t (in J2000 epoch)
    @variables   x(t) y(t) z(t)
    @derivatives D2''~t
    accelerations = zeros(Num, 3)                         # Initialise symbolic expressions
    p = [x, y, z]                                         # Position of spacecraft
    for (body, μ_j) in zip(bodies, μ_list)
        if body == center
            p_j = [0., 0., 0.]                            # Position of central body
        else
            p_j_term = get_pos(body, t, center)           # Pos. of body j at time t rel. to center
            p_j = [p_j_term[1], p_j_term[2], p_j_term[3]] # Explicitly-sized position vector
            accelerations -= μ_j * p_j / norm(p_j)^3
        end
        accelerations  += μ_j * (p_j - p) / norm(p_j - p)^3
    end
    return ODESystem(@.(D2(p) ~ sum(accelerations)), t, p, [])
end
```

Listing 5: Symbolic ODE system defining the equations of motion for the EphemerisNBP model.

resulting expressions, even though it need only be called once per body per integration timestep. As of this writing, it is not possible to inform `ModelingToolkit.jl` to treat functions as pure,* but Julia's meta-programming capabilities allow for interception of the generated code and injection of an intermediate computation of `spkpos` into an intermediate variable, substituting each symbolic use of the `p_j_term` with its result. In assessing the performance of the EphemerisNBP in later sections, we compare both the case with and without these precomputed body positions.

GENERIC SENSITIVITIES & DIFFERENTIAL CORRECTION

The applications of the Julia features discussed so far extend beyond those of simplifying the development of astrodynamical models. In this section, we explain how Julia's capabilities lead to a fully differentiable language, and demonstrate how this can be used in practice to automatically generate derivatives of astrodynamical models for uses such as generating (quasi-)periodic orbits.

Automatic Differentiation (AD)

Automatic Differentiation (AD) is a set of computational techniques to automatically compute derivatives of numerical functions for use in many important scientific applications, including optimisation and correction procedures and various machine learning algorithms.[3] AD differs from the method of Finite Differences (FD) in that it does not result in truncation and round-off errors from approximations, and does not require as many function evaluations to compute derivatives. AD also differs from hand-coded analytical derivatives by being automatic (less time-consuming and less prone to programming errors), and from symbolic differentiation by being simpler and more computationally efficient.[21] The primary drawback of AD is that its generic implementation can be quite complicated; fortunately, due largely to Julia's dynamic multiple dispatch and meta-programming capabilities, several packages exist to provide almost universal differentiation of any Julia code.[22]

Of the AD implementations that exist in Julia, we briefly describe the simpler forward-mode method implemented by the `ForwardDiff.jl` package. The package introduces a `Dual{`N`,`T`}` numerical

*The maintainers of the `ModelingToolkit.jl` package are aware of and working on improvements for this.

type, a structure containing an underlying numerical value of type T (which can itself be a `Dual` number, thus supporting higher-order derivatives), and N numbers representing its orthogonal partial derivatives. Using multiple dispatch and meta-programming, fundamental numerical functions can be defined for an arbitrary `Dual` number, allowing them to be used in place of any existing numerical values. These functions simultaneously compute both the value and its derivative and update the `Dual` number, allowing the partial derivatives to propagate through the computation stack. For example, taking the sine of a `Dual` number dispatches to the `sin(x::Dual)` method which returns a `Dual(sin(x.value), cos(x.value) * x.partials)`. Unlike AD implementations in other high-level languages such as Python and MATLAB, the Julia JIT compiler specialises all code to its parameterised `Dual` types, resulting in highly performant evaluations of a function and its associated derivatives, at the cost of additional memory usage and linear scaling of computations.[22]

Computing State Transition Matrices (STMs)

In astrodynamics, one common use of derivatives of dynamical models is in computing the State Transition Matrix (STM), a linear mapping of the state sensitivities of the system.[23] There are broadly four main ways to compute an STM: by variational equations (analytical or symbolic); or by differencing (Finite (FD) or Automatic Differentiation (AD)).

Variational Equations The STM Φ of a 1^{st}-order system of N equations of motion f (a function of the state X) is found by integrating the differential system augmented with N^2 variational equations (elements of the Jacobian ∇f of the model), as per Equations (3) and (4).[24,25]

$$\dot{X} = f(X) \tag{3}$$

$$\dot{\Phi}(t, t_0) = \nabla f(X_t)\Phi(t, t_0) \tag{4}$$

where the STM's initial state is the identity matrix $\Phi(t_0, t_0) = \mathbb{1}$, and the Jacobian ∇f is computed for some state X_t with respect to the initial state X_{t_0}.

The Jacobian ∇f of an astrodynamical model's equations of motion f can be computed analytically or found through symbolic manipulation. In Julia, we can define a generic method to build the augmented differential system using a Jacobian computed automatically by tracing symbolic variables through the equations of motion, and a symbolic expression of the STM differential system of Equation (4) (Lines 5 and 6 of Listing 6). In this way, Julia effectively automates the process from derivation of variational equations through to their propagation functions, eliminating the need for any manual steps, additional code, or external tools (such as symbolic manipulation in Maple[25]).

```
1  function AugmentedODEFunction(ODE::ModelingToolkit.AbstractODESystem)
2      iv, dvs, params = independent_variable(ODE), states(ODE), parameters(ODE)
3      @variables   Φ[1:length(dvs), 1:length(dvs)](iv)          # State Transition Matrix (N*N)
4      @derivatives D'~iv                                        # Derivative w.r.t. independent var.
5      ∇f     = ODEFunction(ODE; jac=true).jac(dvs, params, iv)  # Build f's Jacobian
6      vareqs = simplify.(D.(Φ) .~ ∇f * Φ)                      # Equation (4)
7      augmented_ODE = ODESystem([equations(ODE)..., vareqs...], iv, [dvs..., Φ...], params)
8      return ODEFunction(augmented_ODE)  # Augment system with vareqns and build integration functions
9  end
```

Listing 6: Example of a generic function for augmenting an arbitrary ODE system with its variational equations, allowing the State Transition Matrix (STM) to be computed simultaneously.

Finite/Automatic Differentiation It is also possible to perform numerical differentiation by Finite (FD) or Automatic Differentiation (AD). In Listing 7, the `DiffEqSensitivity.seed_duals` function on Line 2 seeds a given initial state vector `state.u0` with `Dual` type numbers. Line 3 then rebuilds the `State` wrapper with the seeded initial state vector `u0` and solves the resulting initial value problem, returning the full trajectory together with its computed partial derivatives.

This function exploits the computational efficiency and precision of AD and the differentiability of Julia code[22] to compute the STM at every state, without manually or symbolically specifying variational equations. It is therefore agnostic to the astrodynamical model in use, accepting any of the four models defined previously, wrapped by a `State` type. It is trivial to manually pull back the `Dual` partial AD derivatives to recover a trace of the STM along its fully integrated and interpolatable trajectory, or otherwise to use `ForwardDiff.jacobian` or `FiniteDiff.finite_difference_jacobian` to directly extract an AD or FD Jacobian. It is likewise simple to compute sensitivities with respect to model parameters (by also seeding them with `Dual`), such as the μ parameter of the CR3BP model, or to add intermediate steps such as converting the resulting trajectories to other reference frames.

```julia
function STM_trace(state::State; kwargs...)  # Seed State with Duals for automatic differentiation
    u0 = DiffEqSensitivity.seed_duals(state.u0, typeof(state.model))
    solve(remake(state; u0); kwargs...)      # Propagate trajectory together with its STM
end
```

Listing 7: Generic state sensitivity function providing automatic propagation of trajectories together with their STMs for any differentiable astrodynamical model.

STM Accuracy & Performance

Figure 4 shows the result of tracing the maximum eigenvalue $\lambda_{\max}$ of the STMs propagated using both symbolically-generated Variational Equations (VE) and Automatic Differentiation (AD) for the three trajectory test cases defined by (Reference 25). The test cases' CR3BP initial state is mapped to the ER3BP, BC4BP, and EphemerisNBP models using arbitrarily-chosen initial true anomaly f_0, third-body (Sun) phase angle α_0, and epoch time t_0 parameters respectively. For the sake of comparison, the BC4BP and EphemerisNBP models also include the gravitational force of the Sun. In every model, the trajectories and STM traces for both VE and AD match within relative error tolerances, and as such appear as a single line in the figure. In addition, for the CR3BP model, manually hand-coded equations of motion and variational equations are also compared, and again match both the VE and AD trajectories.

Note that the EphemerisNBP's STM trace matches the behaviour of the other models despite being propagated in an inertial frame (as per Equation (2)). This is because the EphemerisNBP trajectory is converted to a normalised rotating frame using an instantaneous state transformation matrix computed by the C SPICE Toolkit's `sxform` function and using a SPICE dynamic frame built as-needed by `OrbitalTrajectories.jl` at run-time. This reference frame conversion is a simple matrix multiplication with the state vector, and so automatically transforms the AD-computed STM values appropriately by cross-multiplying the partial derivatives stored in the state's `Dual` values. By comparison, the VE version of EphemerisNBP requires a custom state transformation function to properly cross-multiply the partial derivatives. The difference between these two approaches can be seen in Listing 8, further highlighting the benefits of multiple dispatch in separating functionality.

As expected, the astrodynamical models display distinctive yet similar behaviours in their STM's $\lambda_{\max}$ traces. The $\lambda_{\max}$ values for the CR3BP models, which act as a proxy measure of the sensitivity of each orbit, match the reference values given by (Reference 25) to within the relative tolerances specified in Table 1. The FD models were found to be the least accurate in computing $\lambda_{\max}$, especially in the particularly sensitive "PO2" case with multiple close fly-bys of the secondary body. The AD and VE models are the most precise for these cases, and their difference may be due to the method used to compute the reference values, or due to their solver parameters and integration paths.[25]

A representative computational performance comparison between the models and differentiation methods is shown in Table 2, averaged across each model and method from up to $10,000$ individual runs of each of the 3 periodic orbit test cases above, with computational times normalised within each case. These show that the manually optimised, hand-coded variants of trajectory and STM

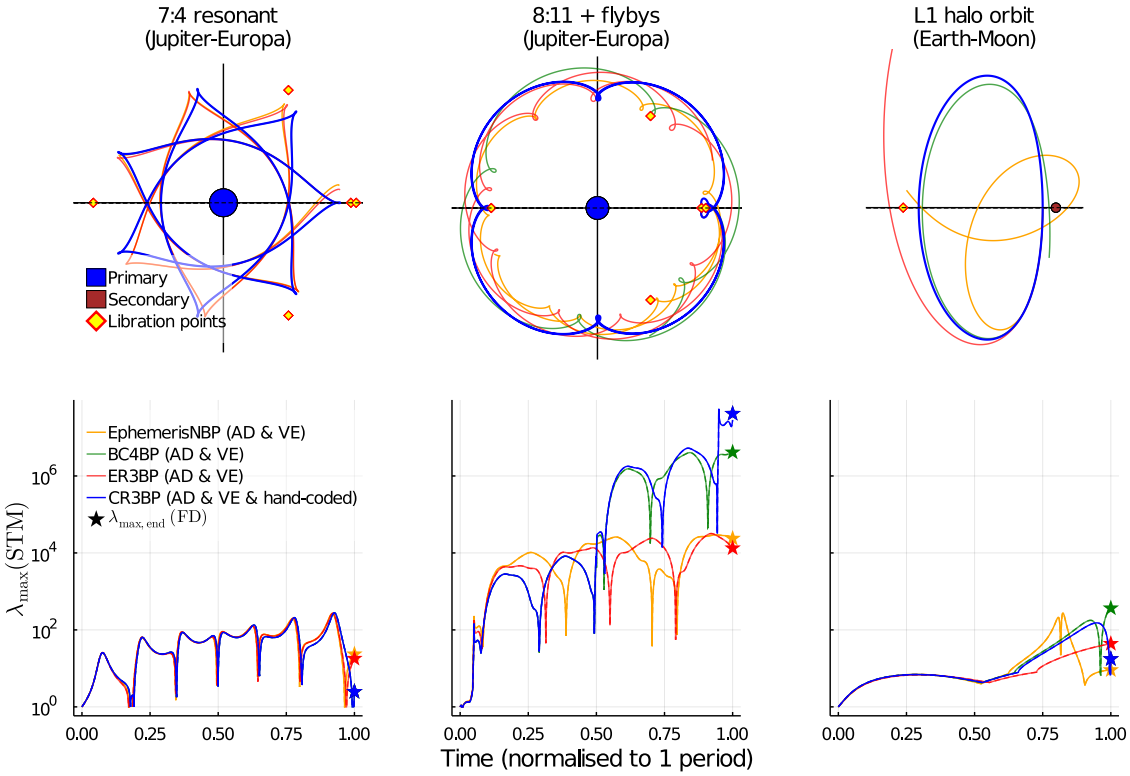

Figure 4: Comparison of 3 trajectory test cases[25] (top; x-y in rotating frame) and their propagated STMs (bottom; maximum eigenvalues $\lambda_{\max}$ shown), as computed by Variational Equations (VE) and Finite (FD) and Automatic Differentiation (AD) functions generated automatically by the Julia compiler.

Table 1: Error in sensitivity index $\lambda_{\max}$ computed using Variational Equations (VE), Finite (FD) and Automatic Differentiation (AD), relative to values given by (Reference 25).

Orbit test case	**Model**	**VE** (rel.error)	**AD** (rel.error)	**FD** (rel.error)
7:4 resonant ("PO1")	CR3BP	8.1×10^{-6}	1.9×10^{-5}	1.1×10^{-5}
8:11 + flybys ("PO2")	CR3BP	2.0×10^{-3}	7.1×10^{-3}	4.8×10^{-1}
L1 halo orbit ("PO3")	CR3BP	3.2×10^{-8}	1.8×10^{-7}	2.7×10^{-3}

propagation for CR3BP are, as expected, the fastest methods and faster than their symbolically-derived counterparts; this includes the generic AD and FD methods used on the hand-coded equations of motion. However, we argue that such hand-coded models, from a programming quality perspective, require comparatively more cumbersome code that bears limited resemblance to their canonical forms typically expressed in literature, and which force developers to change more code due to the coupling of the system and its variational equations. For example, the augmented system requires manually updating the $(6 * 6) = 36$ derived variational equations whenever the equations of motion change, increasing the likelihood of introducing mathematical or programming errors.

By comparison, the symbolic models defined here are simpler to maintain and extend, being directly linked to the canonical equations of motion. Meta-programming and multiple dispatch combine to generate relatively high-performance versions of these models, and allow for exploiting existing or future developments in the ecosystem, such as automatically translating code to run on GPUs.[18] Future efforts to ensure type-stability and minimise dynamic allocations, together with improvements to the symbolic expression manipulation library, may likely lead to further improved performance.

```julia
# Dispatch this method when state vector is Dual numbers -- multiply state by rotation matrix
transform_state(u::AbstractVector{<:ForwardDiff.Dual}, transform) = transform * u
# Otherwise dispatch this generic method -- transform both state and STM variational elements
function transform_state(u, transform)
    new_u = copy(u)
    new_u[begin:STATE_DIMS] .= transform * @view(new_u[1:STATE_DIMS])
    if length(new_u) == (STATE_DIMS + STATE_DIMS*STATE_DIMS)
        STM_elements = reshape(@view(new_u[STATE_DIMS+1:end]), (STATE_DIMS, STATE_DIMS))
        STM_elements .= transform * STM_elements  # Cross-multiply the STM elements appropriately
    end
    return new_u
end
```

Listing 8: Methods for transforming a state vector u to a new reference frame given its instantaneous transformation matrix, comparing AD-based `Dual` number states and VE-augmented states.

Table 2: Representative relative computational costs for propagating a trajectory and an augmented trajectory including a State Transition Matrix (STM), using Finite (FD) and Automatic Differentiation (AD), and symbolically-generated Variational Equations (VE).

	Trajectory	**STM (VE)**	**STM (AD)**	**STM (FD)**
CR3BP (hand-coded)	1.00×	1.22×	1.97×	6.95×
ER3BP	1.64×	2.93×	2.90×	12.79×
CR3BP	1.90×	1.88×	3.18×	14.16×
BC4BP (+Sun)	2.67×	26.84×	4.39×	18.94×
EphemerisNBP (+Sun)	18.75×	36.89×	22.87×	241.21×

The FD method is shown to be both the slowest and least accurate, as it propagates the equations of motion multiple times on slightly differing trajectories. Under FD, only the final STM (the monodromy matrix after a full orbital period of the original CR3BP orbit) is computed, although it should be possible to also extract the full STM trace with relatively minimal additional overhead. The ER3BP model appears to generally outperform the CR3BP model, despite the latter being a simplified version of the former; however this is simply due to the selected `Vern7` adaptive-step solver requiring more steps in particular for the PO2 case's close fly-bys of Europa. In reality, a single step in the CR3BP model is computed slightly faster than the ER3BP model.

Most importantly, we demonstrate the strengths of the AD methods, requiring no additional code (other than to seed and extract the partial derivatives) to compute the STMs of any astrodynamical model provided, and yet resulting in similar accuracies and sometimes better performance than the corresponding VE methods.* Indeed, this matches the experiences of the Celeste project, which combined both hand-coded derivatives and AD to greatly reduce development burden and also provide additional performance improvements.[2]

Single-Shooting Differential Correction

Single-shooting differential correction is a method used to find periodic orbits by iteratively correcting an initial state until a specified periodicity condition is met and convergence to a periodic trajectory is attained. Using a root-finding algorithm and the trajectory sensitivity (STM) as computed previously, we demonstrate a generic differential corrector that finds quasi-periodic axisymmetric-like orbits in any astrodynamical models provided.

The equations of motion of the CR3BP are invariant under the transformation $t = -t$, $y = -y$, for time t and y position of the spacecraft respectively.[26] An initial state propagated forwards in time

*The performance of some VE models is likely hampered by their symbolic derivations, which may improve with future developments. Despite this, their performance is adequate while greatly reducing the development burden.

results in a trajectory that is flipped about the xz-plane as the same state propagated backwards in time. This symmetry implies the existence of axisymmetric orbits: a trajectory that crosses the x-axis perpendicularly in two places will produce one half of the same orbit flipped across the x-axis backwards in time, as in the example of Figure 5. A single-shooting differential corrector finds such orbits from an initial guess of a trajectory that crosses the x-axis at least twice, iteratively correcting it towards the symmetric requirements for some period T of $y = z = \dot{x} = 0$ at $t = 0$ and $T/2$.[24]

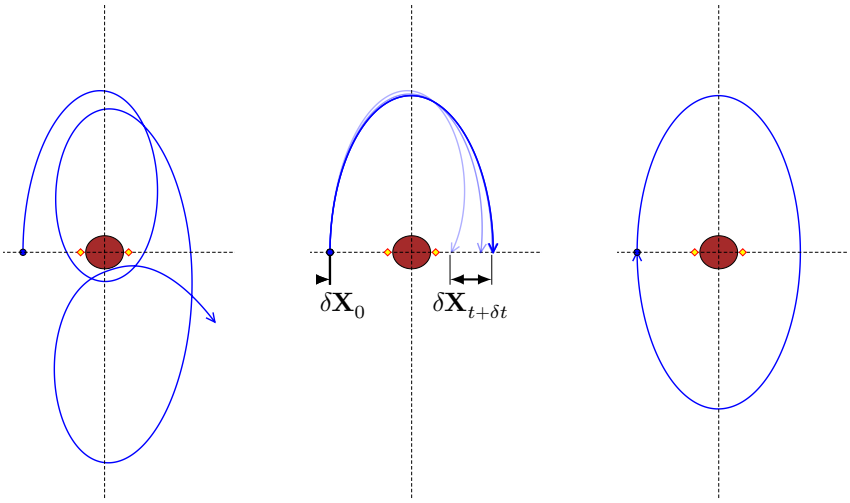

Figure 5: Example of single-shooting differential correction in the Mars-Phobos CR3BP, where an initial guess for a trajectory is iteratively corrected to find a periodic axisymmetric orbit.

Let $\mathbf{X}_t$ be some reference initial trajectory crossing the x-axis at time t. Let $\mathbf{X}^{\star}_{t+\delta t}$ be an ideal trajectory close to $\mathbf{X_t}$ which satisfies the axisymmetric constraints at a time defined as half of its orbital period: $t + \delta t = T/2$. Then, the difference between this ideal trajectory and $\mathbf{X}_t$ is the desired correction $\delta\mathbf{X}_{t+\delta t}$, which depends on a perturbation $\delta\mathbf{X}_0$ to the initial state, as seen in Figure 5. A 1^{st}-order linear approximation of this relation is given by:[24,26]

$$\delta\mathbf{X}_{t+\delta t} = \Phi(t; \mathbf{X}_0)\delta\mathbf{X}_0 + \dot{\mathbf{X}}_t\delta t \tag{5}$$

where $\Phi(t; \mathbf{X}_0)$ is the STM computed at time t with respect to the initial state $\mathbf{X}_0$.

A single-shooting differential corrector applies Equation (5) iteratively within a root finding procedure to find a periodic orbit. An example of such a corrector is shown in Listing 9, showing the inner step of a quasi-Newton method (Newton method with damped step sizes). The procedure is further helped by terminating propagation immediately after a desired number of x-axis crossings (where $\delta y_{t/2} = 0$), and by holding the initial x-position constant ($\delta x_0 = 0$), as per (Reference 26).

Although axisymmetric orbits are defined within the context of the CR3BP, our `AxisymmetricCorrector` corrector can be applied to any of our differentiable astrodynamical models, leading (through simple parameter continuation) to families of (quasi-)periodic orbits as in Figure 6. For example, applying this generic corrector to the EphemerisNBP model makes it possible to generate 3-dimensional Quasi-Satellite Orbits (QSOs), as shown by the two example orbits with initial conditions given in Table 3. While our implementation does not yet extend beyond this simple proof-of-concept, it highlights the benefits designing generic functions and relying on the programming language to do the heavy lifting of composing functionality and providing performance. Future research in similar directions may lead to new methods for generating scientific orbits or capture trajectories in higher-fidelity models, perhaps applying existing approaches, such as those of prior works, directly.[27]

We make note of several caveats of our implemented proof-of-concept. Firstly, it requires some additional steps, not shown explicitly in Listing 9, to convert between reference frames when necessary.

```julia
function (corrector::AxisymmetricCorrector)(state::State; step_size=1.0, num_crossings=1)
    reference_state = state_to_frame(state, frame(corrector))
    callback        = terminate_on_x_crossing(corrector, num_crossings)
    trajectory      = STM_trace(reference_state; callback)      # Propagate initial guess with STM
    @assert trajectory.retcode == :X_Crossing                   # Initial guess ends at x-crossing
    STM     = hcat([ForwardDiff.partials(u) for u in trajectory[end]])  # Extract STM at x-crossing
    # Compute the perturbation to the initial state (see Eq. (13) of Reference 26)
    δX_tdt = ForwardDiff.value.(-trajectory[end][residual_vars(corrector)])              # δX_{t+δt}
    Ẋ_t    = compute_derivative(trajectory, trajectory.t[end])                           # Ẋ_t
    rhs     = 1/Ẋ_t[2] * Ẋ_t[residual_vars(corrector)] * STM[2, free_vars(corrector)]'
    Φ_t     = STM[residual_vars(corrector), free_vars(corrector)] - rhs                  # φ(t; X_0)
    δX_0    = (Φ_t \ δX_tdt)                                  # δX_0 = φ(t; X_0)^{-1} δX_{t+δt} (Equation (5))
    u0      = copy(reference_state.u0)
    u0[free_vars(corrector)] .+= step_size * δX_0             # Apply correction with damped step size
    # Return the perturbed state, corrected error |δX_{t+δt}|, and ending time t
    return remake(reference_state; u0), norm(δX_tdt), trajectory.t[end]
end
```

Listing 9: Illustrative example of a quasi-Newton iteration step for finding axisymmetric orbits, as part of a generic single-shooting differential correction procedure.

Table 3: Initial conditions for the quasi-periodic orbits shown in Figure 6, as propagated in the Mars-Phobos-Sun EphemerisNBP model centered on Mars. Times are relative to J2000.

	$1,264 \times 4,592$km **3D-QSO**	55×306km **3D-QSO**
x_0 [km]	−1721.2905853705563	−1973.7797398622927
y_0 [km]	−8035.387965174319	−9214.066550501659
z_0 [km]	483.12718239638104	553.9951536612949
$\dot{x}_0$ [km/s]	1.8582536023371838	1.8526285717782527
$\dot{y}_0$ [km/s]	−0.4808842668186627	−0.4477211699864998
$\dot{z}_0$ [km/s]	−1.466746074609674	−0.9352172022943867
t_0 [s]	0.0	0.0
$T/2$ [s]	13745.313782798674	13252.01104574985
t_f [s]	2749062.7523703636	2815256.487335143

In particular, trajectories in the EphemerisNBP model must be converted to an inertial frame for propagation, and to a normalised rotating frame for event-finding on and for computing sensitivities relative to the x-axis crossing (found in the callback on Line 3). These cases are largely covered by the use of a SPICE dynamic reference frame built at run-time and a simple matrix multiplication in Listing 8. However, the time correction δt requires a time derivative of the final state ($\dot{\mathbf{X}}_t$), which can not be computed easily symbolically or by AD, as the `SPICE.jl` library calls an untraceable external C procedure (`sxform`) dependent on time for computing the rotation matrix. As such, the generic `AxisymmetricCorrector` must dispatch to a model-specific procedure (Line 9), which for the EphemerisNBP is either an AD approximation that is fast but often does not converge to quasi-periodic orbits, or an FD approximation which can better approximate the time variation, but is slow and can be inaccurate, sometimes resulting in poor convergence.

PROPAGATION PERFORMANCE & BENCHMARKS

An initial evaluation of the developed `OrbitalTrajectories.jl` toolkit assessed its propagation performance against jTOP, a JAXA in-house trajectory design and optimisation toolkit comprising a MATLAB library with a Fortran 90 core, and used in several mission analyses including for upcoming space exploration missions.[28] The Fortran core consists of hand-coded astrodynamical models, including an ephemeris N-body propagator used for this comparison. The jTOP equations of motion are propagated using `DDEABM`, a Fortran-based 12$^{\text{th}}$-order Adams-Bashforth-Moulton ODE

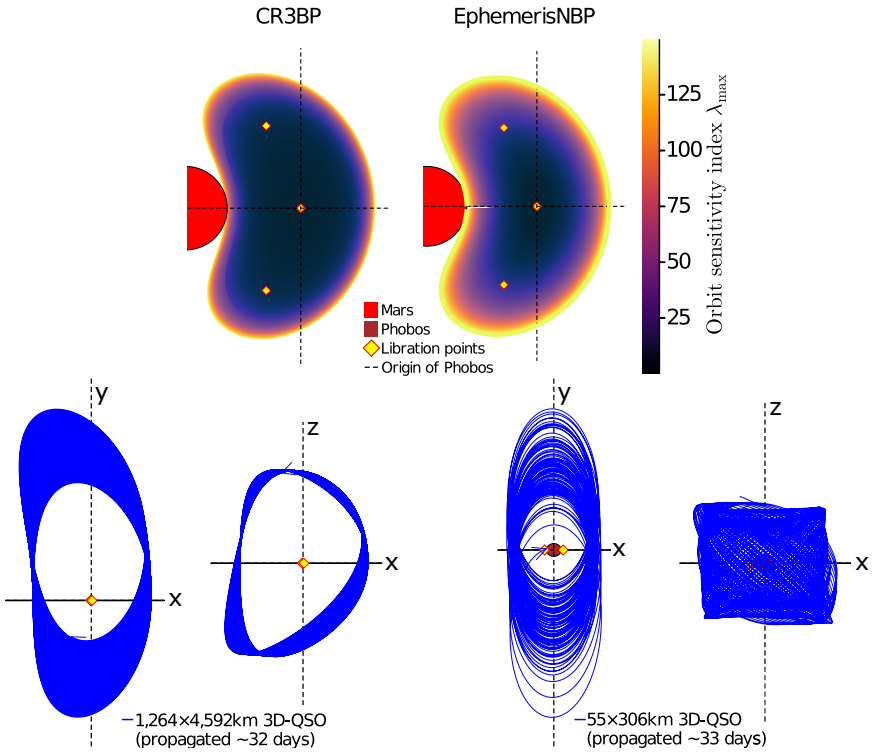

Figure 6: Orbits found using a generic `AxisymmetricCorrector`. (top) Families of Quasi-Satellite Orbits (QSOs) in a normalised rotating x-y frame of the Mars-Phobos(-Sun) system, coloured by relative orbit sensitivity ($\lambda_{\max}$ of STM after approximately 1 orbital period). In the CR3BP model, the orbits are planar, axisymmetric, and periodic. In the EphemerisNBP model, the orbits are quasi-periodic and 3-dimensional. (bottom) Two 3D-QSOs found by applying the `AxisymmetricCorrector` to the EphemerisNBP model.

solver from the SLATEC Common Mathematical Library. `DDEABM` is an adaptive-step method, and jTOP sets its relative and absolute tolerances to 1×10^{-7} and 1×10^{-9} by default respectively.

For comparative tests, we recreated the Forward Propagation (FWP) block of the EQUULEUS mission analysis[28] in Julia in an almost line-for-line direct translation from MATLAB, except calling our `OrbitalTrajectories.jl` toolkit's EphemerisNBP model propagator. The FWP block begins by selecting an initial near-Earth state and propagating it forward for 3.2×10^1 hours ("FWP1"). Then, a massive grid search is performed to propagate the ending state with a large number of initial ΔV impulsive manoeuvres for a further 1.5 years, in the aim of finding optimal lunar flybys ("FWP2").

The times reported here were based on runs from an Intel i3-6100 CPU (3.7 GHz, 2 cores) running single-threaded scripts in MATLAB R2020b and Julia 1.5.3. For the FWP1 tests, the jTOP propagator was timed using a `matlab.perftest.TimeExperiment` measuring only the `calllib` call to the compiled Fortran entry-point of jTOP; as such, these runtimes should include any overheads or bottlenecks caused by MATLAB, providing a useful comparison to Julia. For the FWP2 tests, the jTOP propagator was timed using a `tic`/`toc` pair surrounding the `calllib` call for each individual trajectory. The `OrbitalTrajectories.jl` toolkit's EphemerisNBP propagator was timed using the `BenchmarkTools.jl` package's `@benchmark` macro, which discards the initial time taken to run-time compile the propagator, and the built-in `@elapsed` macro, for the FWP1 and FWP2 tests respectively. As described for the EphemerisNBP model on page 8, we injected an intermediate computation of the body positions into the generated propagation function code using meta-programming, resulting in at least an order of magnitude performance improvement (48 calls to SPICE's `spkpos` reduced to 2 per timestep).

Three separate ODE solvers provided by the `DifferentialEquations.jl` common solver interface were tested,[10] with tolerances set to the default values used by jTOP. Figure 7 shows the runtime distributions of FWP1 and FWP2 propagations for each solver, in which, with intermediate computations, our EphemerisNBP model outperformed the jTOP propagator by up to an order of magnitude on each solver tested. The Julia `DDEABM` solver is a simple wrapper around the same Fortran solver used by jTOP; its performance shows a clear difference to jTOP's MATLAB-Fortran performance, which is likely affected by the MATLAB run-time or communication overheads with Fortran. By comparison, the pure-Julia `VCABM` solver, an adaptive-order/time Adams-Moulton method derived from `DDEABM`, provides the best performance while also additionally supporting higher-order state interpolation and custom timesteps and event functions. Similarly, the pure-Julia `Vern7` solver, a Verner's "Most Efficient" 7/6 Runge-Kutta adaptive-step method, additionally supports lazy 7^{th}-order interpolation, and is generally recommended as an efficient replacement for the `VCABM` solver.

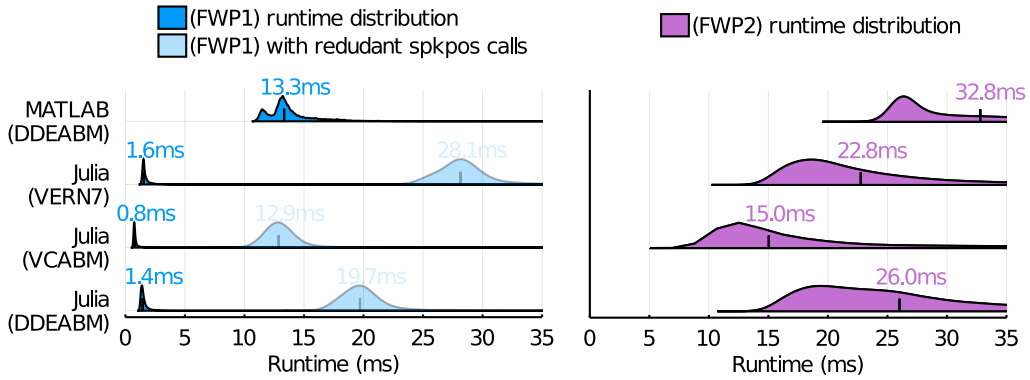

Figure 7: Runtime distributions for trajectory propagation in the Earth-Moon-Sun system, comparing our Julia-based EphemerisNBP model and jTOP's N-body propagator (DDEABM solver called from MATLAB). The median runtimes are labelled for (left) 19 near-Earth trajectories propagated 1000 times each; and (right) $\sim$150,000 initial ΔV burns propagated over a long period for each solver.

The much greater diversity and longer propagation times of FWP2 trajectories manifest as wider runtime distributions with long tails of trajectories requiring more detailed timestepping due to sensitivity to the lunar fly-by. Excluding pre- and post-processing, a single FWP2 run of $\sim$150,000 initial ΔV burns required approximately 93.3 minutes of total propagation with jTOP. By comparison, our EphemerisNBP model required 49.1, 83.8, and 84.4 minutes for the `VCABM`, `DDEABM`, and `Vern7` solvers respectively, demonstrating the potential value of even small gains in computational efficiency.

A comparison of the trajectories' errors in Figure 8 shows that both pure-Julia solvers computed trajectories within a relative tolerance of at worst 1×10^{-7} relative to the jTOP `DDEABM`-computed reference trajectories, with `Vern7` providing a consistently accurate trajectory. The Julia-based `DDEABM` solver only supports linear interpolation, so its accuracy appears to suffer due to being interpolated to match the jTOP reference trajectories' timesteps; in reality, with both using the same underlying Fortran-based `DDEABM` solver, its trajectories should match those from jTOP exactly (as seen at the start and end of each trajectory), with remaining differences being due to propagation of normalised and un-normalised equations of motion for jTOP and our EphemerisNBP respectively.

Finally, we briefly examine the differences between jTOP and `OrbitalTrajectories.jl` from a qualitative programming perspective. As `DDEABM` is a Fortran-based solver, jTOP is limited in its ability to support advanced ODE solver capabilities such as high-order interpolations and custom event functions and numerical types, except with direct modification of its Fortran core to add such support. By comparison, `OrbitalTrajectories.jl` seamlessly allows switching between solvers, interpolating trajectories with high accuracy, using custom event functions, and propagating custom

numerical types such as auto-differentiable `Dual` numbers, thanks to the comprehensive common ODE solver interface provided by `DifferentialEquations.jl`. The high-level syntax and interactive compilation of the Julia language makes these features much more accessible than the Fortran core of jTOP, which must be recompiled after every change. `OrbitalTrajectories.jl`'s symbolic models based on `ModelingToolkit.jl` provide automatic extensibility such as in building augmented systems with variational equations, whereas separate subroutines with hand-coded derivatives must be defined in Fortran for jTOP. However, despite these seeming disadvantages, jTOP provides a comprehensive high-level interface (as a MATLAB library) for space mission analysis, with numerous capabilities in trajectory design and patching, multiple-shooting methods, and non-linear optimisation.[29,30]

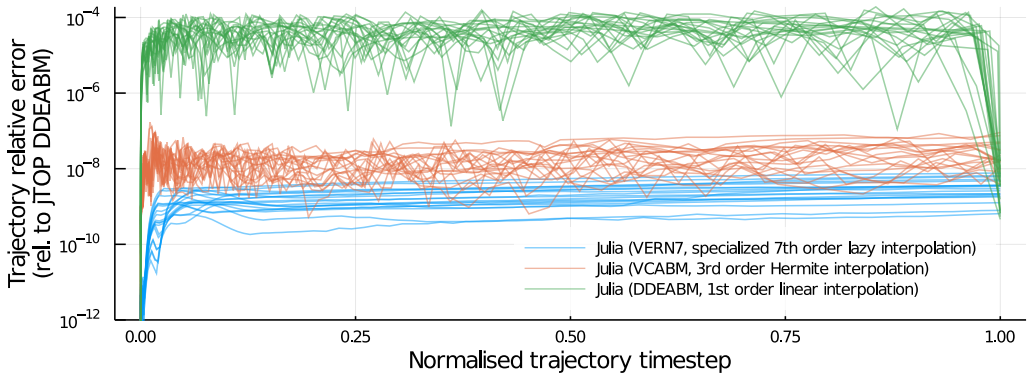

Figure 8: Error in the trajectory propagations of 19 FWP1 initial cases, relative to the reference trajectories computed by jTOP (using the Fortran DDEABM solver). Each trajectory is interpolated to match the reference trajectories' timesteps.

CONCLUSION

In this initial design of the `OrbitalTrajectories.jl` toolkit, we have demonstrated several modern numerical programming techniques made available by the Julia programming language, and discussed their relevance to and possible applications in the field of astrodynamic trajectory design. We showed that the Julia language makes it relatively simple to combine such techniques in perhaps unexpected ways, exploiting its inherent composability to combine unrelated but generic functionality with ease. Through multiple dispatch and meta-programming, the ability to easily compose otherwise independent components has been a powerful driver of innovation, having led some Julia practitioners to significant advances in differential programming,[3] scientific machine learning,[5] and dynamical systems.[31,32]

Our `OrbitalTrajectories.jl` toolkit demonstrates seamless integration with symbolic expressions, differential equation systems, automatic differentiation, and generic methods. Firstly, we showed how it is possible to define astrodynamical models as simple composable systems of symbolic equations, allowing new models to be derived directly – such as in the CR3BP model from the ER3BP, and also in augmenting models with their variational equations – without requiring manual derivation or hand-coding of derived equations by the developer. Being built in a fully-differentiable language, we showed how these models can be differentiated automatically, providing state or parameter sensitivities for applications such as stability analysis and finding periodic and quasi-periodic orbits. In so doing, we described some ways in which methods can be defined generically to accept arbitrary astrodynamical models, maximising the value of individual functions and providing greater code reuse. Finally, we briefly evaluated the performance of a Julia-based restricted N-body propagator, and showed that in at least one mission analysis use case it was able to exceed the performance of an existing and actively-used astrodynamics mission tool by up to an order of magnitude.

Future improvements on `OrbitalTrajectories.jl` will focus towards an open-source release for use as a mission analysis tool, including additions to the astrodynamical models and providing perturbations such as spherical harmonics and shape models, and trajectory optimisation.[33] The universal differentiability of the Julia language may lead to more applications of differentiable systems;[34] for example, a full-featured pure-Julia ephemeris library[14] could provide automatic differentiation on SPICE kernel data, solving some of the problems we faced in the implementation of generic methods such as the `AxisymmetricCorrector`. Furthermore, the rapidly evolving Julia ecosystem may provide further inspiration for novel applications, with interesting developments in optimisation under uncertainties,[32] physics-based machine learning,[35] and GPU acceleration.[18] We believe that applications such as these are in line with the modern needs of space mission analysis and trajectory design, and find that Julia is an attractive option for the future development of astrodynamics methods and tools.

ACKNOWLEDGMENTS

Dan Padilha was supported by the Epson International Scholarship Foundation and JASSO Monbukagakusho Honors Scholarship throughout his M.Eng. studies. This work has been partially supported by Japan Society for the Promotion of Science Grants-in-Aid No. 19F18371.

REFERENCES

[1] N. Arora and A. Petropoulos, "Experiments with Julia for Astrodynamics Applications," *AAS/AIAA Astrodynamics Specialist Conference*, Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2015.

[2] J. Regier, K. Fischer, K. Pamnany, A. Noack, J. Revels, M. Lam, S. Howard, R. Giordano, D. Schlegel, J. McAuliffe, R. Thomas, and Prabhat, "Cataloging the visible universe through Bayesian inference in Julia at petascale," Vol. 127, 2019, pp. 89–104. PII: S0743731518304672, 10.1016/j.jpdc.2018.12.008.

[3] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, "∂P: A Differentiable Programming System to Bridge Machine Learning and Scientific Computing," 2019.

[4] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, "DiffEqFlux.jl - A Julia Library for Neural Differential Equations," 2019.

[5] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, and A. Ramadhan, "Universal Differential Equations for Scientific Machine Learning," 2020.

[6] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," 2017, 10.1137/141000671.

[7] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky, "Julia: dynamism and performance reconciled by design," Vol. 2, No. OOPSLA, 2018, pp. 1–23, 10.1145/3276490.

[8] T. Kwong and S. Karpinski, *Hands-On Design Patterns and Best Practices with Julia*. Packt Publishing, 2020.

[9] S. Campagnola, N. Ozaki, Y. Sugimoto, C. H. Yam, H. Chen, Y. Kawabata, S. Ogura, B. Sarli, Y. Kawakatsu, R. Funase, and S. Nakasuka, "Low-Thrust Trajectory Design and Operations of PROCYON, the First Deep-Space Micro-Spacecraft," *66th International Astronautical Congress*, 2015.

[10] C. Rackauckas and Q. Nie, "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia," Vol. 5, 2017, 10.5334/jors.151.

[11] H. Eichhorn, J. L. Cano, F. McLean, and R. Anderl, "A comparative study of programming languages for next-generation astrodynamics systems," Vol. 10, No. 1, 2018, pp. 115–123. PII: 170, 10.1007/s12567-017-0170-8.

[12] R. A. J. Chagas, F. L. d. Sousa, A. C. Louro, and W. G. dos Santos, "Modeling and design of a multidisciplinary simulator of the concept of operations for space mission pre-phase A studies," Vol. 27, No. 1, 2019, pp. 28–39, 10.1177/1063293X18804006.

[13] G. Hott, R. A. J. Chagas, and F. L. d. Sousa, "Estudos preliminares na otimização de manobras de atitude utilizando o algoritmo MGEOreal e o pacote SatelliteToolbox.jl," *X Workshop em Engenharia e Tecnologia Espaciais*, 2019.

[14] K. M. Martin, T. Minkoff, P. Landon, B. Gray, and D. Landau, "Julia Language 1.1 Ephemeris Reader and Gravitational Modeling Program for Solar System Bodies," *AIAA Scitech Forum*, American Institute of Aeronautics and Astronautics, 2020, p. 313, 10.2514/6.2020-1468.

[15] C. H. Acton, "Ancillary data services of NASA's Navigation and Ancillary Information Facility," Vol. 44, No. 1, 1996, pp. 65–70. PII: 0032063395001077, 10.1016/0032-0633(95)00107-7.

[16] F. Topputo, D. A. Dei Tos, K. V. Mani, S. Ceccherini, C. Giordano, V. Franzese, and Y. Wang, "Trajectory Design in High-Fidelity Models," *7th International Conference on Astrodynamics Tools and Techniques (ICATT)*, 2018.

[17] M. Zenger and M. Odersky, "Independently Extensible Solutions to the Expression Problem," 2004.

[18] T. Besard, C. Foket, and B. d. Sutter, "Effective Extensible Programming: Unleashing Julia on GPUs," Vol. 30, No. 4, 2019, pp. 827–841, 10.1109/TPDS.2018.2872064.

[19] D. A. Dei Tos and F. Topputo, "On the advantages of exploiting the hierarchical structure of astrodynamical models," Vol. 136, 2017, pp. 236–247. PII: S0094576516301928, 10.1016/j.actaastro.2017.02.025.

[20] Z.-F. Luo, F. Topputo, F. Bernelli-Zazzera, and G.-J. Tang, "Constructing ballistic capture orbits in the real Solar System model," Vol. 120, No. 4, 2014, pp. 433–450. PII: 9580, 10.1007/s10569-014-9580-5.

[21] C. C. Margossian, "A Review of automatic differentiation and its efficient implementation," Vol. 9, No. 4, 2019, p. 64, 10.1002/WIDM.1305.

[22] J. Revels, M. Lubin, and T. Papamarkou, "Forward-Mode Automatic Differentiation in Julia," 2016.

[23] D. J. Grebow, "Generating Periodic Orbits in the Circular Restricted Three-Body Problem with Applications to Lunar South Pole Coverage," 2006.

[24] W. S. Koon, M. W. Lo, J. E. Marsden, and S. D. Ross, *Dynamical Systems, the Three-Body Problem, and Space Mission Design*. Marsden Books, 2011.

[25] E. Pellegrini and R. P. Russell, "On the Computation and Accuracy of Trajectory State Transition Matrices," Vol. 39, No. 11, 2016, pp. 2485–2499, 10.2514/1.G001920.

[26] R. P. Russell, "Global Search for Planar and 3D Periodic Orbits Near Europa," Vol. 54, No. 2, 2006.

[27] D. A. Dei Tos, R. P. Russell, and F. Topputo, "Survey of Mars Ballistic Capture Trajectories Using Periodic Orbits as Generating Mechanisms," Vol. 41, No. 6, 2018, pp. 1227–1242, 10.2514/1.G003158.

[28] N. Baresi, T. Chikazawa, D. A. Dei Tos, S. Campagnola, Y. Kawabata, K. Ichinomiya, N. Ozaki, K. Kakihara, K. Oguri, R. Funase, and Y. Kawakatsu, "The New Trajectory Design of EQUULEUS," *32nd International Symposium on Space Technology and Science (ISTS)*, 2019.

[29] K. Oguri, K. Oshima, S. Campagnola, K. Kakihara, N. Ozaki, N. Baresi, Y. Kawakatsu, and R. Funase, "EQUULEUS Trajectory Design," Vol. 67, No. 3, 2020, pp. 950–976, 10.1007/s40295-019-00206-y.

[30] S. Campagnola, C. H. Yam, Y. Tsuda, N. Ogawa, and Y. Kawakatsu, "Mission analysis for the Martian Moons Explorer (MMX) mission," Vol. 146, 2018, pp. 409–417. PII: S0094576517317575, 10.1016/j.actaastro.2018.03.024.

[31] S. Bogomolov, M. Forets, G. Frehse, K. Potomkin, and C. Schilling, "JuliaReach," *HSCC '19* (N. Ozay and P. Prabhakar, eds.), CPS-IoT Week, The Association for Computing Machinery, 2019, pp. 39–44, 10.1145/3302504.3311804.

[32] A. R. Gerlach, A. Leonard, J. Rogers, and C. Rackauckas, "The Koopman Expectation: An Operator Theoretic Method for Efficient Analysis and Optimization of Uncertain Hybrid Dynamical Systems," 2020.

[33] N. Ozaki, "Equation of Motion and Jacobian Matrix in the N-Body Problem with Spherical Harmonics," 2017.

[34] A. Vigliotti and F. Auricchio, "Automatic Differentiation for Solid Mechanics," 2020. PII: 9396, 10.1007/s11831-019-09396-y.

[35] K. Gao, G. Mei, F. Piccialli, S. Cuomo, J. Tu, and Z. Huo, "Julia language in machine learning: Algorithms, applications, and open issues," Vol. 37, 2020, p. 100254, 10.1016/j.cosrev.2020.100254.