|  |  |
| --- | --- |
|  | NBodyUI |

***An N-Body code***

***Author: Jeff Balsley***

Table of Contents

[1 Introduction 2](#_Toc289100700)

[2 User Interface 3](#_Toc289100701)

[2.1 ODE Integration Method 3](#_Toc289100702)

[2.2 Energy 3](#_Toc289100703)

[2.3 Input Parameters 4](#_Toc289100704)

[2.4 Bodies 4](#_Toc289100705)

[2.5 Time 5](#_Toc289100706)

[2.6 Output Box 6](#_Toc289100707)

[3 NBodyUI Class Library 6](#_Toc289100708)

[3.1 PhysicalConstants Class 6](#_Toc289100709)

[3.2 Body Class 8](#_Toc289100710)

[3.3 Energy Struct 10](#_Toc289100711)

[4 NBodyLib Miscellaneous APIs 10](#_Toc289100712)

[4.1 CalculatePotentialEnergy 10](#_Toc289100713)

[4.2 TotalEnergy 10](#_Toc289100714)

[4.3 GravitationallyBound 11](#_Toc289100715)

[4.4 GravitationallyBound (Overloaded) 11](#_Toc289100716)

[4.5 EllipseVelocityAtAphelionX 12](#_Toc289100717)

[4.6 EllipseVelocityAtAphelionY 12](#_Toc289100718)

[5 ODE Integration Method 13](#_Toc289100719)

[5.1 Reducing Second-Order ODEs 13](#_Toc289100720)

[5.2 Leapfrog Integration 14](#_Toc289100721)

[6 Calculating Force 14](#_Toc289100722)

[7 Appendix 15](#_Toc289100723)

## Introduction

The Astronomical N-Body problem is centuries old. As early as the late 1800s, Sweden’s King Oscar II created a prize for anyone who could solve the problem:

“Given a system of arbitrarily many mass points that attract each according to Newton's law, under the assumption that no two points ever collide, try to find a representation of the coordinates of each point as a series in a variable that is some known function of time and for all of whose values the series converges uniformly.”

NBodyUI is a graphical two-dimensional N-body simulator written in C# capable of attempting a solution. This application concentrates solely on the solar system and does not include gravitational interactions between other stars or galaxies. It includes the Sun and the five innermost planets: Mercury, Venus, Earth, Mars, and Jupiter. In addition, the user may add any number of comets and asteroids as well.

This document discusses the NBodyUI User Interface and its gravitational force algorithm, see sections 2 and 6. The integration method used for the equations of motion are examined in detail in section 5. Finally, the class library NBodyLib objects and APIs is discussed in sections 3 and 4.

## User Interface

Unlike other N-Body simulators, the primary goal of this application was to create a user-friendly interface from which to control the simulation. Other N-Body applications, (see appendix), are generally command line based and include complicated initialization files. NBodyUI is purely GUI based. This section outlines the simple UI controls.

### ODE Integration Method

NBodyUI supports three common ODE numerical integration methods as shown in Figure 1. Although the three offer similar results, the leapfrog method is the default and offers the best performance.

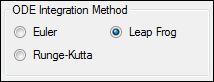


Figure 1 - Selecting different integration methods

### Energy

During any N-Body simulation, it’s paramount to monitor the total energy of the system. In brief tests, the total energy, or TE in Figure 2, stays constant to better than one part in 10,000.

When a simulation is started, both the kinetic energy and total energy are calculated in joules or ergs and displayed in the KE and PE text boxes shown in Figure 2. The sum of the two, the total energy, is displayed in the text box labeled TE.

For circular orbits, the potential energy is always -2× the kinetic energy. This will be easily noted if one is simply including planets in a simulation. If the average kinetic energy grows greater than the average magnitude of the potential energy, the two objects escape and will no longer be in orbit. Therefore, when monitoring the stability of the celestial system, click the box labeled ‘Stop if Energy > 0’ if you wish to stop the simulation if the energy becomes positive. With each update interval, shown in Figure 3, the application will check if the TE is greater than zero. If so, the internal ‘for loop’ will ‘break’ and pop up a message stating the energy has grown positive.

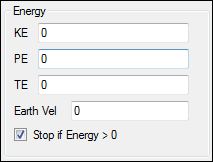


Figure 2 - Energy Panel at launch of application

### Input Parameters

The most important input parameters are shown in Figure 3, timestep, update interval, and runtime. The timestep is the length of time between each force calculation. Because this is a physical system, the timestep is not simply a counting variable inside a ‘for loop’. It has the units of seconds and corresponds to the , or *h,* in the equations introduced in Section 4.

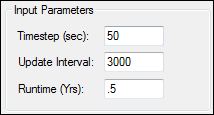


Figure 3 - Input Parameters

The update interval corresponds to the amount of time between each UI update. The default interval is 50,000, as shown in Figure 3. This means every 50,000 seconds in simulation time, the object’s position will be updated in the picture box. Since the default time step is 50, the UI will be updated every 1000 times through the main ‘for loop’.

The runtime is in years, as opposed to the other two input parameters which are in seconds. This parameter indicates where you indicate the length of time you wish your simulation to run. The default is one year of simulation time which, for about 10 objects, lasts about 1 minute of real time using the default parameters.

### Bodies

Eight unique bodies can be included in each simulation. This includes the Sun and the five innermost planets, Mercury, Venus, Earth, Mars, and Jupiter. You may also include any number of comets and asteroids. Be forewarned however, increasing the number of bodies, or N, can severely slow down the simulation since the force calculation is proportional to N2 as discussed later in Section 5.

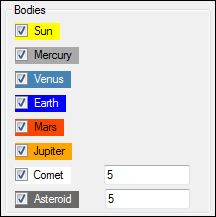
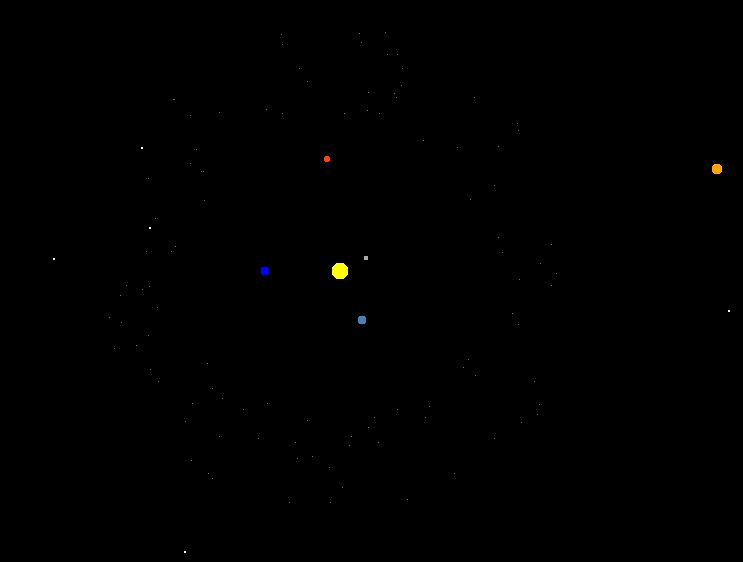


Figure 4 - Included Bodies in a default N-Body simulation

Each object shown in Figure 4 has been uniquely color-coded. Not shown in Figure 4 is the size of each object. The larger ones, such as the Sun and Jupiter, appear larger than the other planets though the sizes are not necessarily proportional. The comets and asteroids are drawn one to two pixels in diameter so as not to be confused with the larger planets. See Figure 5.



**Figure 5 – Sample simulation in NBodyUI. One can clearly find the four inner planets,**

**the asteroid belt, several comets, and Jupiter.**

### Time

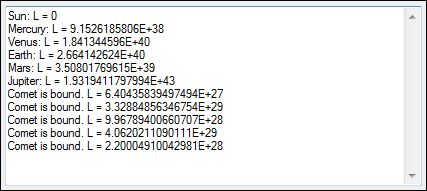
The time box in Figure 6 will display the amount of simulation time that has passed in months and years. After a simulation has finished, the duration will be displayed in real time. For example, if the Earth orbits the Sun in one minute, the simulation time would be one year while the real time is 60 seconds.



Figure 6 - Time output

### Output Box

The output box is saved for specific text output such as information or warnings concerning non-fatal errors. In Figure 7, the textbox is displaying the angular momentum of each object in the N-Body simulation as well as information on orbits of the five comets.



**Figure 7 - Output Box**

## NBodyUI Class Library

NBodyUI.exe relies heavily on the class library, NBodylib.dll, which is public and open to all developers to use and inherit from. Currently, it contains two classes and one struct. The PhysicalConstants class is filled with all the physical and astronomical constants needed to write an N-Body or other astronomical application. With the Body class, a developer can instantiate a body object for each of the astronomical objects in the simulation. The energy struct is used to track the kinetic, potential, and total energy of the simulated system.

### PhysicalConstants Class

This class is filled with astronomical and physical constants that will be referenced repeatedly in the N-Body code. All the fields are readonly and public and a developer must create an instance of the class to reference a constant. This is because the constants are not initialized by default since the class is not aware of which unit system will be used.

When creating an instance of the class, a developer must pass either “mks” or “cgs” strings to the constructor. “Mks” stands for meters/kilograms/seconds and is the most common unit system used by scientists. However, “cgs”, or centimeters/grams/seconds is the preferred unit system used by astronomers.

To instantiate a class, developers should do the following in their code:

PhysicalConstants Constants = new PhysicalConstants("mks");

The library will set the readonly physical constants as follows:

else if (String.Equals(units, "MKS", StringComparison.OrdinalIgnoreCase))

{

GRAVITATIONAL\_CONSTANT = 6.67e-11; // Nm^2 / kg^2 Gravitational constant

SPEED\_OF\_LIGHT = 2.99792458e8; // m/s speed of light

e\_ = 1.60e-19; // Coulombs Charge of electron

ASTRONOMICAL\_UNIT = 1.496e11; // Astronomical Unit; m

PARSEC = 3.086e16; // Parsec; m

LIGHT\_YEAR = 9.46e15; // light year; m

MASS\_EARTH = 5.98e24; // Mass of Earth; kg

RADIUS\_EARTH = 6.378e6; // Radius of Earth; m

MASS\_SUN = 1.99e30; // Mass of Sun; kg

MASS\_MOON = 7.3477e22; // Mass of moon; kg

MASS\_MERCURY = 3.3022e23; // Mass; kg

MASS\_VENUS = 4.8685e24; // Mass; kg

MASS\_MARS = 6.4185e23; // Mass; kg

MASS\_JUPITER = 1.8986e27; // Mass; kg

MERCURY\_ORBIT\_SPEED = 47870; // Average orbital speed; m/s

VENUS\_ORBIT\_SPEED = 35020; // Average orbital speed; m/s

EARTH\_ORBIT\_SPEED = 29780; // Average orbital speed; m/s

MARS\_ORBIT\_SPEED = 24077; // Average orbital speed; m/s

JUPITER\_ORBIT\_SPEED = 13070; // Average orbital speed; m/s

MERCURY\_ORBIT = 5.79e10; // Ave Orbit; m

VENUS\_ORBIT = 1.08e11; // Ave Orbit; m

MARS\_ORBIT = 2.27e11; // Ave Orbit; m

JUPITER\_ORBIT = 7.78547e11; // Ave Orbit; m

RADIUS\_MOON = 1.738e6; // Radius of Moon; m

MOON\_ORBIT\_RADIUS = 3.84e8; // Radius of Moon Orbit; m

AVERAGE\_MOON\_ORBIT\_SPEED = 1023; // Average orbital speed of Moon; m/s

MASS\_HALLEYS\_COMET = 2.2e14; // Mass of Halley's comet; kg

}

When including the library in a third party N-Body application, remember to use the library namespace:

using NBodyLib;

Developers will need to add a reference to the NBodyLib.dll to compile the project. If familiar with Visual Studio, simply go to Project -> Add Reference and search for the .dll. Or right-click the project name in the class view panel and select Add Reference. The NBodyLib can be found under the list of project references in the class view as shown in Figure 8.

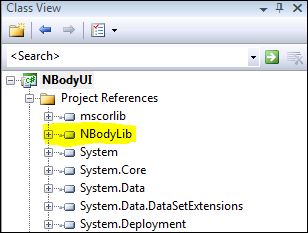


Figure 8 - Project References

Developers should now be able to use any of the constants in the class.

### Body Class

The body class contains all the properties of a celestial object needed to include in an N-Body simulation.

#### Name

This is the name of an object. The body can be unique such as “Sun” or “Jupiter” or one of many bodies, such as “Comet” or “Asteroid”.

#### X

The total NBodyUI package is for two dimensional simulations only. This property is used for the *x*-coordinate in the *x-y* plane. The units are either meters or centimeters.

#### Y

The total NBodyUI package is for two dimensional simulations only. This property is used for the *y*-coordinate in the *x-y* plane. The units are either meters or centimeters.

#### Vx

The total NBodyUI package is for two dimensional simulations only. This property is used for the *x*-component of the velocity vector in the *x-y* plane. Units are either meters/sec or centimeters/sec.

#### Vy

The total NBodyUI package is for two dimensional simulations only. This property is used for the *y*-component of the velocity vector in the *x-y* plane. Units are either meters/sec or centimeters/sec.

#### Mass

This is the mass of the object. The units specified in your PhysicalConstants object (kilograms or grams) must be used.

#### Fx

This is the gravitational force in the *x*-direction *on the object*. It will be used when updating the velocity of the body. Units will be Newtons if using mks units and dynes if you selected cgs.

#### Fy

This is the gravitational force in the *y*-direction *on the object*. It will be used when updating the velocity of the body. Units will be Newtons if using mks units and dynes if you selected cgs.

#### Electric Charge

This is currently not implemented.

#### LastPrintedX

This is the *x*-coordinate of the last location the body was drawn in the picture window.

#### LastPrintedY

This is the *y*-coordinate of the last location the body was drawn in the picture window.

#### KineticEnergy

This is the kinetic energy of the body in Joules (mks) or ergs (cgs).

#### Angular Momentum

This method returns the angular momentum of the body. It assumes the object is orbiting the sun located at *x*=0, *y*=0.

#### Constructor

The Body class needs six items to instantiate a Body:

* The Body mass
* Initial *x*-position
* Initial *y*-position
* Initial *x*-velocity
* Initial *y*-velocity
* A name for the celestial body

Here is an example creating the Sun in the NBodyUI code located in the center of the *x-y* plane:

new Body(Constants.MASS\_SUN, 0, 0, 0, 0, "Sun")

And here is an example of a comet being placed at 7 AU with an initial orbital speed equal to Jupiter’s:

Body Comet = new Body(Constants.MASS\_HALLEYS\_COMET,

7\*Constants.ASTRONOMICAL\_UNIT,

0,

0,

Constants.JUPITER\_ORBIT\_SPEED,

"Comet");

### Energy Struct

The energy structure is used in the code to track the total energy of the system. The structure’s constructor can be used to initialize the system energy.

EnergyStruct Energy = new EnergyStruct(0,0,0);

public struct EnergyStruct

{

public double Kinetic;

public double Potential;

public double Total;

public EnergyStruct(double KE, double PE, double TE)

{

Kinetic = KE;

Potential = PE;

Total = TE;

}

}

## NBodyLib Miscellaneous APIs

NBodyUI uses dozens of useful APIs many of which are also included in the public library, NBodylib. Developers are free to use these at their discretion; six are discussed below.

### CalculatePotentialEnergy

Calculates the potential energy of an entire system of Body objects.

public double CalculatePotentialEnergy

(

List<Body> Bodies

)

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| Bodies | List<Body> | A List of Body objects representing each celestial body in an N-Body system. |

#### Return Value

The API returns a System.Double representing the potential energy of the system in joules or ergs, depending on which unit system was specified in the PhysicalConstants class.

### TotalEnergy

Calculates the total energy (both potential and kinetic) of an entire system of Body objects.

public double TotalEnergy

(

List<Body> Bodies,

ref EnergyStruct Energy

)

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| Bodies | List<Body> | A List of Body objects representing each celestial body in an N-Body system. |
| Energy | ref EnergyStruct | A reference to an EnergyStruct. |

#### Return Value

The API returns a System.Double value representing the total energy of the system.

### GravitationallyBound

This API is used to check whether or not an object is gravitationally bound to the solar system. Objects that escape the gravitational bonds of the Sun can be dynamically removed from the List of Body objects which will make the force calculation less computationally taxing.

public int GravitationallyBound

(

double mass,

double velocity,

double orbitRadius

)

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| Mass | System.Double | Mass of object in kilograms or grams. |
| Velocity | System.Double | Velocity of the object in m/s or cm/s. |
| orbitRadius | System.Double | Orbital radius or object. |

#### Return Value

The API returns a System.Int32 value of 1, 0, or -1.

|  |  |
| --- | --- |
| **Return Value** | **Description** |
| 1 | Object is gravitationally bound with its KE < |PE|. |
| 0 | Object is bound in a circular orbit with its KE = |PE|. |
| -1 | Object is not gravitationally bound, with its KE > |PE|. |

### GravitationallyBound (Overloaded)

This overloaded API is used to check whether or not an object is gravitationally bound to the solar system. Objects that escape the gravitational bonds of the Sun can be dynamically removed from the List of Body objects which will make the force calculation less computationally taxing.

public double GravitationallyBound

(

Body b

)

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| b | NBodylib.Body | Body object declared in NBodylib |

#### Return Value

The API returns a System.In32 value of 1, 0, or -1.

|  |  |
| --- | --- |
| **Return Value** | **Description** |
| 1 | Object is gravitationally bound with its KE < |PE|. |
| 0 | Object is bound in a circular orbit with its KE = |PE|. |
| -1 | Object is not gravitationally bound, with its KE > |PE|. |

### EllipseVelocityAtAphelionX

When initially populating an N-Body system with objects, it is useful to know the velocity of an object at aphelion. This API will return the *x*-component of the velocity given the initial position and eccentricity *e*.

public double EllipseVelocityAtAphelionX

(

double X,

double Y,

double e

)

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| X | System.Double | The *x*-coordinate of the celestial object. |
| Y | System.Double | The *y*-coordinate of the celestial object. |
| e | System.Double | The eccentricity of the elliptical orbit. |

#### Return Value

The API returns a System.Double value representing the velocity in the *x*-direction. It can be either positive or negative.

### EllipseVelocityAtAphelionY

When initially populating an N-Body system with objects, it is useful to know the velocity of an object at aphelion. This API will return the *y*-component of the velocity given the initial position and eccentricity *e*.

public double EllipseVelocityAtAphelionY

(

double X,

double Y,

double e

)

#### Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| X | System.Double | The *x*-coordinate of the celestial object. |
| Y | System.Double | The *y*-coordinate of the celestial object. |
| e | System.Double | The eccentricity of the elliptical orbit. |

#### Return Value

The API returns a System.Double value representing the velocity in the *y*-direction. It can be either positive or negative.

## ODE Integration Method

Countless methods for integrating differential equations have been developed. The Euler method, developed by Leonhard Euler over 200 years ago is a simple and easily comprehensible integration method. It is sometimes referred to as the tangent line method because it relies strictly on the tangent, or derivative, of a curve at point x1 to predict the value at x2. However, its accuracy is poor and is called into question when applied to most engineering problems.

The Runge-Kutta method is widely seen as the default numerical integration method for engineering problems. It is far more accurate than the somewhat simplistic Euler method but is more computationally expensive. While very precise, over long scales this method can tend to diverge from the curve it is integrating, a common problem with numerical integration. In physical systems, it therefore does not necessarily conserve energy and momentum.

For problems in physics, we require an accurate method that will conserve energy over long time scales—longer being defined as millions or even billions of years. The most widely accepted method of integration that accomplishes this goal is known as the Leapfrog technique. This will be discussed in section 4.2 but first we should introduce the idea of reducing a second-order differential equations into a series of first order equations.

### Reducing Second-Order ODEs

The heart of the N-Body problem revolves around integrating the equations of motion given by:

Because this application will be integrating in small time steps, one can assume the acceleration, *a*, is constant over the given .

The methods used to solve ODE’s numerically are the Euler method, Runge-Kutta method, and Leapfrog method among others. The Leapfrog was chosen as the default method because it is both simple and symplectic—it conserves energy and momentum.

Although all of these methods are useful for solving a first order ODE, they cannot solve a second-order ODE. Fortunately, a second-order ODE can be converted into two first-order ODEs and then solved with the before mentioned numerical methods.

Defining the first derivative of *x* to be the velocity, *v*, the above equation can be rewritten as:

This leaves two first-order differential equations suitable for plugging into the Leapfrog integration method.

### Leapfrog Integration

The classic Leapfrog equations are:

A similar form of these equations, called the Verlet method, will be used in the application. To be more specific, the *position* Verlet equations will be used.

## Calculating Force

At the heart of any N-Body code is the gravitational force algorithm. In order to advance the position and velocity of every object one time step, the user must know the total force on each body. In a simulation with N bodies, to find the total gravitational force on a body, we must do N-1 calculations:

With N bodies, the application must therefore perform N\*(N-1) calculations at each time step. This algorithm is a O(N2) algorithm. Dozens of other techniques have been created with less computational difficulty, (see bibliography for several O(Nlog(N)) methods), but the current code described in this document is acceptable because it typically deals with a low number N.

In NBodyUI, The above equation is solved in the three blocks of code below. Step one is to iterate over each body and sum the gravitational force from the remaining N-1 objects. This is done in the API CalculateForces():

public void CalculateForces

(

List<Body> Bodies

)

CalculateForces will loop through the entire Body List and sum the forces on each object. During this process, it will call the API SumForces() to sum the force in the *x*-direction and *y*-direction.

public void SumForces

(

Body b1,

Body b2

)

The SumForces function will solve the gravitational force equation

with the function below, CalculateTotalForce().

public double CalculateTotalForce

(

Body b1,

Body b2

)

For examples of other N-Body force algorithms, see the Appendix.

## Appendix

**Table 1 - Definitions**

|  |  |
| --- | --- |
| Term | Definition |
| Leapfrog | Symplectic integration method widely used in celestial mechanics. |
| Runge-Kutta | Accurate integration method widely seen as the standard for engineering problems. |
| Verlet | Integration method closely related to the Leapfrog technique. |
| Symplectic | Integration method that conserves energy. |
| Simulation time |  |

**Table 2 - NBody Resources**

|  |  |
| --- | --- |
| Description | URL |
| Java N-Body Code | <http://introcs.cs.princeton.edu/34nbody/> |
| Leapfrog integration | <http://en.wikipedia.org/wiki/Leapfrog_integration> |
| Euler Method | <http://en.wikipedia.org/wiki/Euler_method> |
| N-Body simulation | <http://en.wikipedia.org/wiki/N-body_simulation> |
| N-Body simulation tutorial | <http://www.bugcommunity.com/wiki/index.php/Nbody_tutorial> |
| N-Body Particle simulation methods | <http://www.amara.com/papers/nbody.html> |
| Good N-Body problem introduction | <http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-97to98/bryan/page1.html> |
| Runge-Kutta methods | <http://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods> |
| N-Body starter code | <http://www.sns.ias.edu/~piet/act/comp/algorithms/starter/index.html> |
| Sverre Aarseth’s N-Body website | <http://www.ast.cam.ac.uk/~sverre/web/pages/nbody.htm> |