# NBody Simulation

## Writing the Body Class

In our program, we'll have instances of the **Body** class do the job of calculating all the numbers we learned about in the previous example. We'll write helper methods, one by one, until our Body class is complete.

### calcDistance

Start by adding a method called **calcDistance** that calculates the distance between two **Body**s. This method will take in a single Body and should return a double equal to the distance between the supplied body and the body that is doing the calculation, e.g.

```
samh.calcDistance(rocinante);
```

It is up to you this time to figure out the signature of the method. Once you have completed this method, go ahead and recompile and run the next unit test to see if your code is correct.

Compile and run with:

```
javac Body.java TestCalcDistance.java
java TestCalcDistance
```

Note that in Java, there is no built-in operator that does squaring or exponentiation.

## calcForceExertedBy

The next method that you will implement is **calcForceExertedBy**. The **calcForceExertedBy** method takes in a **Body**, and returns a double describing the force exerted on this body by the given body. You should be calling the **calcDistance** method inside this method. As an example, **samh.calcForceExertedBy(rocinante)** for the numbers in "Double Check Your Understanding" return **1.334 · 10^–9.**

Once you've finished **calcForceExertedBy**, re-compile and run the next unit test.

```
javac Body.java TestCalcForceExertedBy.java
java TestCalcForceExertedBy
```

Hint: It is good practice to declare any constants as a 'static final' variable in your class, and to use that variable anytime you wish to use the constant.

Hint 2: Java supports scientific notation. For example, I can write **double someNumber = 1.03e-7;**.

## calcForceExertedByX and calcForceExertedByY

The next two methods that you should write are **calcForceExertedByX** and **calcForceExertedByY**. Unlike the **calcForceExertedBy** method, which returns the total force, these two methods describe the force exerted in the X and Y directions, respectively. *Remember to check your signs!* Once you've finished, you can recompile and run the next unit test. As an example, **samh.calcForceExertedByX(rocinante)** in "Double Check Your Understanding" should return 1.0672·10^–9.

NOTE: Do not use Math.abs to fix sign issues with these methods. This will cause issues later when drawing planets.

```
javac Body.java TestCalcForceExertedByXY.java
java TestCalcForceExertedByXY
```

## calcNetForceExertedByX and calcNetForceExertedByY

Write methods **calcNetForceExertedByX** and **calcNetForceExertedByY** that each take in an array of **Body**s and calculates the net X and net Y force exerted by all bodies in that array upon the current Body. For example, consider the code snippet below:

```
Body[] allBodys = {samh, rocinante, aegir};
samh.calcNetForceExertedByX(allBodys);
samh.calcNetForceExertedByY(allBodys);
```

The two calls here would return the values given in “Double Check Your Understanding.”

As you implement these methods, remember that **Body**s cannot exert gravitational forces on themselves! Can you think of why that is the case (hint: the universe will possibly collapse in on itself, destroying everything including you)? To avoid this problem, ignore any body in the array that is equal to the current body. To compare two bodies, use the **.equals** method instead of **==**: **samh.equals(samh)** (which would return **true**). In week 2, we will explain the difference.

When you are done go ahead and run:

```
javac Body.java TestCalcNetForceExertedByXY.java
java TestCalcNetForceExertedByXY
```

If you're tired of the verbosity of for loops, you might consider reading about less verbose looping constructs (for and the 'enhanced for') given on page 114-116 of HFJ, or online at this link. HW0 also goes into detail about enhanced for loops. This is not necessary to complete the project.

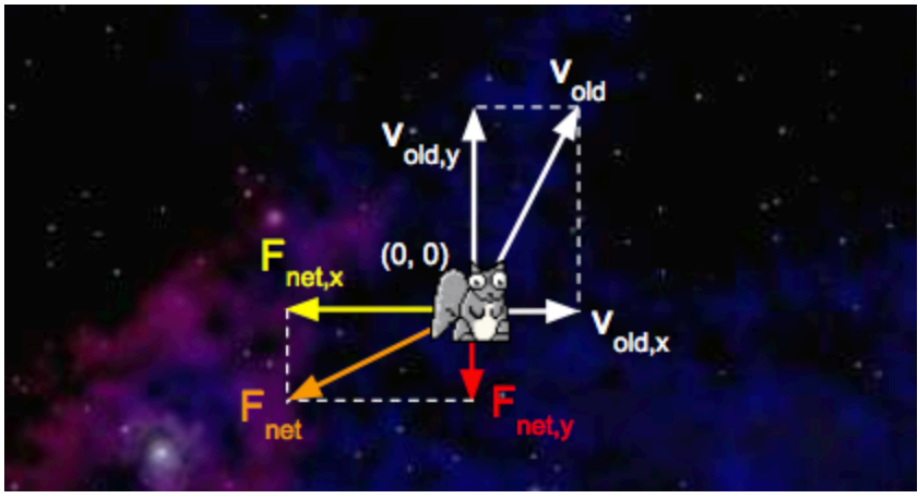
## update

Next, you'll add a method that determines how much the forces exerted on the body will cause that body to accelerate, and the resulting change in the body's velocity and position in a small period of time $dt$. For example, `samh.update(0.005, 10, 3)` would adjust the velocity and position if an $x$-force of $10$ Newtons and a $y$-force of $3$ Newtons were applied for $0.005$ seconds.

You must compute the movement of the Body using the following steps:

1. Calculate the acceleration using the provided $x$- and $y$-forces.
2. Calculate the new velocity by using the acceleration and current velocity. Recall that acceleration describes the change in velocity per unit time, so the new velocity is $(v_x + dt \cdot a_x, v_y + dt \cdot a_y)$.
3. Calculate the new position by using the velocity computed in step 2 and the current position. The new position is $(p_x + dt \cdot v_x, p_y + dt \cdot v_y)$.

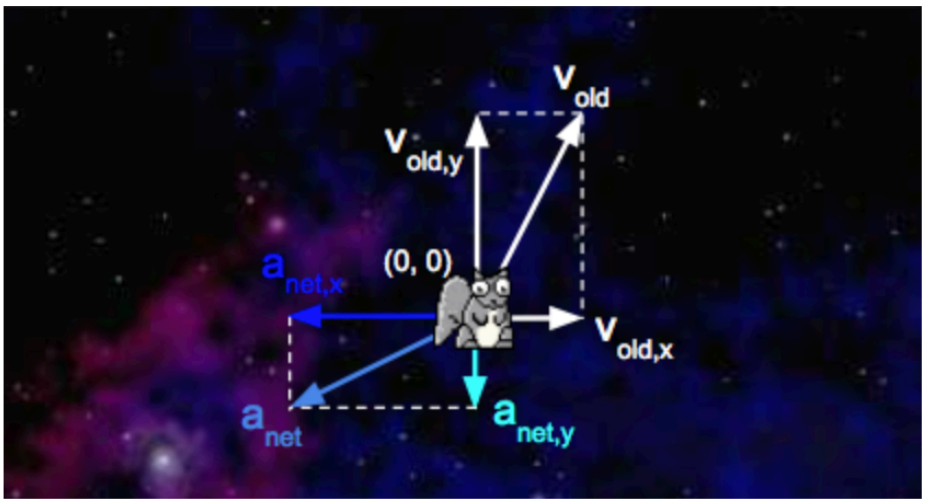
Let's try an example! Consider a squirrel initially at position $(0, 0)$ with a $v_x$ of $3\frac{\text{m}}{\text{s}}$ and a $v_y$ of $5\frac{\text{m}}{\text{s}}$. $F_{\text{net},x}$ is $-5$ N and $F_{\text{net},y}$ is $-2$ N. Here's a diagram of this system:



We'd like to update with a time step of $1\ \text{second}$. First, we'll calculate the squirrel's net acceleration:

- $a_{\text{net},x} = \dfrac{F_{\text{net},x}}{m} = \dfrac{-5\ \text{N}}{1\ \text{kg}} = -5\dfrac{\text{m}}{\text{s}^2}$
- $a_{\text{net},y} = \dfrac{F_{\text{net},y}}{m} = \dfrac{-2\ \text{N}}{1\ \text{kg}} = -2\dfrac{\text{m}}{\text{s}^2}$

With the addition of the acceleration vectors we just calculated, our system now looks like this:



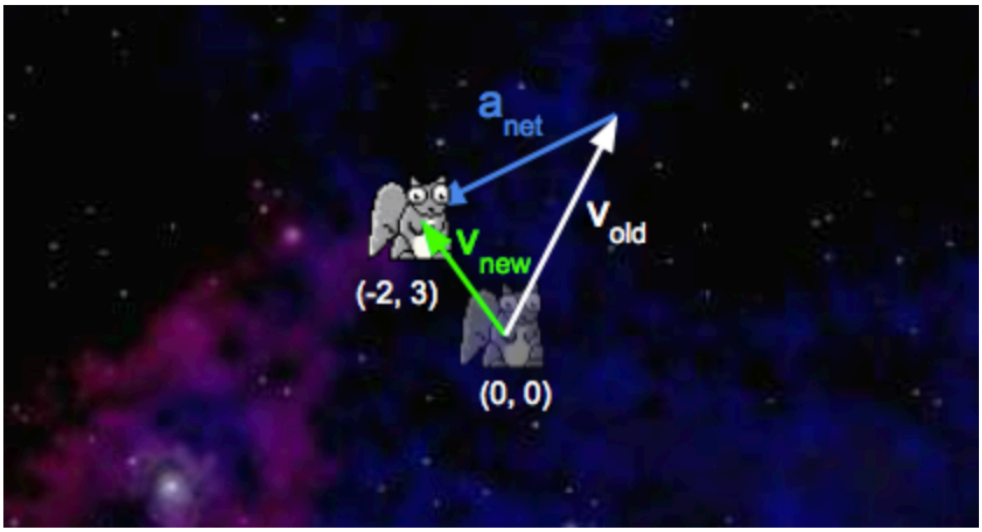

Second, we'll calculate the squirrel's new velocity:

- $v_{\text{new},x} = v_{\text{old},x} + dt \cdot a_{\text{net},x} = 3\dfrac{\text{m}}{\text{s}} + 1\ \text{s} \cdot -5\dfrac{\text{m}}{\text{s}^2} = -2\dfrac{\text{m}}{\text{s}}$
- $v_{\text{new},y} = v_{\text{old},y} + dt \cdot a_{\text{net},y} = 5\dfrac{\text{m}}{\text{s}} + 1\ \text{s} \cdot -2\dfrac{\text{m}}{\text{s}^2} = -3\dfrac{\text{m}}{\text{s}}$

Third, we'll calculate the new position of the squirrel:

- $p_{\text{new},x} = p_{\text{old},x} + dt \cdot v_{\text{new},x} = 0\ \text{m} + 1\ \text{s} \cdot -2\dfrac{\text{m}}{\text{s}} = -2\ \text{m}$
- $p_{\text{new},y} = p_{\text{old},y} + dt \cdot v_{\text{new},y} = 0\ \text{m} + 1\ \text{s} \cdot 3\dfrac{\text{m}}{\text{s}} = 3\ \text{m}$

Here's a diagram of the updated system:




For math/physics experts: You may be tempted to write a more accurate simulation where the force gradually increases over the specified time window. Don't! Your simulation must follow exactly the rules above.

Write a method `update(dt, fX, fY)` that uses the steps above to update the body's position and velocity instance variables (this method does not need to return anything).

Once you're done, recompile and test your method with:

```
javac Body.java TestUpdate.java
java TestUpdate
```

Once you've done this, you've finished implementing the physics. Hoorah! You're halfway there.