Design & Architecture Report

Jacobi Contour Visualiser

Version: 1.0

Date: 5 December 2025

Contents

1. Introduction

This document provides a comprehensive overview for the CR3BP.html single-page HTML application designed to render the potential fields and Zero Velocity Curves (ZVCs) associated with the Circular Restricted Three-Body Problem (CR3BP). The application utilises HTML5, CSS3, and JavaScript (ES6+), leveraging the D3.js (Data-Driven Documents) library for high-performance Scalable Vector Graphics (SVG) rendering.

The system calculates the scalar field of the Jacobi Constant (C) across a normalised 2D grid, identifies the equilibrium points (Lagrange Points L_1 through L_5), and renders contour lines representing energy states. It features a bidirectional user interface allowing users to manipulate physical constants (mass, distance) or dimensionless parameters (mass ratio μ), with real-time visualisation updates.

By decoupling the mathematical model from the rendering pipeline and using adaptive Level-of-Detail techniques (dynamic zooming, counter-scaling), it allows for the exploration of both macro-scale orbital mechanics (Sun-Jupiter) and micro-scale dynamics (Earth-Moon L1 gateway) within a single unified interface.

2. Mathematical & Physical Foundations

2.1 The Circular Restricted Three-Body Problem (CR3BP)

The application considers the motion of a negligible mass particle under the gravitational influence of two massive bodies (M_1 and M_2) that orbit their common centre of mass in circular trajectories.

2.1.1 Normalised Units

To simplify computation, the system uses a dimensionless system:

- **Total Mass**: $M_1 + M_2 = 1$
- **Distance**: The separation between primaries $R = 1$
- **Angular Velocity**: $\omega = 1$
- **Gravitational Constant**: $G = 1$

The Mass Ratio parameter, μ, is defined as

$$\mu = \frac{M_2}{M_1+M_2}.$$

In the rotating barycentric reference frame:

- **Primary Body (M_1)**: Located at $(-\mu, 0)$ with mass $1 - \mu$.
- **Secondary Body (M_2)**: Located at $(1 - \mu, 0)$ with mass μ.

2.2 The Jacobi Constant (C)

The Jacobi Constant is the sole conserved quantity (integral of motion) in the CR3BP. It relates the particle's position and velocity to its energy relative to the rotating frame.

The pseudo-potential function $\Omega(x, y)$ (assuming planar motion $z = 0$) is defined as

$$\Omega(x, y) = \frac{1}{2}(x^2 + y^2) + \frac{1-\mu}{r_1} + \frac{\mu}{r_2},$$

where r_1 and r_2 are the distances to the primary and secondary bodies,

$$r_1 = \sqrt{(x + \mu)^2 + y^2},$$

$$r_2 = \sqrt{\big(x - (1 - \mu)\big)^2 + y^2}.$$

The Jacobi Constant C is defined as

$$C = 2\Omega(x, y) - V^2,$$

where V is the velocity of the particle in the rotating frame.

2.2.1 Zero Velocity Curves (ZVCs)

The "Lagrange Point zero-velocity curves" are visualised in the application to represent the boundaries where velocity $V = 0$. These define the Hill Regions where a particle with a specific energy C cannot exist. The equation for these curves is

$$C = 2\Omega(x, y) = (x^2 + y^2) + \frac{2(1-\mu)}{r_1} + \frac{2\mu}{r_2}.$$

2.3 Lagrange Points

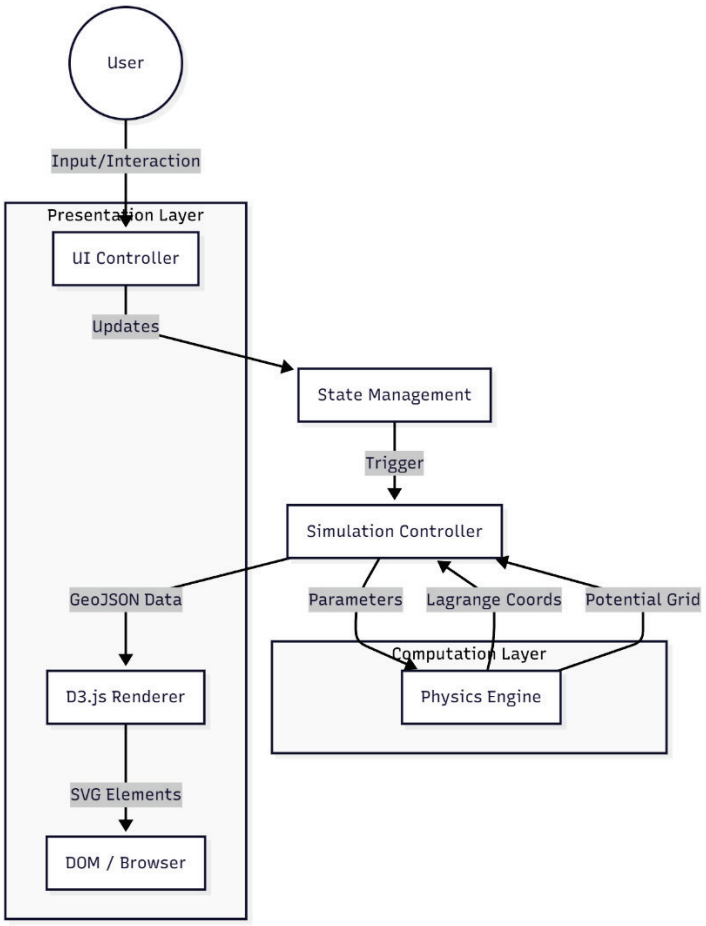
The system calculates the five equilibrium points where the gradient of the potential is zero ($\nabla \Omega = 0$).

- **Collinear Points (L_1, L_2, L_3):** Found along the x-axis ($y = 0$). The script uses the Newton-Raphson method to solve for the roots of the derivative of the potential function.
- **Equilateral Points (L_4, L_5):** Found analytically forming equilateral triangles with the primaries.
  - $x_{L4,L5} = 1/2 - \mu$
  - $y_{L4,L5} = \pm\sqrt{3}/2$

3. System Architecture

The application follows a modular architecture pattern, separating Physics logic, State management, and View rendering.

3.1 High-Level Component Diagram



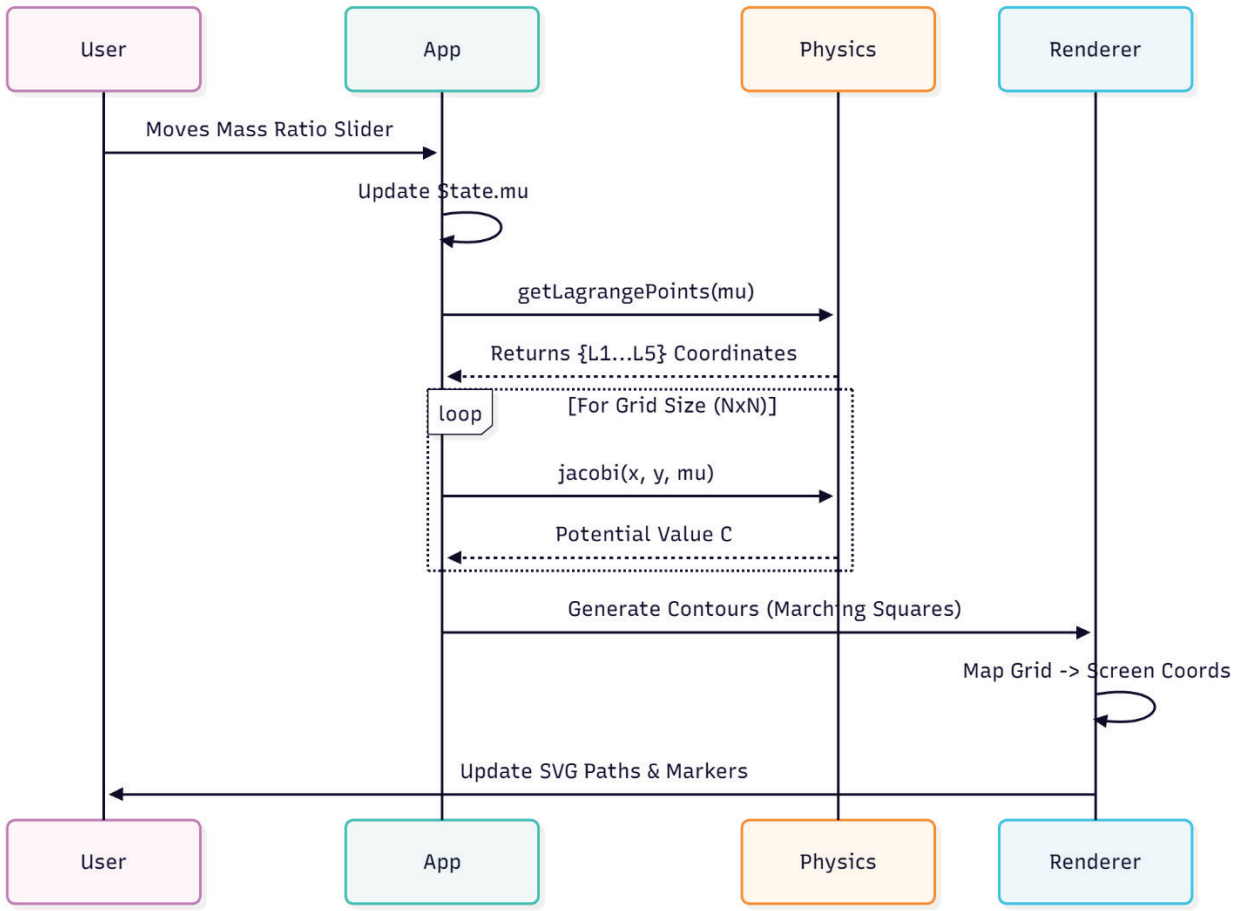

```
graph TD
    User((User)) -->|Input/Interaction| UI[UI
Controller]
    UI -->|Updates| State[State Management]
    State -->|Trigger| Sim[Simulation
Controller]
    Sim -->|Parameters| Physics[Physics Engine]
    Physics -->|Lagrange Coords| Sim
    Physics -->|Potential Grid| Sim
    Sim -->|GeoJSON Data| D3[D3.js Renderer]
    D3 -->|SVG Elements| DOM[DOM / Browser]

    subgraph Computation Layer
    Physics
    end

    subgraph Presentation Layer
    UI
    D3
    DOM
    end
```

3.2 Logic Flow: Simulation Update

When a user changes the Mass Ratio (μ) or applies a Preset:




```
sequenceDiagram
    participant User
    participant App
    participant Physics
    participant Renderer

    User->>App: Moves Mass Ratio Slider
    App->>App: Update State.mu
    App->>Physics: getLagrangePoints(mu)
    Physics-->>App: Returns {L1...L5} Coordinates

    loop For Grid Size (NxN)
        App->>Physics: jacobi(x, y, mu)
        Physics-->>App: Potential Value C
    end
     App->>Renderer: Generate Contours (Marching Squares)
    Renderer->>Renderer: Map Grid -> Screen Coords
    Renderer->>User: Update SVG Paths & Markers
```

4. Detailed Design & Function Definitions

4.1 Global Objects

The script uses three primary namespaced objects to organise code and avoid global scope pollution.

4.1.1 `Physics` Object

Encapsulates all pure mathematical functions derived from orbital mechanics.

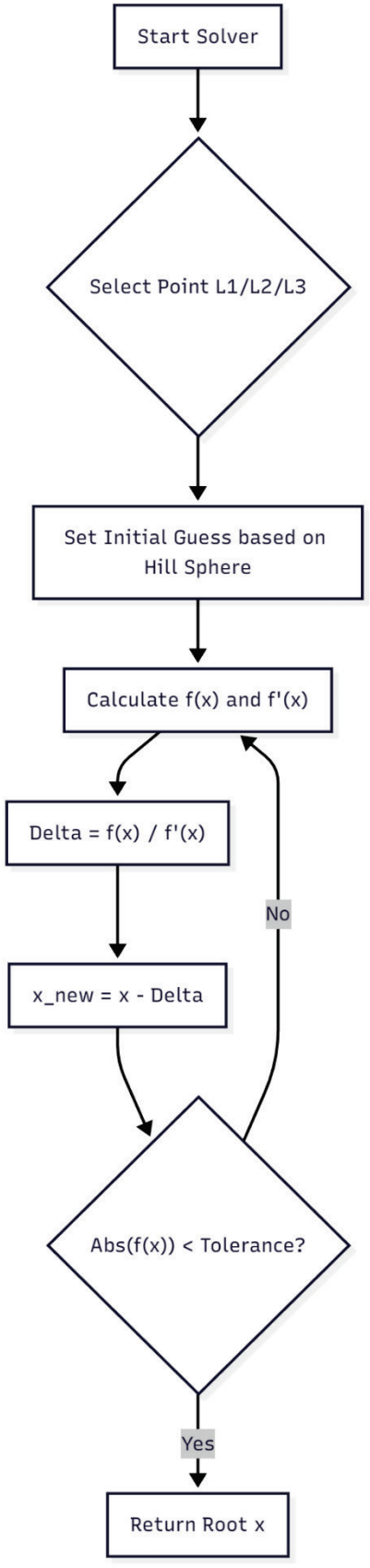
Function	Parameters	Description
`getLagrangePoints`	`mu` (float)	Calculates positions of L1-L5. Uses Newton-Raphson for collinear points. Returns object `{L1:{x,y}, ...}`.
`jacobi`	`x`, `y`, `mu`	Calculates the Jacobi constant at a specific coordinate. Includes a clamp for singularities ($r < 1e^{-5}$).

Newton-Raphson Logic (Internal to `getLagrangePoints`):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where $f(x)$ is the derivative of the effective potential $\partial\Omega/\partial x$.

```
flowchart TD
    A[Start Solver] --> B{Select Point L1/L2/L3}
    B --> C[Set Initial Guess based on Hill Sphere]
    C --> D["Calculate f(x) and f'(x)"]
    D --> E["Delta = f(x) / f'(x)"]
    E --> F[x_new = x - Delta]
    F --> G{"Abs(f(x)) < Tolerance?"}
    G -- No --> D
    G -- Yes --> H[Return Root x]
```


Start Solver
Select Point L1/L2/L3
Set Initial Guess based on Hill Sphere
Calculate f(x) and f'(x)
Delta = f(x) / f'(x)
x_new = x - Delta
Abs(f(x)) < Tolerance?
No
Yes
Return Root x

4.1.2 `State` Object

The single source of truth for the application status.

- `mu`: Current Mass Ratio.
- `transform`: Current D3 Zoom transform (k, x, y).
- `gridSize`: Resolution of the contour grid (e.g., 600x600).
- `contours`: Number of density levels.
- `cMin`, `cMax`: The energy spectrum range for visualisation.

4.1.3 `UI` Object

A cache of HTML DOM elements (Inputs, Sliders, Containers) to prevent repetitive `document.getElementById` calls during the render loop.

4.2 Core Controller Functions

`init()`
Role: Bootstrapper.

Actions:

1. Initialises the D3 SVG context.
2. Creates SVG Groups (`<g>`) for layering: Background, Gates, Bodies.
3. Attaches Event Listeners (Mouse move, Zoom, Input changes).
4. Calls `applyPreset()` to load default data.

`updateSim(fromSlider)`
Role: Orchestrator.

Logic:

1. **Input Handling:** If triggered by the slider, it reverse-calculates the secondary mass (M_2) to maintain consistency. If triggered by text inputs, it recalculates μ.
2. **Physics Calculation:** Calls `Physics.getLagrangePoints`.
3. **Visual Scaling:** Calculates collision-safe radii for the celestial bodies.
   - *Logic:* $R_{secondary}$ is capped at 25% of the distance to L1. $R_{primary}$ is scaled relative to $R_{secondary}$ by mass density approximation ($R \propto M^{1/3}$).
4. **Render Trigger:** Calls `drawSystem`, `updateInfoPanel`, and `generateContours`.

`generateContours(lp)`
Role: The Rendering Engine.

Algorithm:

1. **Grid Population:** Iterates $N \times N$ times over the viewport range (approx. -2.5 to +2.5 normalised units). Calls `Physics.jacobi` for every cell.
2. **Threshold Generation:** Creates an array of energy values between `State.cMin` and `State.cMax`.
3. **Contour Generation:** Uses `d3.contours()` (implementation of Marching Squares) to convert the grid of values into GeoJSON MultiPolygon geometries.
4. **Projection:** Maps the grid coordinates $[0, N]$ to physical coordinates $[-2.5, 2.5]$ via `d3.geoTransform`.
5. **Gate Rendering**:
    - Standard contours act on the scalar field C.
    - **ZVCs:** To avoid rendering a 'box' around the screen (the "infinite" allowed region), the function inverts the field values ($-C$) and contours the "holes" (Forbidden Regions) defined by $[-C_{L1}, -C_{L2}, -C_{L3}]$.

`handleZoom(e)`
Role: Interaction Handler.

Logic:

1. Applies the SVG Transform Matrix (Scale/Translate).
2. **Counter-Scaling:** As the user zooms in (Scale k increases), the script reduces the radius of points and stroke width of text by $1/k$. This ensures labels and markers remain a constant size in screen pixels, regardless of zoom level.
3. **Adaptive Detail:** If $k > 500$, it sets a debounce timer to trigger `generateContours` again with higher fidelity thresholds for the specific viewport.

5. Visualisation & Interfaces

5.1 The Visual Output (SVG)

The visualisation is rendered into an SVG element with specific `vector-effect: non-scaling-stroke` CSS properties. This standard ensures that contour lines remain hairline-thin even when zoomed in by a factor of 200,000.

Layering Hierarchy

1. `g.layer-bg`: The coloured contour map (Turbo colour scheme). Represents the potential field gradient.
2. `g.layer-gates`: Thick red lines representing the Critical ZVCs ($C = C_{L1}, C_{L2}, C_{L3}$).
3. `g.layer-bodies`: Circles for Planets, Lagrange Points, and Text Labels.

5.2 Coordinate Transformations

The tool creates a bridge between Screen Coordinates (Pixels) and Physics Coordinates (Normalised Distance).

Mouse Hover (Tooltip) Transformation:

$$x_{phys} = \frac{x_{mouse} - translate_x}{scale_k}$$

$$y_{phys} = \frac{y_{mouse} - translate_y}{scale_k}$$

Grid Projection (Marching Squares):

$$x_{phys} = x_{min} + \frac{i}{N(x_{max} - x_{min})}$$

6. Standards, Protocols & Dependencies

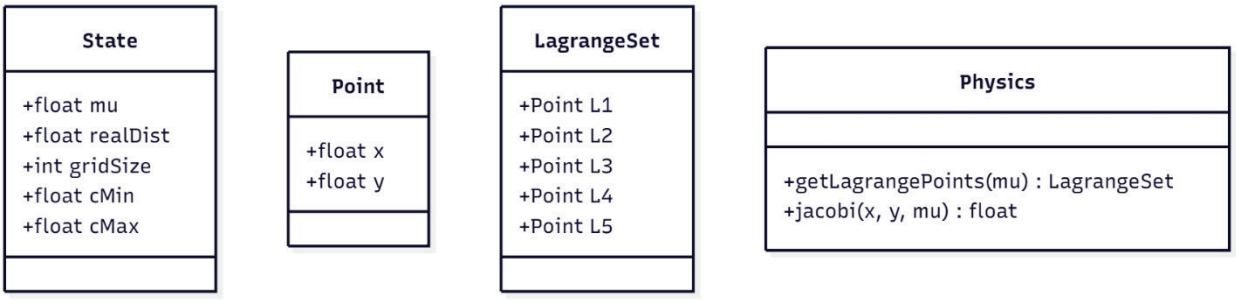
6.1 Standards

- **Language:** UK English is enforced for all UI labels and internal comments (e.g., "Visualiser", "Centre", "Colour").
- **Units**:
  - **Internal:** Dimensionless (Canonical Units).
  - **Display:** Metric (Kilograms, Kilometres).
  - **Large Numbers:** Displayed using "Mn" (Millions) or Scientific Notation where appropriate.

6.2 Dependencies

- **D3.js (v7)**: Used for:
  - `d3-selection`: DOM manipulation.
  - `d3-contour`: Implementation of Marching Squares algorithm.
  - `d3-geo`: Coordinate projection and path generation.
  - `d3-zoom`: Pan and zoom interaction handling.
  - `d3-scale-chromatic`: 'Turbo' colour interpolation.

6.3 Interface Definitions (Types)




```
classDiagram
    class State {
        +float mu
        +float realDist
        +int gridSize
        +float cMin
        +float cMax
    }

    class Point {
        +float x
        +float y
    }
```

```
class LagrangeSet {
        +Point L1
        +Point L2
        +Point L3
        +Point L4
        +Point L5
    }

    class Physics {
        +getLagrangePoints(mu) LagrangeSet
        +jacobi(x, y, mu) float
    }
```