# LunaSim: A Lightweight, Web-Based, Open-Source System Dynamics Modeling Software

Karthik S. Vedula, Sienna Simms, Aditya Patil, Ishan Khetarpal, and Mark R. Estep
Poolesville High School, Poolesville, Maryland, USA

***Abstract*—System dynamics modeling is the process of understanding and representing various elements of a complex system. The majority of system dynamics modeling software are desktop apps, and many web-based alternatives are commercial, lending them unfavorable to educational settings. We developed LunaSim, a lightweight, web-based, system dynamics modeling software to address this gap. LunaSim includes a graphical editor that facilitates the creation of stock and flow diagrams, incorporates JavaScript-based equations for elements in the simulation, simulates using numerical methods, and facilitates the creation of web-based, user-defined charts and tables that display simulation results. We tested this app extensively, and is currently being used by over 60 students.**

***Index Terms*—system dynamics, software development, web app**

## I. Introduction

System dynamics modeling is the process of graphically outlining various elements and their interactions in a complex system (which often represents real-world mechanisms) and simulating them to get theoretical results on how that system will play out over a period of time. Specifically, there are stock-and-flow diagrams, where the user defines stocks, which are elements in a system that will accumulate over time. Flows regulate the accumulation of those stocks; variables/converters allow calculations performed every timestep to be grouped for abstraction; influences/connectors are arrows that display which elements affect which other elements in the simulation.

The ability to create these stock-and-flow diagrams is provided by many software products, such as STELLA [1], Vensim [2], Berkeley Madonna [3], etc. However, many of these software products run as desktop applications with operating system dependencies and require specific installation (which is often time-consuming and require maintenance in institution environments). For example, a system-wide software upgrade made the previous modeling software at our school inoperable on Windows and school-issued Chromebooks, upending students' curriculum. Our software, *LunaSim* was developed to meet this critical need by providing an online alternative that can be accessed from any computer with a standard browser, allowing students to use both the desktop computers and Chromebooks to build their simulations.

## II. Related Work

The majority of system dynamics modeling software are desktop apps. STELLA, Vensim, and Berkeley Madonna all run on the desktop. However, with the rise in popularity in web applications, there has been a lot of development on web-based system dynamics modeling software. STELLA Online [8], Forio [9], and Insight Maker [4], for example, all are web-based options. STELLA Online and Forio are commercial software, while Insight Maker is free and open-source. However, while Insight Maker is feature rich, LunaSim is a simple lightweight tool primarily intended for educational use. It also provides full flexibility by allowing the user to input the equations as JavaScript code, which is favorable as it is a standardized, well-known programming language. Furthermore, LunaSim is also suitable for self-hosting.

Corresponding author: K. S. Vedula (email: karthik@vedula.me).

## III. Methods

We followed industry standard practices to develop LunaSim - developing a Software Requirements Document (SRS), Software Project Management Plan (SPMP), and Software Test Plan (STP). The SRS outlined the requirements for our software to comply with, and these requirements were approved by our client (Teacher) prior to software development. Our software development process, outlined by the SPMP, was a mixture of the iterative and incremental process models. The iterative model consisted of the development and addition of specific functional groups (features) that were added iteratively to the product; this technique was employed for the source code development. The incremental model consisted of refining the product over time, continuously improving the product in response to feedback; this incremental model was employed during the testing phase of the software development process.

We used Git for version control and performed code reviews when merging branches (each feature had its own branch). Each milestone was tagged with a release number.

Each component of the software was unit-tested along with the entire system through benchmark simulations being run on both LunaSim and another system dynamics modeling software. After undergoing beta testing, the app is now being used by over 60 students at Poolesville High School.

## IV. Software Architecture

Like most system dynamics modeling software, LunaSim can be broken into *three* parts: the *model editor* (which is the graphical user interface that allows the creation of the simulation diagram and definition of equations for stocks, flows, and variables), the *simulation engine* (which calculates the values of elements in the simulation over time), and a *data*

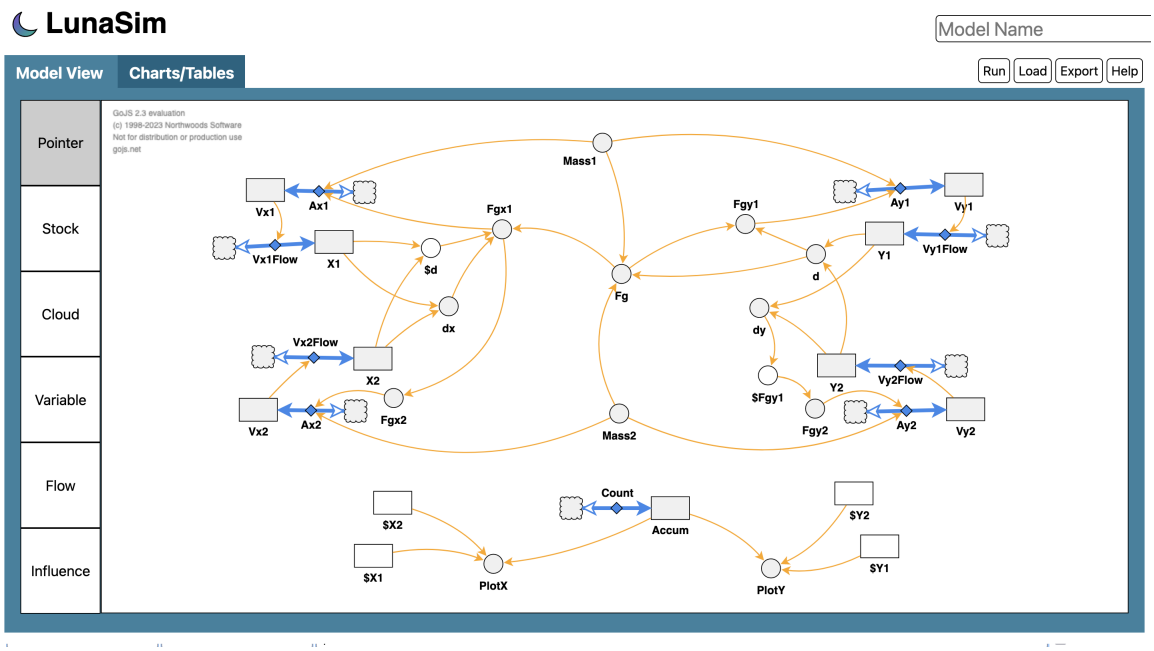


Fig. 1. Sample model that simulates the trajectories of the two bodies in a binary star system.

*visualizer* (which displays user-defined tables and charts). All of these components are written in JavaScript, which all run on the client-side.

### A. Model editor

The model editor facilitates the creation, movement, and deletion of stocks, clouds (essentially stocks of infinite value), flows, variables, and influences. This is done through the GoJS [5] library, which is a library for building interactive diagrams in JavaScript. Specifically, stocks, clouds, and variables are represented as GoJS nodes, while flows and influences are represented as GoJS links. These nodes and links are formatted through GoJS according to standard conventions, with stocks as rectangles, clouds as cloud icons, variables as circles, flows as straight arrows, and influences as curved arrows. GoJS stores this node and link data in JavaScript Object Notation (JSON), and therefore, LunaSim models are stored internally as JSON objects. These models can be saved as JSON files and can be loaded and exported by the user to share their work.

The model editor was built from a GoJS demo which had originally contained the basic functionality of adding and deleting stocks, flows, variables and influences [10]. Features such as double arrowed flows, curvature adjustment for influences, and element name validation were added as enhancements. Fig. 1 shows a sample model with stocks, clouds, variables, and influences in LunaSim.

Ghosting is a feature that is implemented in most system dynamics modeling software. Ghosting is the ability to make a reference to a stock, flow, or variable by making a copy of the shape in the diagram. This makes complex models much more organized, as it prevents influences overlapping with each other since one can create a copy of an element to be much closer to the element that it is supposed to influence. It is important to note that this does not create a copy of the element itself but instead is just a copy of the shape that represents the element. This ghosting feature is implemented in LunaSim by interpreting elements with names that start with a dollar sign ($) as ghosts of the elements whose name are the same except for the dollar sign (e.g. *$stock1* is the ghost of *stock1*).

Equations are entered by the user in a table, with each stock, variable, and flow having an entry. Each entry consists of the associated element's name, type (stock, variable, or flow), and its equation and whether it can be negative (both of which are user-defined through text box inputs and checkboxes respectively). This table is automatically updated if the simulation diagram is modified (i.e. a new element is added, an element's name changed, or if an element was deleted).

This design of a table-based equation editor was chosen over an individual pop-up that is displayed when a specific element was clicked on due to the table's ability to show all equations at once, which can be useful during model debugging. Fig. 2 shows a sample table. The model editor also features a method of entering the start and end time, along with the timestep (*dt*) value.

When the model is run, the nodes and links format is changed into a format in which each stock encapsulates connected flows (see section *Internal File Format Examples* in appendix). Though ghosts are internally represented in the model editor as separate GoJS nodes, the equivalence between ghosts and their original elements is reconciled at this stage. This new format is then sent to the *simulation engine* for calculating results.

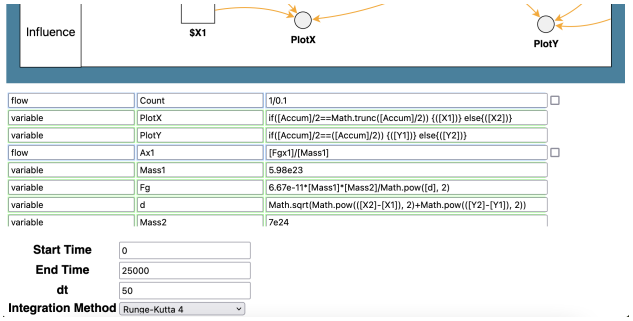


Fig. 2. Equation table which allows user to edit equations of different elements in the simulation.

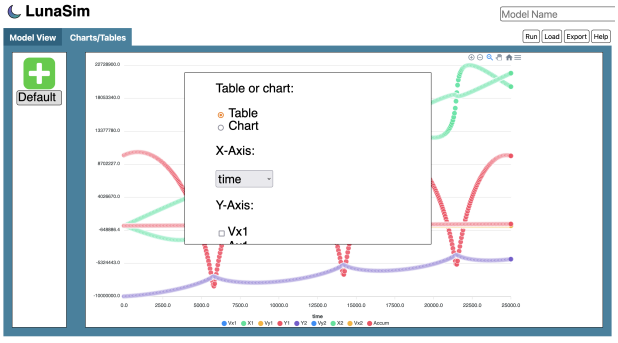


Fig. 3. Pop-up displayed that provides user ability to configure table/plot.

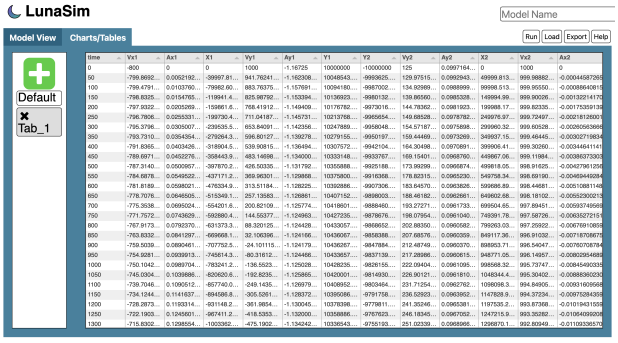


Fig. 4. Results of binary star simulation in user-specified tabular format.

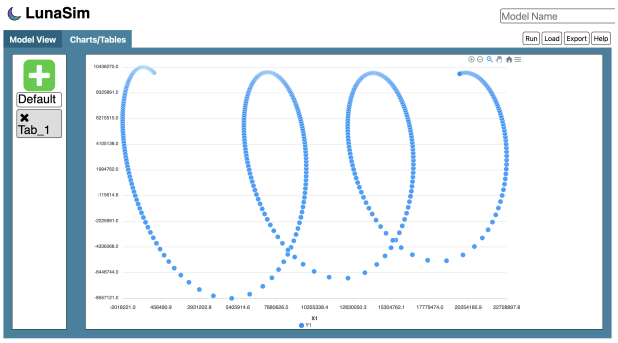


Fig. 5. Results of binary star simulation (Body 1's Y position vs X position) in user-specified scatterplot.

## B. Simulation engine

The simulation engine can be broken down into *two* parts: *equation parsing* and *integration*. Equation parsing involves, for each element's equation, recursively replacing each reference of another element with its equation. Each reference is denoted by the element's name enclosed in brackets. For example, if the equation of *stock1* is *[stock2] + 2*, then the parsed version would replace *[stock2]* with the parsed version of *stock2*'s own equation. If an equation does not have any further references, it is simply returned. This recursive function ensures, as long as there are not any circular definitions, that each parsed equation can be directly evaluated by the JavaScript `eval()` function. If the user writes an invalid equation, a JavaScript runtime error is thrown, which is reformatted into an error that is shown to the user for model debugging. See Fig. 6 for a sample error popup.

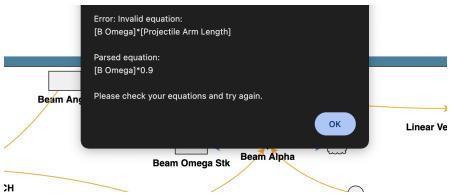


Fig. 6. Example of an error shown to the user.

LunaSim provides two integration methods to be used for running the simulation: *Euler's method* and *Runge-Kutta 4*. Integration requires taking each flow's parsed equation (which inherently requires any dependent equations to be parsed) and updating the connected stock values accordingly. In Euler's method, the flow value is multiplied by *dt* and then added to the stock value. In Runge-Kutta 4, flow values of intermediate timesteps are factored in and then added to the stock value.

The advantage of LunaSim's simulation engine is the fact that equations are directly put through the `eval()` function, which allows the user to leverage the JavaScript language and its libraries to make complex equations. For example, all the mathematical functions (trigonometric, logarithmic, etc) are provided by the `Math` namespace. If-and-else statements can simply be written in JavaScript as `if (...) {...} else {...}`. The JavaScript methods `alert()` and `prompt()` can be included in equations to show and ask for user input when the simulation is run. This functionality also opens up the possibility of having the simulation query for data through HTTP APIs as well, which we leave for future work.

## C. Data visualization

Like most system dynamics modeling software, LunaSim allows the user to create custom charts and tables, where the user specifies which elements' (specifically stocks, flows, variables) simulation data are displayed. LunaSim displays these tables and charts as tabs, where the user can select a tab to have its associated chart/table to be displayed. Charts are created using the *ApexCharts* library [6], and tables are created using the *Tabulator* [7] JavaScript library. Fig. 3, 4, and 5 show samples of this functionality.

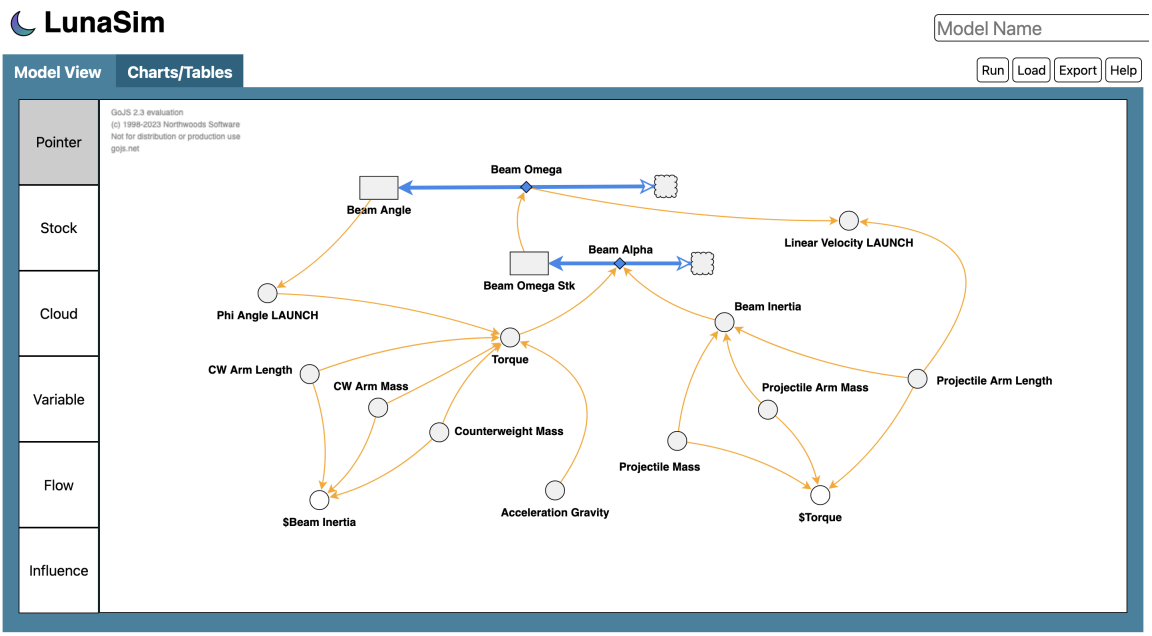


Fig. 7. Sample model that simulates mechanics of a catapult.

## V. Testing

### A. Unit & Integration Testing

Each requirement developed in the SRS was covered by test cases generated in the STP, and tests were organized according to the specific feature they are assessing. In order to test the application fully, we created and ran two simulation models: the *catapult simulation* and the *binary star system simulation*.

The catapult model simulates a see-saw-like catapult. The model splits the catapult into the counterweight, the section of the beam on the side of the counterweight, the section of the beam on the side of the projectile, and the projectile itself. Given an initial beam angle, the model calculates the launch velocity (providing launch speed and angle) of a projectile if the projectile were to leave the catapult at that timestep. This can be used to calculate the optimum angle at which the projectile should be launched. See Fig. 9 for results.

The binary star model simulates the trajectories of two planetary objects that exert a gravitational force on each other. The model breaks the system down into the position, velocity, acceleration, and force (in both $x$ and $y$ directions) exerted by each planetary object. See Fig. 4 and 5 for results.

The equations used to create these two simulations are in the appendix of this paper.

## VI. Operational Deployment

### A. Catapult Project

Over 60 students have been using LunaSim after it was integrated into Poolesville High School's 9th Grade curriculum. Students were tasked to build a miniature catapult to launch a projectile onto a target. Prior to building the catapult, students were assigned to prototype small-scale concepts and to use computer-aided design (CAD) software to plan their catapult design (see Fig 8). After designing the catapult in CAD, students designed a stock-and-flow model (similar to the model shown in Fig. 7) in LunaSim to model the *mechanics of the catapult*. Specifically, they were tasked to create a model that, given the parameters such as the beam, counterweight, and projectile weights, calculates the velocity of the projectile if it were to leave the catapult at a certain timestep. Students then created a stock-and-flow model that modeled a *projectile's trajectory* (given the initial mass and velocity) that factored-in air resistance (by using drag constant calculated empirically by students) to simulate how far their catapult will launch the projectile. Students then built the catapult they designed in CAD and simulated in LunaSim and compared theoretical to actual results.

This project was taught in the 9th Grade Magnet Computer Science course where it provided interdisciplinary learning by combining Mathematics, Physics, Computer Science, and Engineering concepts.

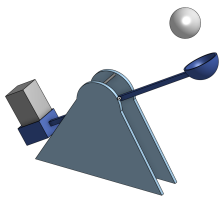

Fig. 8. Sample CAD design made by students.

### B. Hosting and Compatible Platforms

The app is hosted on an Amazon Web Services S3 Storage Bucket, as LunaSim runs fully on the client side (which makes

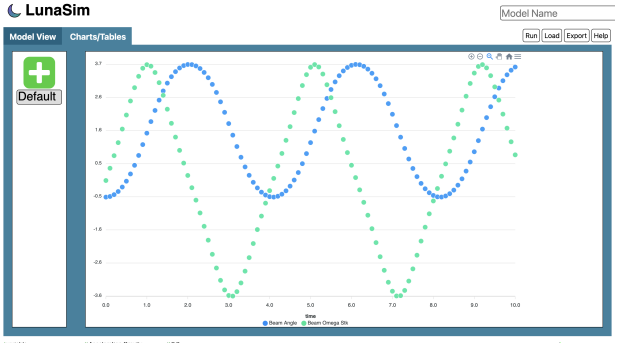


Fig. 9. Results of catapult simulation in user-specified scatterplot. Beam angle (blue) and Beam Angular Velocity (green) vs. time.

it very easy to deploy). The students typically access this app through their Chromebooks but has been tested on Windows, Mac, and iPad as well; browser support includes Chrome, Firefox, Safari, and Edge.

## VII. Conclusion

We developed a web-based system dynamics modeling software that uses a JavaScript-based equation definition system that can be easily deployed due to it being a fully client-side app. We tested this app extensively and the app has been used in educational settings to teach system dynamics modelling.

## VIII. Code Availability

LunaSim's code is available on GitHub at **`https://github.com/oboy-1/LunaSim`**. LunaSim uses *unpkg* to load dependencies.

## References


[1] *STELLA Architect*. Version 2.0.3, ISEE Systems. Available: https://www.iseesystems.com/store/products/stella-architect.aspx

[2] *Vensim*. Ventana Systems. Available: https://vensim.com/

[3] *Berkeley Madonna*. Berkeley Madonna. Available: https://berkeley-madonna.myshopify.com/

[4] *Insight Maker*. S. Fortmann-Roe, 'Insight Maker: A general-purpose tool for web-based modeling & simulation', Simulation Modelling Practice and Theory, vol. 47, pp. 28–45, 2014. Available https://insightmaker.com/

[5] *GoJS*. Version 2.3. Northwoods Software. Available: https://gojs.net/latest/index.html

[6] *ApexCharts*. ApexCharts. Available: https://apexcharts.com/

[7] *Tabulator*. Version 5.5, Tabulator. Available: https://tabulator.info

[8] *STELLA Online*. ISEE Systems. Available: https://www.iseesystems.com/store/products/stella-online.aspx

[9] *Forio*. Forio. Available: https://forio.com/

[10] R. Muetzelfeldt. GoJS System Dynamics demo. Available: https://gojs.net/latest/samples/systemDynamics.html


## IX. Acknowledgments


The authors are grateful to Northwoods Software for permitting the use of GoJS for free with LunaSim for non-commercial educational purposes.


## Appendix

Note that in a system of equations, each equation is evaluated for each timestep, including variables and flows. Therefore, in an equation such as $\alpha = \frac{T_{total}}{I_{total}}$, $\alpha$ can be thought of as $\alpha_t$, where $t$ is the given timestep.

### A. Catapult Model Equations

- Initial Angle (v[1]): $\phi$
- Angular Position (s): $\theta_0 = \frac{\pi}{2} - \phi, \theta_{t+1} = \theta_t + \omega_{flow} \cdot dt$
- Angular Velocity Flow (f): $\omega_{flow} = \omega$
- Angular Velocity (s): $\omega_0 = 0, \omega_{t+1} = \omega_t + \alpha \cdot dt$
- Angular Acceleration (f): $\alpha = \frac{T_{total}}{I_{total}}$
- Total MoI[2] (v): $I_{total} = I_{CWBeam} + I_{CW} + I_p + I_{pBeam}$
- Total Torque (v): $T_{total} = T_{CWBeam} + T_{CW} + T_p + T_{pBeam}$
- Projectile MoI (v): $I_p = M_p \cdot L_{pBeam}^2$
- Projectile Beam MoI (v): $I_{pBeam} = \frac{1}{3} \cdot M_{pBeam} \cdot \left(\frac{L_{pBeam}}{2}\right)^2$
- Counterweight MoI (v): $I_{CW} = M_p \cdot L_{pBeam}^2$
- Counterweight Beam MoI (v): $I_{CWBeam} = \frac{1}{3} \cdot M_{CWBeam} \cdot \left(\frac{L_{CWBeam}}{2}\right)^2$
- Mass of Projectile (v): $M_p$
- Mass of Counterweight (v): $M_{CW}$
- Mass of Beam (Projectile side) (v): $M_{pBeam}$
- Mass of Beam (Counterweight side) (v): $M_{CWBeam}$
- Length of Beam (Projectile side) (v): $L_{pBeam}$
- Length of Beam (Counterweight side) (v): $L_{CWBeam}$
- Projectile Torque (v): $T_p = F_p \cdot L_{pBeam} \cdot \cos(\theta)$
- Projectile Beam Torque (v): $T_{pBeam} = F_{pBeam} \cdot L_{pBeam} \cdot \cos(\theta)$
- Counterweight Torque (v): $T_{CW} = F_{CW} \cdot L_{CWBeam} \cdot \cos(\theta)$
- Counterweight Beam Torque (v): $T_{CWBeam} = F_{CWBeam} \cdot L_{CWBeam} \cdot \cos(\theta)$
- Force for object $x$ (v): $F_x = 9.8 \cdot M_x$
- Launch Degrees (v): $\theta_{degrees} = \left(\frac{\pi}{2} + \theta\right) \cdot \frac{180}{\pi}$
- Launch Speed (v): $v_{launch} = -\omega \cdot L_{pBeam}$

### B. Binary Star Model Equations

- Mass of Body $n$ (v): $M_n$
- X-Position of Body $n$ (s): $x_{(n)0} = 0, x_{(n)t+1} = x_{(n)t} + v_{x(n)flow} \cdot dt$
- Y-Position of Body $n$ (s): $y_{(n)0} = 0, y_{(n)t+1} = y_{(n)t} + v_{y(n)flow} \cdot dt$
- X-Velocity Flow of Body $n$ (f): $v_{x(n)flow} = v_{x(n)}$
- Y-Velocity Flow of Body $n$ (f): $v_{y(n)flow} = v_{y(n)}$
- X-Velocity of Body $n$ (s): $v_{x(n)t+1} = v_{x(n)t} + a_{x(n)} \cdot dt$
- Y-Velocity of Body $n$ (s): $v_{y(n)t+1} = v_{y(n)t} + a_{y(n)} \cdot dt$
- X-Acceleration of Body $n$ (f): $a_{x(n)} = \frac{F_{gx(n)}}{M_n}$
- Y-Acceleration of Body $n$ (f): $a_{y(n)} = \frac{F_{gy(n)}}{M_n}$
- X-Force Exerted by Body 1 (v): $F_{gx1} = -\frac{F_g}{D} \cdot D_x$
- Y-Force Exerted by Body 1 (v): $F_{gy1} = -\frac{F_g}{D} \cdot D_y$
- X and Y-Force Exerted by Body 2 (v): $F_{gx2} = -F_{gx1}, F_{gy2} = -F_{gy1}$
- Gravitational Force on Bodies 1 and 2 (v): $F_g = G \cdot \frac{M_1 \cdot M_2}{d^2}$
- Distance between planets (v): $D = \sqrt{D_x^2 + D_y^2}$
- X and Y-Distance between planets (v): $D_x = x_1 - x_2, D_y = y_1 - y_2$
- Gravitational Constant (v): $G = 6.67 \cdot 10^{-11}$

[1]The letter in parentheses before the colon in each equation signifies whether the value is represented as a stock, flow, or variable.

[2]MoI: Moment of Inertia

### C. Internal File Format Examples

```
{
  "nodeDataArray" : [
    {
      "category":"stock",
      "label":"Population",
      "location":"...",
      * Fields on whether stock is
          nonnegative, stock's equation,
          etc *
    },

    {"key":"cloud1"...}
    ...
  ]

  "linkDataArray" : [
    {"category":"flow","text":"flow","from
        ":"cloud1","to":"stock1"...}
    ...
  ]

  * Simulation parameter fields - dt,
      start and end time, integration
      method *
}
```

Listing 1. The sample JSON file above is in the "nodes and links format," where all elements in the model are stored either as nodes (stocks and converters) or as links (flows and influences). Note that this is a simplified version of the GoJS format used in LunaSim.

```
{
  "stocks" : {
    "population" : {
       * Fields on whether stock is
           nonnegative, stock's equation,
           etc *
      "inflows" : {
        "births" : {
          * Fields on whether flow is
              uniflow/biflow, flow's
              equation, etc *
        }
      }
      "outflows" : {
        "deaths" : {
          * Fields on whether flow is
              uniflow/biflow, flow's
              equation, etc *
        }
      }
    }
  }

  "converters" : {
    "birthrate" : {
      * Equation field *
    }
    "mortalityrate" : {
      * Equation field *
    }
  }

  * Simulation parameter fields - dt,
      start and end time, integration
      method *

}
```

Listing 2. The sample JSON file above has stocks encapsulating the flows that connect to them.