🚀 Going to Jupiter with Julia

```julia
using GeneralAstrodynamics # a new package for common astrodynamics calculations!
```

- **Joe Carpinelli**
- Title à la **Going to Mars with Python using poliastro**
- Presented at **JuliaCon** 2021, available at **jcarpinelli.dev/juliacon**

Table of Contents

Directions to Jupiter

Where do we even start?

- Astrodynamics coursework, research, and online resources
- A new Julia package, `GeneralAstrodynamics.jl`

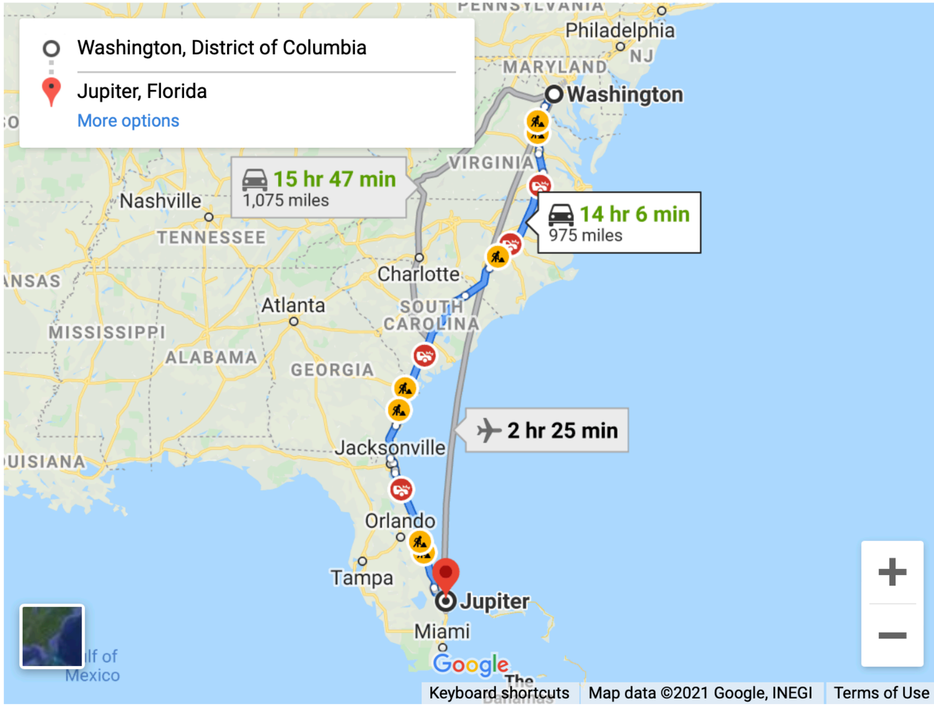


Table of Contents

Challenges in Astrodynamics

As a field, astrodynamics is incredibly demanding of scientific computing.

- Calculations are expensive, e.g. numerical integration, iterative solvers
- Data may be large, e.g. solar system ephemeris
- Equations are complicated, e.g. analytical Halo orbit approximations
- Unit errors are an easy trap!

Good News

Julia can help! As others have found (including the talks shown below), Julia has enormous potential for scientific computing within the field of Astrodynamics!

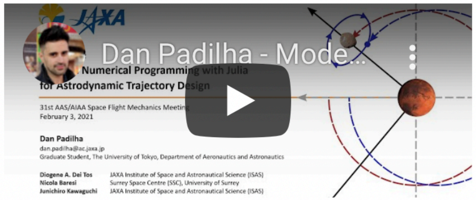


Table of Contents

Let's plan a trip to Jupiter!

Introducing Astrodynamics research alongside GeneralAstrodynamics.jl!

Goals

- Design a trajectory between Earth and Jupiter using simple models
- Highlight language features, and packages in the ecosystem, that meet the challenges of computational astrodynamics
- Present a new astrodynamics package, `GeneralAstrodynamics.jl`

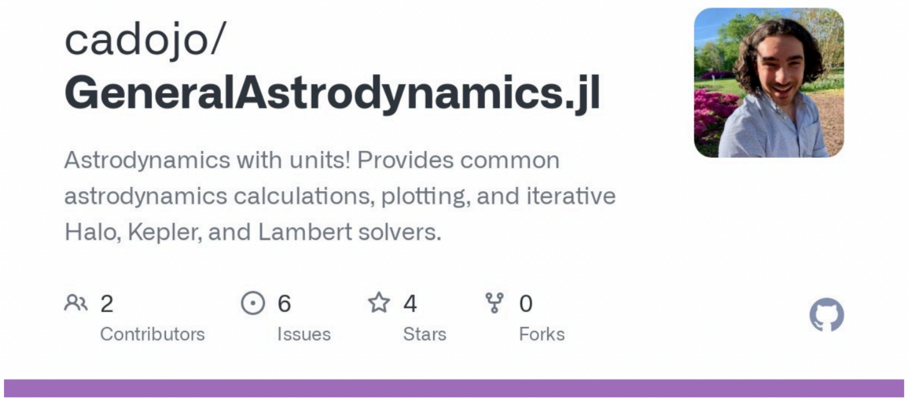


*Image credit to **[Twitter's](#)** new GitHub Card format.*

Table of Contents

Equations of Motion (R2BP)

How will our spacecraft move?

Restricted Two-body Problem

Assume your spacecraft is only pulled by the gravity of **one** celestial body. Let $\mathbf{r}$, $\mathbf{v}$, μ be the spacecraft's position, spacecraft's velocity, and celestial body's mass parameter respectively.

$$\dot{\mathbf{r}} = \mathbf{v}$$
$$\dot{\mathbf{v}} = -\mu \frac{\mathbf{r}}{||\mathbf{r}||}$$

Write these dynamics into a function, and plug that function into a numerical integrator (i.e. the excellent `DifferentialEquations.jl`) **simulate** (a.k.a. **propagate**) your dynamics forward or backward in time.

Tip

If your self-coded function allocates memory, your numerical integration **will** be sub-optimal. Use `ModelingToolkit.jl` and `AstrodynamicalModels.jl` to generate fast functions for you!

```
A `ModelingToolkit.ODESystem` for the Restricted Two-body Problem.

The Restricted Two-body Problem is a simplified dynamical model  describing one sm
all body (spacecraft, etc.) and one celestial  body. The gravity of the celestial
body exhibits a force on the  small body. This model is commonly used as a simplif
ication to  descibe our solar systems' planets orbiting our sun, or a  spacecraft
orbiting Earth.
```

```
@terminal print(@doc R2BP)
```

Table of Contents

Hohmann Transfer

The first interplanetary transfer design they teach you!

Symbol	Label	Expression
r_1	`Earth's orbital radius w.r.t. Sun`	1.496×10^8 km
r_2	`Jupiter's orbital radius w.r.t. Sun`	7.786×10^8 km
v_1	`Initial spacecraft velocity w.r.t. Sun`	$\sqrt{\frac{2\mu}{r_1} - \frac{2\mu}{r_1+r_2}}$
v_2	`Final spacecraft velocity w.r.t. Sun`	$\sqrt{\frac{2\mu}{r_2} - \frac{2\mu}{r_1+r_2}}$

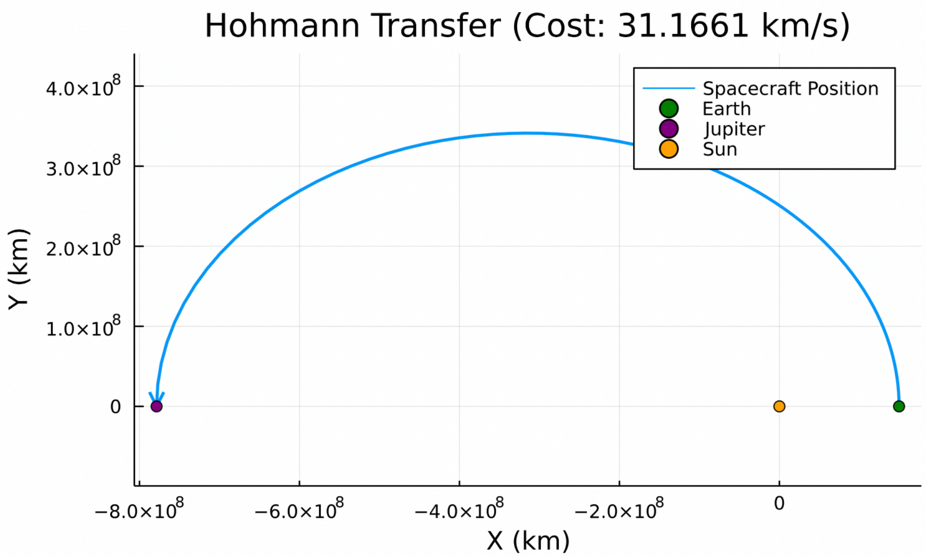


Table of Contents

← →

More Equations of Motion (CR3BP)

Interplanetary superhighways, under active research!

- A more advanced interplanetary transfer concept uses **Circular Restricted Three-body** dynamics

$$
\begin{aligned}
r_1 &= \sqrt{(x^\star + \mu^\star)^2 + (y^\star)^2 + (z^\star)^2} \\
r_2 &= \sqrt{(x^\star + \mu^\star - 1)^2 + (y^\star)^2 + (z^\star)^2} \\
\ddot{x}^\star &= 2\dot{y}^\star + x^\star - \frac{(1-\mu^\star)(x^\star + \mu^\star)}{r_1^3} - \frac{\mu^\star(x^\star - 1 + \mu^\star)}{r_2^3} \\
\ddot{y}^\star &= -2\dot{x}^\star + y^\star - \frac{(1-\mu^\star)y^\star}{r_1^3} - \frac{\mu^\star y^\star}{r_2^3} \\
\ddot{z}^\star &= -\frac{(1-\mu^\star)z^\star}{r_1^3} - \frac{\mu^\star z^\star}{r_2^3}
\end{aligned}
$$

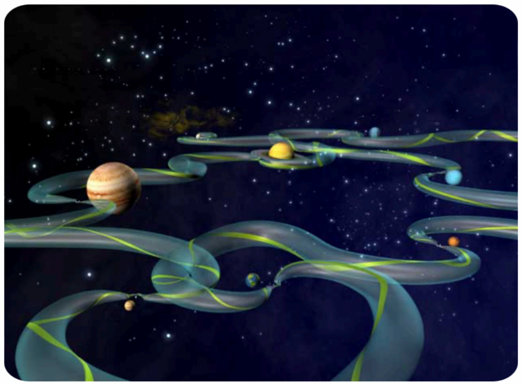


*Image credit to **NASA's** overview page on manifold transfer designs.*

Table of Contents

Manifold Transfer Concept

Manifold-based interplanetary missions are a 3 step process.

1. Launch from Earth into a **stable manifold** within the Sun-Earth system
2. Perturb from the Sun-Earth Halo onto an **unstable manifold**
3. Transfer onto a **stable manifold** of the desired destination Halo orbit

Mission Phase: Phase One

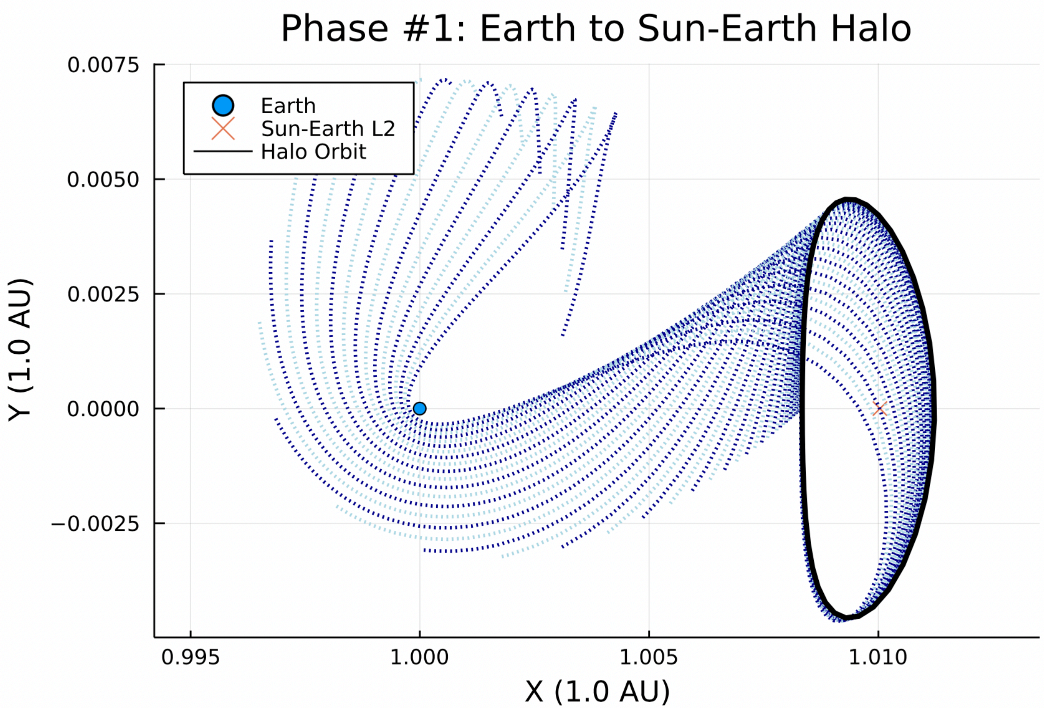


Table of Contents

Manifold Transfer Concept

Manifold-based interplanetary missions are a 3 step process.

1. Launch from Earth into a **stable manifold** within the Sun-Earth system
2. Perturb from the Sun-Earth Halo onto an **unstable manifold**
3. Transfer onto a **stable manifold** of the desired destination Halo orbit

Mission Phase: Phase Two

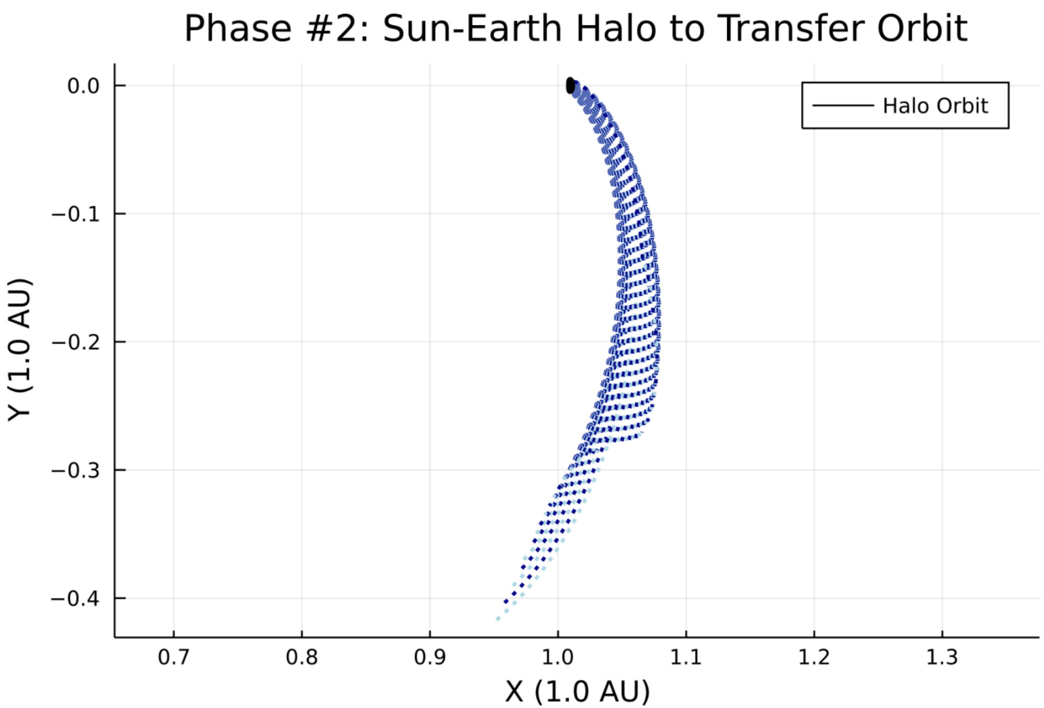


Table of Contents

Manifold Transfer Concept

Manifold-based interplanetary missions are a 3 step process.

1. Launch from Earth into a **stable manifold** within the Sun-Earth system
2. Perturb from the Sun-Earth Halo onto an **unstable manifold**
3. Transfer onto a **stable manifold** of the desired destination Halo orbit

Mission Phase: Phase Three

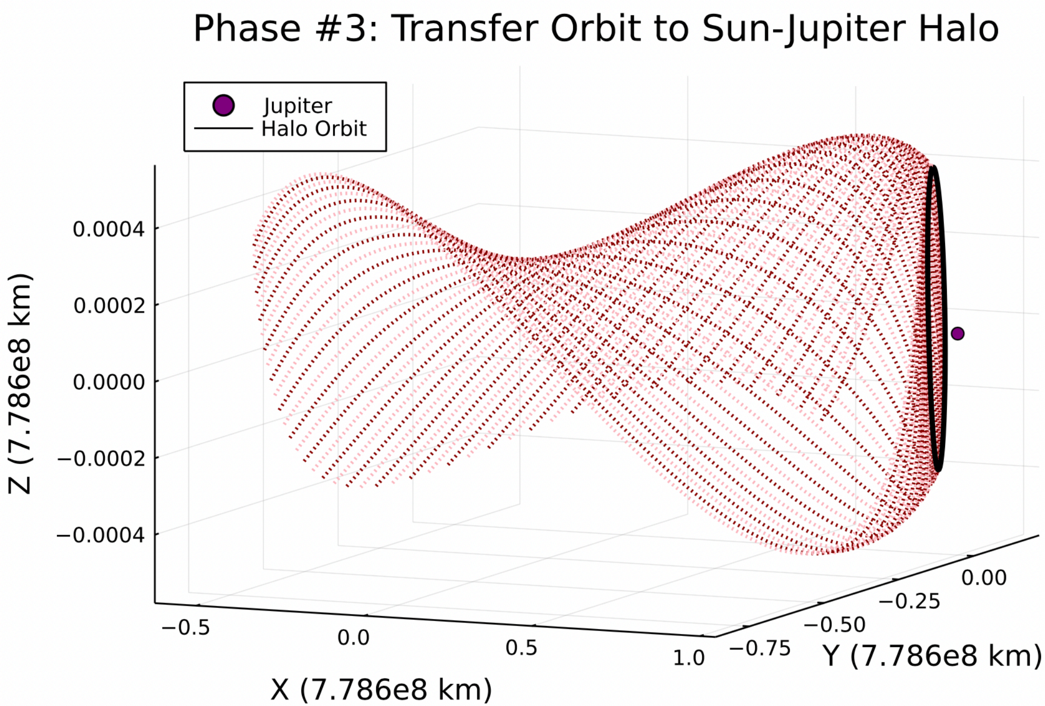


Table of Contents

Conclusions

Julia and its package ecosystem let me be a (productive) lazy astrodynamics developer.

Language Features

- Julia's speed, unicode character support, and clever type system with multiple dispatch allow for fast, mathematically expressive, and easy-to-type astrodynamics code

Ecosystem Packages

- `Unitful.jl` and associated extensions are fantastic unit-handling packages
- `DifferentialEquations.jl`, `ModelingToolkit.jl`, and model packages like `AstrodynamicalModels.jl` let you type dynamics for **correctness**, not **computational efficiency**
- `Plots.jl` is very simple and intuitive to use

Interested in Astrodynamics? Help out!

- I've had a lot of fun developing `GeneralAstrodynamics.jl`, and there's more work to do!
- Don't be shy about filing issues, making pull requests, and otherwise contacting me if you're interested in helping out with features. No astrodynamics experience is required!

Psst!

Hey everyone, switch your Twitter handles to your favorite Julia macros! I've claimed `@code_typed`.

Table of Contents

Credits

Thanks, all!

- The Julia Programming Language, `Unitful.jl`, `UnitfulAstro.jl`, `DifferentialEquations.jl`, and `ModelingToolkit.jl` are **excellent** tools for scientific computing
- `Pluto.jl` is excellent for showing demos, concepts, and JuliaCon presentations 😁
- Thanks to Dr. Barbee for continued guidance relating to finding periodic orbits, and designing manifold-based interplanetary transfers
- Thanks to Dr. Mireles for providing well-explained and publicly available notes and periodic orbit-finding MATLAB code on his **website**

Finally, thank you for watching!

Slides are available at **jcarpinelli.dev/juliacon**. Please don't hesitate to reach out with questions!

Table of Contents

📦 Extras

Toggle Presentation

Table of Contents

Dependencies

Extra Packages

```
begin

    using Plots ✓
    using PlutoUI ✓
    using Latexify ✓
    using Rotations ✓
    using Symbolics ✓
    using StaticArrays ✓
    using LinearAlgebra ✓
    using DifferentialEquations ✓
    using Unitful ✓ , UnitfulAstro ✓
    using AstrodynamicalModels ✓
    using GeneralAstrodynamics ✓

end
```

Macros

```
"""
A wrapper macro for PlutoUI.with_terminal.
"""
macro terminal(expr)
    quote
        with_terminal() do
            $(esc(expr))
        end
    end
end;
```

Table of Contents

Unicode, Units, Multiple Dispatch

Oh my!

Unicode Characters Help with Equations

- Equations are mathematically expressive!

```
ṙ = -μ * (r / norm(r))
```

Unit Handling with `Unitful.jl`

- `Unitful.jl` and extensions (i.e. `UnitfulAstro.jl`, `UnitfulAngles.jl`) are the best unit handling packages I've seen yet

```
using LinearAlgebra, Unitful
r = [0.0, 11_500, 0.0]u"km"
norm(r) == 11_500_000u"m"
```

Multiple Dispatch and Speed

- Julia's type system allows for multiple equations for the same "calculation", e.g. orbital period calculations for different conic sections

```
period(a, μ) = 2π * √(a^3 / μ)
period(orbit::Orbit) = period(semimajor_axis(orbit), mass_parameter(orbit.system))
period(orbit::Orbit{Parabolic} = Inf * timeunit(orbit.state)
period(orbit::Orbit{Hyperbolic} = NaN * timeunit(orbit.state)
```

Table of Contents

Orbit Determination

Structures and functions for describing orbits!

Scratchspace

```
Circular Restricted Three-body Orbit

  Synodic Cartesian State:

    t =  0.0 s
    r = [-0.124337 0.378513 1.104311] km
    v = [0.092258 0.773022 -0.580882] km s⁻¹

  Circular Restricted Three-body System (Sun-Mars)

        Length Unit:    2.279e8 km
          Time Unit:    5.933903358476423e7 s
    Mass Parameters:    (1.327124400419393e11, 42828.37362069909) km³ s⁻²
```

```julia
@terminal let system = SunMars

    orbit = Orbit(randn(3), randn(3), system)
    print(orbit)

end
```

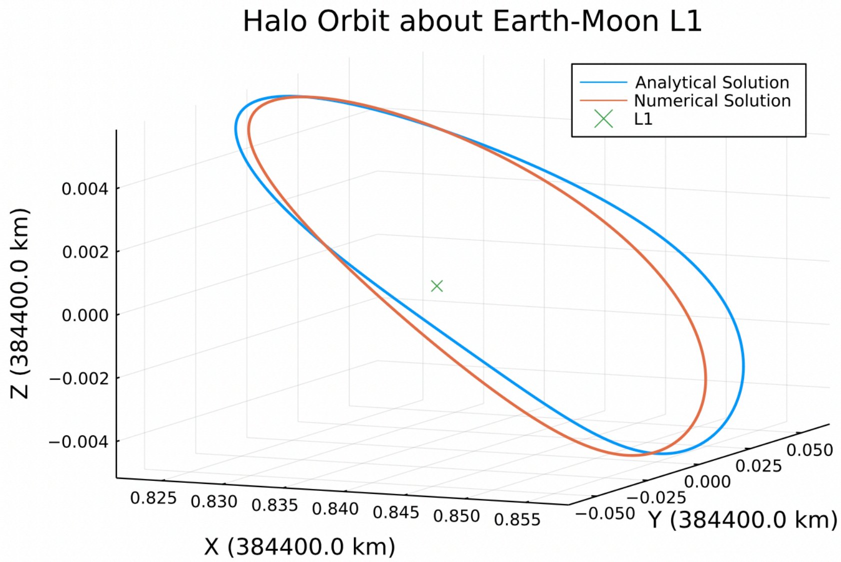
Table of Contents

← →

Periodic Orbits about Lagrange Points

Just like any other dynamics!

System: Earth-Moon **Lagrange Point:** 1 **Z-axis Amplitude:** 0.005



Halo Solver Usage

```julia
# Analytical Approximation (not numerically periodic)
r, v, T  = analyticalhalo(μ; Az = 0.005, L = 1, hemisphere = :northern)

# Numerically Periodic (found with iterative differential correction)
orbit, T = halo(μ; Az = 0.005, L = 2, hemisphere = :southern)
```

Table of Contents