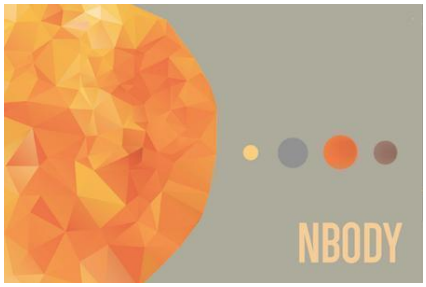
N-Body Simulation

Goals



- To learn about reading input using the StdIn library and printing formatted output using the StdOut library.
- To learn about graphics and animation using the StdDraw library.
- To learn about using the command line to redirect standard input to read from a file.
- To gain more experience using arrays and loops.
- Decompose a large program into small, manageable steps - key to becoming a power programmer!

Before you begin

Getting started. Before you begin coding, do the following:

- *Get familiar with the command line.* Use the `javac-introcs` and `java-introcs` commands to access the standard libraries.
- *Make sure you understand lab exercises.* `Students.java`, `BouncingBallDeluxe.java`, `Distinct.java`
- *Download the data files.* To test your program, you will need `planets.txt` and the accompanying image and sound files. The project folder nbody.zip contains these files, along with a `readme.txt` template and additional test universes.

Background

In this assignment, you will write a program to simulate the motion of *n* particles (bodies) in the plane, mutually affected by gravitational forces, and animate the results.

Here's a video demonstration.

Such methods are widely used in cosmology, semiconductors, and fluid dynamics to study complex physical systems. Scientists also apply the same techniques to other pairwise interactions including Coulombic, Biot–Savart, and van der Waals.

In 1687, Isaac Newton formulated the principles governing the motion of two particles under the influence of their mutual gravitational attraction in his famous *Principia*. However, Newton was unable to solve the problem for three particles. Indeed, in general, solutions to systems of three or more particles must be approximated via numerical simulations.

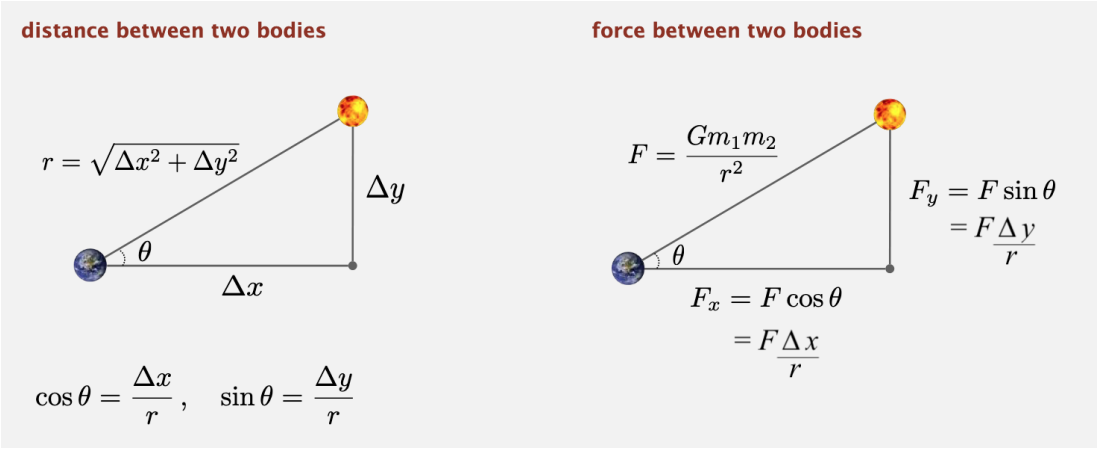
Approach / Design

Simulating the universe: the physics. We review the equations governing the motion of the particles, according to Newton's laws of motion and gravitation. Don't worry if your physics is a bit rusty; all of the necessary formulas are included below. We'll assume for now that the position (p_x, p_y) and velocity (v_x, v_y) of each particle is known. In order to model the dynamics of the system, we must know the net force exerted on each particle.

- **Pairwise force.** *Newton's law of universal gravitation* asserts that the strength of the gravitational force between two particles is given by the product of their masses divided by the square of the distance between them, scaled by the gravitational constant *G*

  $$\mathbf{6.67 \times 10^{-11}\ N \cdot m^2 \cdot kg^{-2}}$$

  The pull of one particle towards another acts on the line between them. Since we are using Cartesian coordinates to represent the position of a particle, it is convenient to break up the force into its *x*- and *y*-components (F_x, F_y) as illustrated below.



- **Net force.** The *principle of superposition* says that the net force acting on a particle in the *x*- or *y*-direction is the sum of the pairwise forces acting on the particle in that direction. Calculating the force (between all pairs of bodies at time t) involves adding all the pairwise forces.

  For example, the forces on earth (at time t) are:

  $$\vec{F}_{earth} = \vec{F}_{mars \to earth} + \vec{F}_{mercury \to earth} + \vec{F}_{sun \to earth} + \vec{F}_{venus \to earth}$$

- **Acceleration.** *Newton's second law of motion* postulates that the accelerations in the *x*- and *y*-directions are given by:

$$a_x = \frac{F_x}{m} \qquad a_y = \frac{F_y}{m}$$

Simulating the universe: the numerics. We use the *leapfrog finite difference approximation scheme* to numerically integrate the above equations: this is the basis for most astrophysical simulations of gravitational systems. In the leapfrog scheme, we discretize time, and update the time variable t in increments of the *time quantum* Δt (measured in seconds). We maintain the position (p_x, p_y) and velocity (v_x, v_y) of each particle at each time step. The steps below illustrate how to evolve the positions and velocities of the particles.

- **Step A (calculate the forces).** For each particle, calculate the net force (F_x, F_y) at the current time t acting on that particle using Newton's law of gravitation and the principle of superposition. Note that force is a vector (i.e., it has direction). In particular, Δx and Δy are signed (positive or negative). In the diagram above, when you compute the force the sun exerts on the earth, the sun is pulling the earth up (Δy positive) and to the right (Δx positive).
- **Step B (update the velocities and positions).** For each particle:
    1. Calculate its acceleration (a_x, a_y) at time t using the net force computed in Step A and Newton's second law of motion: $a_x = F_x / m$, $a_y = F_y / m$.
    2. Calculate its new velocity (v_x, v_y) at the next time step by using the acceleration computed in (i) and the velocity from the old time step: Assuming the acceleration remains constant in this interval, the new velocity is $(v_x + \Delta t\, a_x, v_y + \Delta t\, a_y)$.
    3. Calculate its new position (p_x, p_y) at time $t + \Delta t$ by using the velocity computed in (ii) and its old position at time t: Assuming the velocity remains constant in this interval, the new position is $(p_x + \Delta t\, v_x, p_y + \Delta t\, v_y)$.
- **Step C (draw the universe).** Draw each particle, using the position computed in Step B.

```
for(int second = 0.0;  seconds <10;  second += 0.0001){

    for( all  particles){

        For (all particles that are not equal to current){

            A0 = m1 * mi  * G / ri * ri

        }

        v0 = Old V0 + old A0 * delta T

        P0 = old P0 + oldV0 * delta T

    }

    A0 = m1 * mi  * G / ri * ri

    v0 = Old V0 + old A0 * delta T

    P0 = old P0 + oldV0 * delta T
```

}

Do not interleave steps A and B; otherwise, you will be computing the forces at time *t* using the positions of some of the particles at time *t* and others at time *t* + Δ*t*. The simulation is more accurate when Δ*t* is very small, but this comes at the price of more computation.

Creating an animation. Draw each particle at its current position to standard drawing, and repeat this process at each time step until the designated stopping time. By displaying this sequence of snapshots (or frames) in rapid succession, you will create the illusion of movement.

Implementation Tasks

Overall Requirements - `NBody.java`

- Write a program to simulate the motion of *n* particles in the plane, mutually affected by gravitational forces, and animate the results.

- You MUST implement one class:
    - `NBody.java`

- You MUST follow the assignment specifications for reading input from command-line arguments and standard input, and writing output to standard drawing and standard output.

Program specification.

- Takes **two** double *command-line arguments*: the duration of the simulation (T) and the simulation time increment (Δ*t*).
- Reads in information about the universe from *standard input* using `StdIn`, using several *parallel arrays* to store the data.
- Simulates the universe, starting at time t = 0.0, and continuing in Δ*t* increments as long as *t* < *T*, using the *leapfrog scheme* described above.
- Animates the results using `StdDraw`. That is, for each time increment, draws each planet's image.
- After the simulation completes (reach time T) prints the state of the universe at the end of the simulation (in the same format as the input file) to *standard output* using `StdOut`.

Compiling and executing your program.

To compile your program *from the command line*, type the following in your embedded terminal:

```
> javac-introcs NBody.java
```

To execute your program *from the command line*, redirecting from the file `planets.txt` to standard input, type:

```
> java-introcs NBody 157788000.0 25000.0 < planets.txt
```

After the animation stops, your program must output the final state of the universe in the same format as the input.

Possible Progress Steps.

The key to composing a large program is decomposing it into smaller steps that can be implemented and tested incrementally. Here is the decomposition we recommend for this assignment.

0. Start with comments
1. Parse command-line arguments
2. Read universe from standard input
3. Initialize standard drawing
4. Play music on standard audio
5. Simulate the universe - loop:
    a. Calculate net forces
    b. Update velocities and positions
    c. Draw universe to standard drawing
6. Print universe to standard output

These are the order in which the components will appear in your code. Although your final code will appear in order 1–6, we recommend implementing these steps in order 1, 2, 6, 3, 4, 5, 5B, 5C, 5A. Why? This will facilitate testing and debugging.

Step 0: start with comments

- This outline should help you organize your program.

```
// Step 1. Parse command-line arguments.

// Step 2. Read universe from standard input.

// Step 3. Initialize standard drawing.

// Step 4. Play music on standard audio.

// Step 5. Simulate the universe.

// Step 5A. Calculate net forces.
// Step 5B. Update velocities and positions.
// Step 5C. Draw universe to standard drawing.
```

```
// Step 6. Print universe to standard output.
```

Step 1: parse command-line arguments

- Parse the two command-line arguments *T* and Δ*t* and store their values. Name the first command-line argument `tau` or `stoppingTime` since you should not begin a Java variable name with an uppercase letter.
- **Print** the variables to make sure you parsed them correctly (and in the specified order). For example:
-

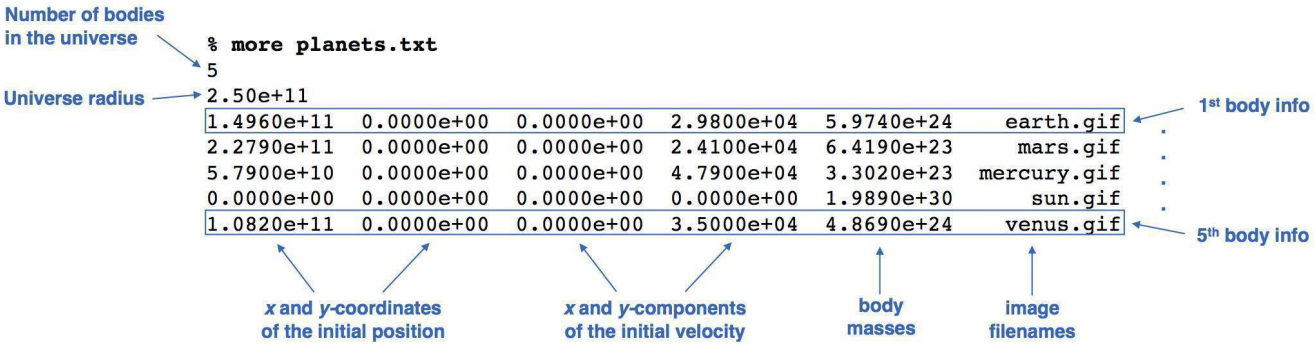
```
> java-introcs NBody 10 1    > java-introcs NBody 157788000.0 25000.0 < planets.txt
T = 10                       T = 1.57788E8
dt = 1.0                     dt = 25000.0
```

- Remove the print statements after you confirm you are parsing the command-line arguments correctly.

Step 2: read universe from standard input

The input format is a text file that contains the information for a particular universe.

- The first value is an integer `n` that represents the number of particles.
- The second value is a real number `radius` that represents the radius of the universe; it is used to determine the scaling of the drawing window.
- Next, there are `n` lines, one per particle.
  - The first two values are the `x`- and `y`-coordinates of the initial position.
  - The next two values are the `x`- and `y`-components of the initial velocity.
  - The fifth value is the mass.
  - The sixth value is a `String` that is the name of an image file used to display the particle.
- The remainder of the file optionally contains a description of the universe, which your program must ignore.
- As an example, planets.txt contains real data from part of our Solar System.



- Read number of bodies n, as an `int`, from standard input.
- Read `radius` of universe standard input, as a `double`.
- Create six (6) parallel arrays, each of length n, to store the six (6) pieces of information characterizing a body. Let `px[i]`, `py[i]`, `vx[i]`, `vy[i]`, and `mass[i]` be `doubles` that store the current position (*x*- and *y*-coordinates), velocity (*x*- and *y*-components), and mass of particle i; let `image[i]` be a `String` that represents the filename of the image used to display particle i.
- Read data associated with each body and store in parallel arrays.

> ***Hint!***
> Recall `Students.java`

- Run your program with `planets.txt`.

```
> java-introcs NBody 157788000.0 25000.0 < planets.txt
[no output]
```

Step 6: print universe to standard output

- Print the universe in the specified format.
- Our input files were created with the following `StdOut.printf()` statements.

```
StdOut.printf("%d\n", n);
StdOut.printf("%.2e\n", radius);
for (int i = 0; i < n; i++) {
      StdOut.printf("%11.4e %11.4e %11.4e %11.4e %11.4e %12s\n",
                  px[i], py[i], vx[i], vy[i], mass[i], image[i]);
}
```

- *Per the course collaboration policy, copying or adapting code that is not yours is permitted only if it comes from the course materials. If you do so, you must cite any code that you copy or adapt (with the exception of code that is included with the assignment). So, you are free to copy or adapt this code fragment, with or without citation.*
- Run your program with `planets.txt`. You should see exactly the output below.

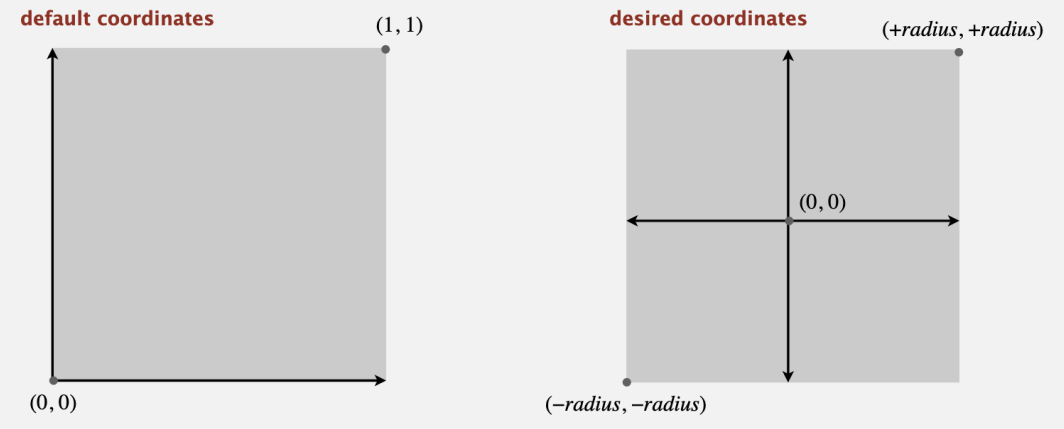
```
> java-introcs NBody 0.0 25000.0 < planets.txt
5
2.50e+11
 1.4960e+11  0.0000e+00  0.0000e+00  2.9800e+04  5.9740e+24    earth.gif
 2.2790e+11  0.0000e+00  0.0000e+00  2.4100e+04  6.4190e+23     mars.gif
 5.7900e+10  0.0000e+00  0.0000e+00  4.7900e+04  3.3020e+23  mercury.gif
 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  1.9890e+30      sun.gif
 1.0820e+11  0.0000e+00  0.0000e+00  3.5000e+04  4.8690e+24    venus.gif
```

- Test your program on another input file.

```
> java-introcs NBody 0.0 1.0 < 3body-zero-gravity.txt
3
5.12e+02
 0.0000e+00  0.0000e+00  1.0000e+00  1.0000e+00  1.0000e-30    earth.gif
 1.2800e+02  0.0000e+00  2.0000e+00  1.0000e+00  1.0000e-40    venus.gif
 0.0000e+00  1.2800e+02  1.0000e+00  2.0000e+00  1.0000e-50     mars.gif
```

- Submit your code via CodePost. If everything is correct so far (reading and printing the universe), **your code should pass Tests 1–7**.
- **Do not continue** to the next step until it does.

Step 3: initialize standard drawing



- The default x- and y-scale supports coordinates between 0 and 1; change scale to be between -radius and +radius.

> ***Hint!***
>
> Recall `StdDraw.setXscale()` and `StdDraw.setYscale()`

- Set the scale of the standard drawing window so that (0, 0) is at the center, (-*radius*, -*radius*) is the lower-left corner, and (+*radius*, +*radius*) is the upper-right corner.
- Call `StdDraw.enableDoubleBuffering()` to enable double buffering and support animation.

Step 4: play music on standard audio

- Use the StdAudio.play method to play the background music.
- If you are running Windows, be sure that the audio stream that Java uses is not muted via *Start -> Programs -> Accessories -> Multimedia -> Volume Control -> Wave Out*.

Step 5: simulate the universe (the *big time loop*)

- Create the *time simulation* loop. Its particle will perform a single time step of the simulation, starting at $t = 0.0$, and continuing in Δt increments as long as $t < T$. This code goes between the code where you read the input and the code where you print the output.
- Test your program by printing the time t at each time step.

```
> java-introcs NBody 23.0 2.5 < planets.txt
t = 0.0
t = 2.5
t = 5.0
t = 7.5
t = 10.0
t = 12.5
t = 15.0
t = 17.5
t = 20.0
t = 22.5
```

```
> java-introcs NBody 25.0 2.5 < planets.txt
t = 0.0
t = 2.5
t = 5.0
t = 7.5
t = 10.0
t = 12.5
t = 15.0
t = 17.5
t = 20.0
t = 22.5
```

- Once you are confident that your time loop is correct, comment out the code that prints the time *t* at each time step.

Steps 5B, 5C, and 5A (below) will go inside this time loop.

Step 5. Simulate the universe. At each time step t :
5A. Calculate the net force on each body.
5B. Update the velocities and positions.
5C. Draw the universe.

Question: In which order should I implement these 3 sub-steps? Answer: 5B, 5C, 5A because calculating forces is hardest.

Question: Can I interleave steps 5A, 5B, and 5C? Answer: No. Not only is it bad design, but it ruins the physics (need position of all bodies at time t, not some at time t + Δt).

Step 5B: update the velocities and positions

- Since we have not implemented Step 5A yet, the forces, F_x and F_y, are 0, so we can initially assume that acceleration, a_x and a_y, is zero and velocity is constant.
- Write a loop to calculate the new velocity and position for each particle.
- Update the velocity of each body: $v_x = v_x + a_x \Delta t$ and $v_y = v_y + a_y \Delta t$ (note a_x and a_y are 0)
- Update the position of each body: $p_x = p_x + v_x \Delta t$ and $p_y = p_y + v_y \Delta t$
- Test on an artificial universe, where it is easy to check the expected results by hand:

```
> java-introcs NBody 1 1 < 3body-zero-gravity.txt
3
```

```
5.12e+02
 1.0000e+00  1.0000e+00  1.0000e+00  1.0000e+00  1.0000e-30    earth.gif
 1.3000e+02  1.0000e+00  2.0000e+00  1.0000e+00  1.0000e-40    venus.gif
 1.0000e+00  1.3000e+02  1.0000e+00  2.0000e+00  1.0000e-50     mars.gif
```

```
> java-introcs NBody 192 1 < 3body-zero-gravity.txt
3
5.12e+02
 1.9200e+02  1.9200e+02  1.0000e+00  1.0000e+00  1.0000e-30 earth.gif
 5.1200e+02  1.9200e+02  2.0000e+00  1.0000e+00  1.0000e-40 venus.gif
 1.9200e+02  5.1200e+02  1.0000e+00  2.0000e+00  1.0000e-50        mars.gif
```

- Submit your code via CodePost. If everything is correct so far (ignoring the gravitational forces and drawing), **your code should pass Tests 1–10**. Do not continue to the next step until it does.

Step 5C: draw universe to standard drawing

- Draw each particle at its current position to standard drawing, and repeat this process at each time step until the designated stopping time. By displaying this sequence of snapshots (or frames) in rapid succession, you will create the illusion of movement. After each time step, draw the background image `starfield.jpg`; redraw all the particles in their new positions; and control the animation speed (about 40 frames per second looks good).
    - Use `StdDraw.picture(x, y, filename)` to draw the background image `starfield.jpg`, centered at `(0, 0)`.
    - Then, write a loop to display the pnarticles.
    - Call `StdDraw.show()` to display results on screen
    - Call `StdDraw.pause(20)` to control animation speed

- Run the program with `planets.txt`. You should now see the four planets moving off the screen in a straight line, with constant velocity.

```
> java-introcs NBody 157788000.0 25000.0 < planets.txt
```

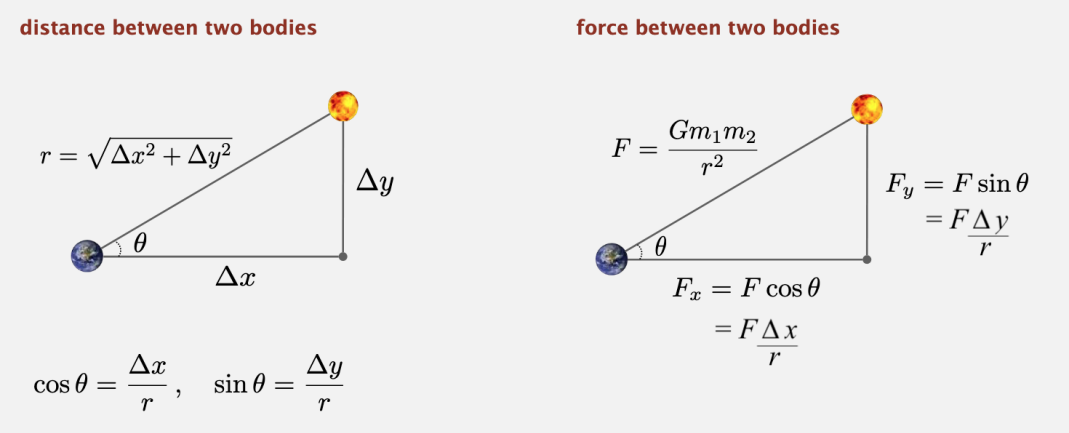
- If the planets are flickering, be sure you are using *double buffering*, following the template in BouncingBallsDeluxe.java. In particular, call `StdDraw.enableDoubleBuffering()` once at the beginning of the program; and call `StdDraw.show()` and `StdDraw.pause(20)` at the end of each time step (frame), not after each call to `StdDraw.picture()`.

- Run your program on another input file, such as `kaleidoscope.txt` - produces this animation.

```
> java-introcs NBody 157788000.0 25000.0 < kaleidoscope.txt
```

Step 5A: calculate the forces

- This step is the most difficult one to implement. Let's review:
  - **Pairwise force.** Recall this diagram from above:



P old position

V old speed

M[n] =

```
for(int j = 0; j < n; j++){
        if(i != j){


        }
}
```

- **Net force.** The *principle of superposition* says that the net force acting on a particle in the *x*- or *y*-direction is the sum of the pairwise forces acting on the particle in that direction. Calculating the force (between all pairs of bodies at time t) involves adding all the pairwise forces. For example, the forces on earth (at time t) are:

$$\vec{F}_{earth} = \vec{F}_{mars \to earth} + \vec{F}_{mercury \to earth} + \vec{F}_{sun \to earth} + \vec{F}_{venus \to earth}$$

- Implementing Step 5A:
  - To calculate the net force acting on each body, you will need two additional arrays `fx[i]` and `fy[i]` to store the net force acting on body `i`.
  - The net force (arrays `fx[i]` and `fy[i]`) must be initialized to 0 at each time step.
  - Write two nested for loops to calculate the net force exerted by body `j` on body `i`. Add these values to `fx[i]` and `fy[i]`, but skip the case when `i` equals `j`.

> ***Hint!***
> Before calculating the net forces, can you write two nested loops that enumerates all pairs of bodies? For example - what pairs are not printed?

`n = 5`
`0-1 0-2 0-3 0-4`
`1-0 1-2 1-3 1-4`
`2-0 2-1 2-3 2-4`
`3-0 3-1 3-2 3-4`
`4-0 4-1 4-2 4-3`
	`n = 4`
	`0-1 0-2 0-3`
	`1-0 1-2 1-3`
	`2-0 2-1 2-3`
	`3-0 3-1 3-2`

  - Once you have these values computed, **go back to Step 5B** and use them to compute the acceleration (instead of assuming it is zero).
    - Calculate its acceleration (a_x, a_y) at time *t* using the net force computed in Step 5A and Newton's second law of motion: $a_x = \frac{F_x}{m}$ $a_y = \frac{F_y}{m}$
  - Test your program on *several* data files.
  - Submit your code via CodePost. Your code should now pass all of the tests.

Debugging

Checkstyle complains about the variable I named T because it begins with an uppercase letter. Which name should I use instead? By convention, a variable in Java should begin with a lowercase letter and use camel case, such as `isLeapYear`. A constant variable (a variable whose value does

not change during the execution of a program, or from one execution of the program to the next) should begin with an uppercase letter and use underscores to separate any word boundaries, such as `GRAVITATIONAL_CONSTANT`. So, a Java variable name might not always perfectly align with the corresponding domain-specific mathematical notation. In this case, `tao` or `stoppingTime` would be fine choices.

Why do I get an `InputMismatchException` or a `NoSuchElementException` when I read data using `StdIn`? `InputMismatchException` means that your program is attempting to read data of one type but the next piece of data on standard input is of an incompatible type. For example, if you use `StdIn.readInt()` and the next token on standard input is `3.14`. `NoSuchElementException` means that your program is trying to read data from standard input when there is no more data available from standard input. For example, if you use `StdIn.readInt()` to read two integers but standard input contains only a single integer. (This can also happen if you wipe out an input file, such as `planets.txt`, by errantly typing > instead of a < during a redirection command.)

My animation looks fine, but the numbers are off a little bit when I follow the tests below, what could be causing this? Check that you followed the assignment instructions exactly. In particular, you should have separate loops for Steps A, B, and C and you should update the velocities before the positions in Step B.

I draw the bodies, but they do not appear on the screen. Why? Did you use StdDraw.setXscale() and StdDraw.setYscale() to change the x- and y-coordinate systems to use the physics coordinates instead of the unit box? Since you want it centered on the origin with a *square radius* of radius, the minimum of each axis should be –radius and the maximum should be +radius.

My bodies repel each other. (Animation.) Why don't they attract each other? Make sure that you get the sign right when you apply Newton's law of gravitation: `(x[j] - x[i]) vs. (x[i] - x[j])`. Note that `Δx and Δy` can be positive or negative so do not use `Math.abs()`. Do not consider changing the universal gravitational constant `G` to patch your code!

Why do my bodies disappear? (Animation.) Make sure you are using the correct index variables. It's very easy to confuse an i with a j, especially when cutting and pasting Java statements.

Testing Output

Here are our results for a few sample inputs.

```
> java-introcs NBody 0.0 25000.0 < planets.txt           // zero steps
5
2.50e+11
```

```
 1.4960e+11  0.0000e+00  0.0000e+00  2.9800e+04  5.9740e+24    earth.gif
 2.2790e+11  0.0000e+00  0.0000e+00  2.4100e+04  6.4190e+23     mars.gif
 5.7900e+10  0.0000e+00  0.0000e+00  4.7900e+04  3.3020e+23  mercury.gif
 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  1.9890e+30      sun.gif
 1.0820e+11  0.0000e+00  0.0000e+00  3.5000e+04  4.8690e+24    venus.gif
```

```
> java-introcs NBody 25000.0 25000.0 < planets.txt       // one step
5
2.50e+11
 1.4960e+11  7.4500e+08 -1.4820e+02  2.9800e+04  5.9740e+24    earth.gif
 2.2790e+11  6.0250e+08 -6.3860e+01  2.4100e+04  6.4190e+23     mars.gif
 5.7875e+10  1.1975e+09 -9.8933e+02  4.7900e+04  3.3020e+23  mercury.gif
 3.3087e+01  0.0000e+00  1.3235e-03  0.0000e+00  1.9890e+30      sun.gif
 1.0819e+11  8.7500e+08 -2.8329e+02  3.5000e+04  4.8690e+24    venus.gif
```

```
> java-introcs NBody 50000.0 25000.0 < planets.txt       // two steps
5
2.50e+11
 1.4959e+11  1.4900e+09 -2.9640e+02  2.9799e+04  5.9740e+24    earth.gif
 2.2790e+11  1.2050e+09 -1.2772e+02  2.4100e+04  6.4190e+23     mars.gif
 5.7826e+10  2.3945e+09 -1.9789e+03  4.7880e+04  3.3020e+23  mercury.gif
 9.9262e+01  2.8198e-01  2.6470e-03  1.1279e-05  1.9890e+30      sun.gif
 1.0818e+11  1.7499e+09 -5.6660e+02  3.4998e+04  4.8690e+24    venus.gif
```

```
> java-introcs NBody 60000.0 25000.0 < planets.txt       // three steps
5
2.50e+11
 1.4958e+11  2.2349e+09 -4.4460e+02  2.9798e+04  5.9740e+24    earth.gif
 2.2789e+11  1.8075e+09 -1.9158e+02  2.4099e+04  6.4190e+23     mars.gif
 5.7752e+10  3.5905e+09 -2.9682e+03  4.7839e+04  3.3020e+23  mercury.gif
 1.9852e+02  1.1280e+00  3.9705e-03  3.3841e-05  1.9890e+30      sun.gif
 1.0816e+11  2.6248e+09 -8.4989e+02  3.4993e+04  4.8690e+24    venus.gif
```

```
> java-introcs NBody 31557600.0 25000.0 < planets.txt    // one year
5
```

```
2.50e+11
 1.4959e+11 -1.6531e+09  3.2949e+02  2.9798e+04  5.9740e+24    earth.gif
-2.2153e+11 -4.9263e+10  5.1805e+03 -2.3640e+04  6.4190e+23     mars.gif
 3.4771e+10  4.5752e+10 -3.8269e+04  2.9415e+04  3.3020e+23  mercury.gif
 5.9426e+05  6.2357e+06 -5.8569e-02  1.6285e-01  1.9890e+30      sun.gif
-7.3731e+10 -7.9391e+10  2.5433e+04 -2.3973e+04  4.8690e+24    venus.gif
```

```
// this test should take only a few seconds. 4.294E9 is bigger than the largest int
> java-introcs NBody 4.294E9 2.147E9 < 3body.txt
3
1.25e+11
 2.1470e+12 -7.8082e-03  5.0000e+02 -3.6368e-12  5.9740e+24    earth.gif
 1.2882e+14 -1.5100e+17  3.0000e+04 -3.5165e+07  1.9890e+30      sun.gif
-1.2882e+14  1.5100e+17 -3.0000e+04  3.5165e+07  1.9890e+30      sun.gif
```

Submission

Submit `NBody.java` and a completed `readme.txt` file.

Enrichment

What is the music in 2001.wav? It's the fanfare to Also sprach Zarathustra by Richard Strauss. It was popularized as the key musical motif in Stanley Kubrick's 1968 film 2001: A Space Odyssey.

I'm a physicist. Why should I use the leapfrog method instead of the formula I derived in high school? In other words, why does the position update formula use the velocity at the *updated* time step rather than the previous one? Why not use the $\frac{1}{2}\, a\, t^2$ formula? The leapfrog method is more stable for integrating Hamiltonian systems than conventional numerical methods like Euler's method or Runge–Kutta. The leapfrog method is *symplectic*, which means it preserves properties specific to Hamiltonian systems (conservation of linear and angular momentum, time-reversibility, and conservation of energy of the discrete Hamiltonian). In contrast, ordinary numerical methods become dissipative and exhibit qualitatively different long-term behavior. For example, the earth would slowly spiral into (or away from) the sun. For these reasons, symplectic methods are extremely popular for *n*-body calculations in practice. You asked!

Here's a more complete explanation of how you should interpret the variables. The classic *Euler method* updates the position using the velocity at time t instead of using the updated velocity at time $t + \Delta t$. A better idea is to use the velocity at the midpoint $t + \Delta t / 2$. The leapfrog method does this in a clever way. It maintains the position and velocity one-half time step out of phase: At the beginning of an iteration, (p_x, p_y) represents the position at time t and (v_x, v_y) represents the velocity at time $t - \Delta t / 2$. Interpreting the position and velocity in this way, the updated position $(p_x + \Delta t\, v_x, p_y + \Delta t\, v_y)$ uses the velocity at time $t + \Delta t / 2$. Almost magically, the only special care needed to deal with the half time-steps is to initialize the system's velocity at time $t = -\Delta t / 2$ (instead of $t = 0.0$), and you can assume that we have already done this for you. Note also that the acceleration is computed at time t so that when we update the velocity, we are using the acceleration at the midpoint of the interval under consideration.

Are there analytic solutions known for n-body systems? Yes, Sundman and Wang developed global analytic solutions using convergent power series. However, the series converge so slowly that they are not useful in practice. See The Solution of the N-body Problem for a brief history of the problem.

Who uses n-body simulation? N-body simulations play a crucial role in our understanding of the universe. Astrophysicists use it to study stellar dynamics at the galactic center, stellar dynamics in a globular cluster, colliding galaxies, and the formation of the structure of the Universe. The strongest evidence we have for the belief that there is a black hole in the center of the Milky Way comes from very accurate *n*-body simulations. Many of the problems that astrophysicists want to solve have millions or billions of particles. More sophisticated computational techniques are needed.

The same methods are also widely used in molecular dynamics, except that the heavenly bodies are replaced by atoms, gravity is replaced by some other force, and the leapfrog method is called *Verlet's method*. With van der Waals forces, the interaction energy may decay as $1/R^6$ instead of an inverse square law. Occasionally, 3-way interactions must be taken into account, e.g., to account for the covalent bonds in diamond crystals. Kepler's Orrery uses an *n*-body simulator to compose and play music.

What techniques are used to do n-body simulation in practice? Here's a wealth of information in n-body simulation.

Any ideas for creating my own universe? Here are some interesting two-body systems. Here is beautiful 21-body system in a figure-8 —reproducing this system (in a data file in our format) will earn you a small amount of extra credit.

> There are limitless opportunities for additional excitement and discovery here. Create an alternate universe (using the same input format). Try adding other features, such as supporting elastic or inelastic collisions. Or, make the simulation three-dimensional by doing calculations for x-, y-, and z-coordinates, then using the z-coordinate to vary the sizes of the planets. Add a rocket ship that launches from one planet and has to land on another. Allow the rocket ship to exert force with the consumption of fuel.