Introduction To Rigid Body Physics

Module 2: Forces in One Dimension and Numerical Integration

University of Reddit
Instructor: Brian Dolhansky
Course website: http://universityofreddit.com/v2/class.php?id=111
Course email: intro.to.rbp@gmail.com
Instructor email: bdolmail@gmail.com

Contents

1 Introduction

In this module, we learn about forces and their effect on simple zero-dimensional bodies. In addition, we will learn how to numerically integrate acceleration and velocity. Numerical integration is necessary because finding closed form solutions to integration problems is relatively hard to program. Finally, we will implement the theory we learned in this and the previous lesson to create a fireworks simulator.

Note: Reddit users `dprimedx` and `r4and0muser9482` pointed out some bugs and issues in the code. Specifically, the scalar multiplication and vector addition functions in the Vector class were incorrect. In addition, when running the simulation from IDLE, the program would hang on quit. Fixes to these issues have been added to the latest version of the code, which can be found in the root `src/` directory.

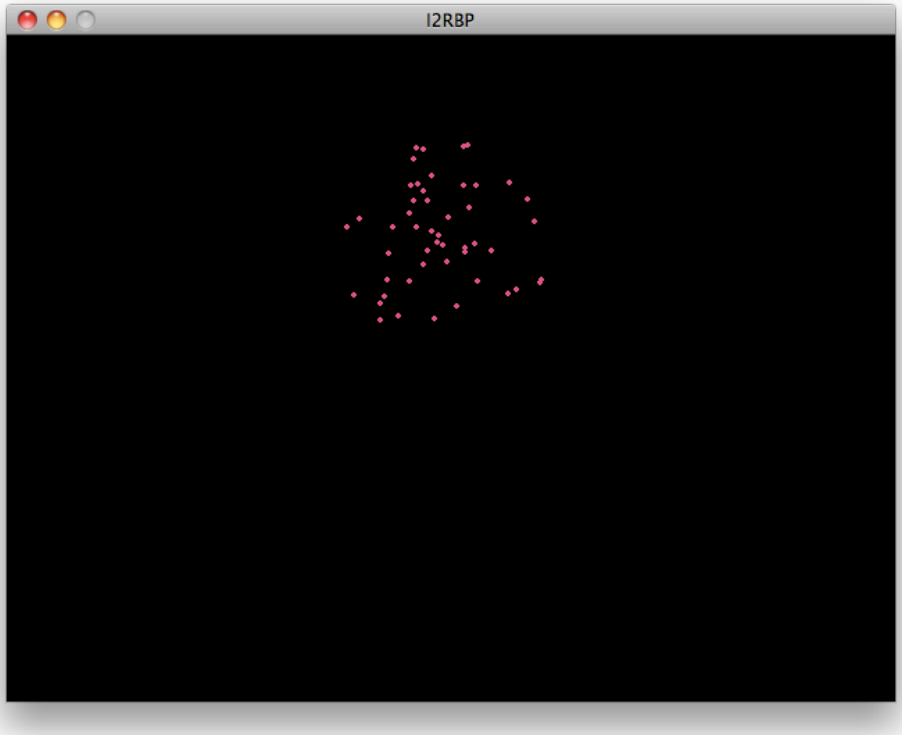


Figure 1: The final product after two lessons.

2 Suggested reading

The first website provides some background information on differential equations from a physical modelling perspective. The second link covers much of what we'll be going over in this lesson, namely linear forces and integrators. The third link provides some in-depth coverage of Euler's method.

- http://www.cs.cmu.edu/afs/cs/user/baraff/www/pbm/diffyq.pdf
- http://chrishecker.com/images/d/df/Gdmphys1.pdf
- http://tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx

3 Forces

3.1 Overview

If you wanted to move an object from one place to another, how would you do it? If it were light enough, you might pick it up in your hand and simply carry it to its destination. If it were large or you needed to transport it to someplace far away, maybe you would use a forklift to put it in the cargo hold of a plane. In each case, you are expending energy to change an object's position. We call this influence on motion force.

Any time the position or velocity of an object changes, it was due to a force. You may remember Isaac Newton's more elegant definition, his First Law of Motion, something we all learned in our first physics class:

Newton's First Law of Motion[1]
Every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it.

Despite what Newton's First Law seems to imply, we cannot directly change an object's position by applying a force.[2] We can't even change an object's velocity! How, then, does an object begin moving, or a moving object stop? Well, it turns out that force and acceleration are directly related. Again we turn to Newton.

Newton's Second Law of Motion[1]
The relationship between an object's mass m, its acceleration $\mathbf{a}$, and the applied force $\mathbf{F}$ is $\mathbf{F} = m\mathbf{a}$.

Remember from the previous lesson, vectors are denoted with a bold font. In two dimensions, force and acceleration are vectors. But what exactly is m?

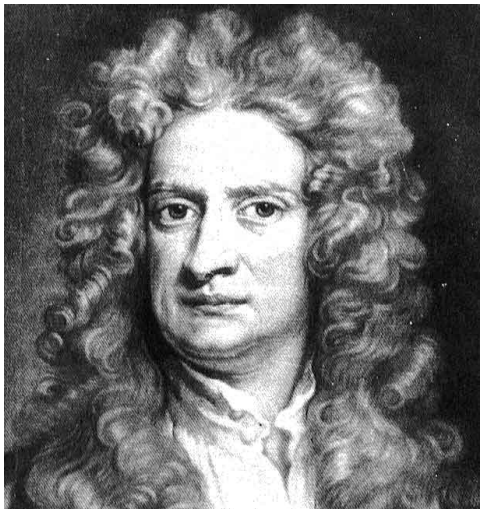


Figure 2: Thanks, Newton. Thewton!

It's here we introduce another property of an object, mass. Just as any dynamic physical object has the intrinsic properties of position, velocity and acceleration, all physical objects have mass. In the real world, mass is analogous to the amount of "stuff" in an object, or the amount of matter it contains. Objects with large mass exhibit a strong gravitational field, and, according to special relativity, mass is proportional to a body's energy. Most importantly (in our case), mass defines how "hard" it is to move an object. This property of mass is known as *inertial* mass.

If you remember from Newton's second law, mass is related to both force and acceleration through the Equation 1:

$$\mathbf{F} = m\mathbf{a} \tag{1}$$

Rearranging that equation gives:

$$m = \frac{\mathbf{F}}{\mathbf{a}}$$

Therefore, we can think of mass as *resistance to force*. A more massive object requires a larger force to move the same distance as a less massive object, given the same force.

So what is the importance of force and mass? Where do they fit into our overall simulation? Our dataflow model at the end of last lesson is shown in Figure 3. We didn't go over what comes before acceleration.

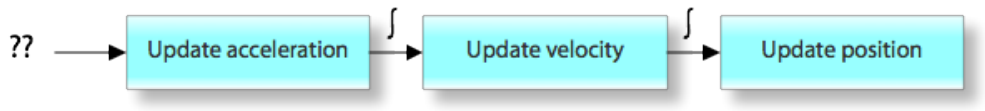


Figure 3: What goes before acceleration?! TELL ME NOW.

I hinted at the first step of our simulation last lesson. If you have been paying attention, you know that force is directly related to acceleration. Therefore, force *must* come before acceleration. Using this realization, here's how our simulator now works:

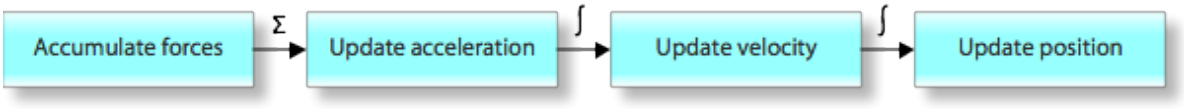


Figure 4: Force is the missing piece to our puzzle.

Notice that we don't integrate our force vectors to update acceleration. Instead, we sum them. We'll cover this in more detail in the section on integrators. For now, let's delve deeper into forces.

3.2 Types of forces

In the physical world, there are many things that can influence an object's motion. For instance, when a baseball player hits a home run, the bat he swung exerted a large amount of force on a ball. In addition, the same amount of force was exerted on the bat (Newton's Third Law). Alternatively, the Earth revolves around the Sun due to a constant gravitational force. We can group all the forces we'll encounter into two broad groups, unary and n-ary forces.[3]

3.2.1 Unary forces

Unary forces act on all particles, but may vary according to particle position, velocity or acceleration. Using our example above, the Sun's gravitational field is a unary force (if we

specify that the sun is not an object in our simulation), as it acts on all planets. Even though every planet experiences the Sun's gravity, not all planets experience the same magnitude of gravitational force. The Sun's gravitational magnitude decreases as distance increases. Therefore, we can say that gravity is a unary force, dependent on position.

Likewise, drag due to air resistance acts on all objects, but changes based on a object's velocity. A fast-moving object experiences a lot of air resistance, but one at rest experiences none. Air resistance is a unary force, dependent on velocity.

The common thread between all unary forces is that they don't depend on the state of other objects. It doesn't matter if a hundred sky divers are parachuting somewhere else on the Earth. You will experience the same amount of air resistance you normally would if you were to jump out of a plane (ignoring turbulence caused by the army of other sky divers).

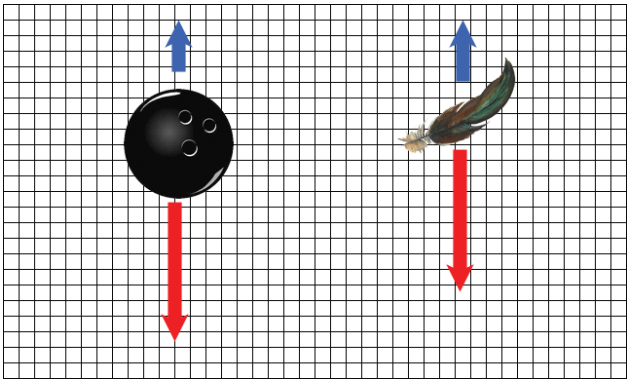


Figure 5: The bowling ball and feather both experience the same amount of gravitational pull, but different amounts of air resistance. However, the bowling ball and the feather don't affect each other.

3.2.2 N-ary forces

The other type of force we will be dealing with are n-ary forces. These are forces that act on a subset of particles, and often rely on the state of other particles to compute. For instance, objects that collide exert a force on each other. This reaction force depends on the particle's velocity and mass, among other things. In addition, forces attached with some kind of constraint, such as a spring, experience a directional force. Particles not attached to the spring do not experience this force.

These forces are often a little trickier to calculate. More accurately, they are more computationally complex. We can't just run through the list of particles and apply the same force to each. We need to first detect which particles should undergo our specific n-ary force, and then update those particles accordingly. However, the updated particles might affect *other* particles. This is a problem that we need to overcome with our simulator. We won't cover n-ary forces in detail in this lesson. However, n-ary forces play a key role in collision response, which will be covered in the future.

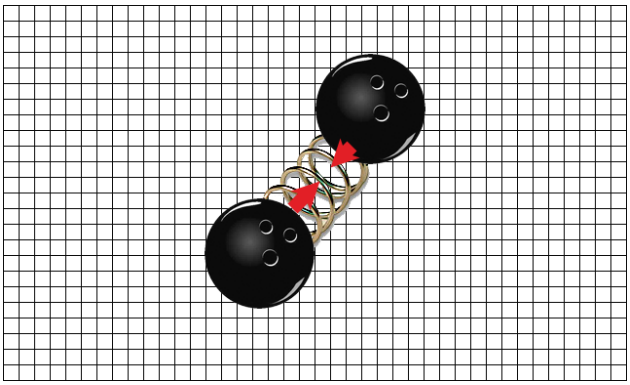


Figure 6: The spring exerts a force on each bowling ball based on its extension. But the spring's length depends on the position of each bowling ball, which is in turn affected by the force exerted by the spring.

3.3 Exploring force mathematically

While we have a nice qualitative definition of force (an influence on a body that causes acceleration), we need to come up with a more rigorous idea of what force is so that we can translate it into code. I have confidence in our ability to take derivatives and integrals of functions involving vectors, so we will skip one-dimensional forces and dive right into two dimensions.

3.3.1 Force as a vector

In two dimensions, force has both a magnitude (strength) and a direction. It makes sense, therefore, to define force using vectors. Recall Equation 1:

$$\mathbf{F} = m\mathbf{a}$$

This looks a lot like our definition of vector scalar multiplication! In fact, if we rearrange

that equation,

$$\mathbf{a} = \frac{1}{m}\mathbf{F}$$

we see that acceleration is simply a scaled version of the force vector. Force is inversely scaled according to mass, which matches our earlier definition of inertial mass, *resistance to change in motion*! If we know the forces acting on an object, we can then determine the acceleration, and, in turn, the velocity and position. Acceleration is simply the sum of forces on an object, divided by that object's mass.

3.3.2 Accumulating forces

From the particle's point of view, it doesn't matter if the force acting on it is unary or n-ary. A push is a push. Therefore, we can accumulate all forces (once known) on an object in a similar manner without regard to the source. How do forces on an object sum? It turns out that since we are working with particles, we can simply use vector addition and sum all the forces acting on object to arrive at the overall force acting on an object.

$$\mathbf{F} = \sum_i \mathbf{f_i} \tag{2}$$

According to Equation 2, we add all individual forces, $\mathbf{f_i}$, acting on an object to arrive at the overall force, $\mathbf{F}$. Why can we do this? Because we are using zero-dimensional particles, there is only one location a force can act on an object, that object's center of mass. Center of mass is a concept we will explore further when we move to full two dimensional rigid bodies. The quick answer is that forces that act through the center of mass directly affect acceleration. For now, we will generalize and say that all forces act through the center of mass.

If we combine Equations 1 and 2, we arrive at Equation 3:

$$\mathbf{a} = \frac{1}{m}\sum_i \mathbf{f_i} \tag{3}$$

We can figure out the acceleration vector for each time step by summing all forces acting on an object, and dividing the summed forces by the object's mass. We can then determine the object's velocity and position by integrating acceleration. We're one step closer to having a working particle simulator. However, we skipped the fact about how we calculate force in the first place.

3.3.3 Figuring out force

The forces we are dealing with can be grouped into another two broad categories, easy-to-calculate forces and hard-to-calculate forces. Let's look at two unary forces, gravity and drag.

Gravity is an easy-to-calculate force. At trivial distances (where the force of gravity is not severely diminished, such as on the Earth's surface), gravity can be defined as:

$$\mathbf{F_g} = m\mathbf{g} \tag{4}$$

If you remember Galileo's apocryphal kinematics experiment, whereupon he dropped two weights from the top of the Leaning Tower of Pisa, you may remember one of the defining characteristics of gravity - it is not affected by the mass of the objects under its influence. This is proved by dividing Equation 4 by m to determine an object's acceleration due to gravity. We simply arrive at $\mathbf{a} = \mathbf{g}$, which is certainly an easy calculation! To update particles under the influence of gravity in a simulator, we would simply determine the gravity vector, and then set each object's acceleration to the global gravity. We could then integrate to update position.

But you may remember in the section on unary forces, there exist forces that depend on the velocity of a particle. For instance, take the force exerted due to viscous drag at low speeds:

$$\mathbf{F_d} = -b\mathbf{v} \tag{5}$$

This is an issue, and this is what makes drag a hard-to-calculate force. If we substitute $m\mathbf{a}$ for $\mathbf{F_d}$, and we remember that acceleration is the derivative of velocity, we get

$$\dot{\mathbf{v}}(t) = -\frac{b}{m}\mathbf{v}(t)$$

How do we solve the above equation for $\mathbf{v}$, when the definition for change in velocity includes velocity itself? We need to know the velocity over some period of time to figure out its derivative, but in order to figure out the current velocity, we need to know its derivative. It seems like we're stuck in a catch 22 situation. However, it is possible to solve that equation.

Equations like the above are called differential equations. Entire classes and texts are devoted to their solutions because most real-world physical systems can be modeled with them. If you've ever taken a differential equations course, you could probably tell me the closed form solution for velocity ($v(t) = e^{-bt/m}$ for the curious), but this requires a knowledge

of differential equations. How do we integrate forces in the general case, especially when the force is differential? Thankfully, computers are pretty good at numerically solving differential equations. This is where numerical integrators come into play.

4 Numerical Integration

4.1 Overview

Let's look at the previous example I gave about a force that is hard to calculate. The equation for simple viscous drag is a differential equation:

$$\dot{\mathbf{v}} = -\frac{b}{m}\mathbf{v}$$

Again, it is hard to integrate this function to find velocity (especially when other forces are involved), because the integral includes velocity itself. If we were to plot the 1 dimensional velocities of several particles with different initial velocities (v_0), we'd end up with something similar to the graph shown in Figure 7.

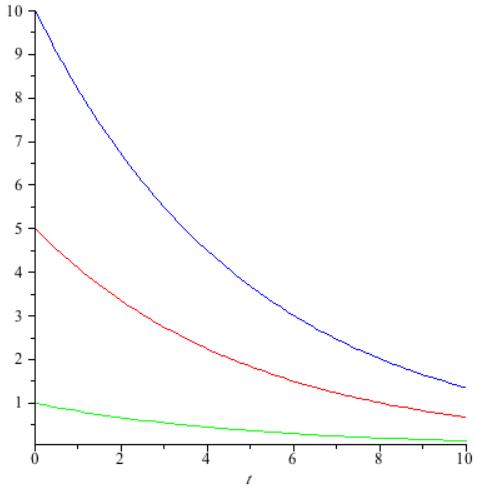


Figure 7: Plotting the velocities of particles with different initial velocities,$v_0 = 10$ (blue), $v_0 = 5$ (red) and $v_0 = 1$ (green). Note that $-b/m = 1/5$ in this case.

Now, I did provide the solution for velocity, but our goal is to integrate acceleration without finding a closed-form solution. Forget about solving velocity for all time, let's just examine the case when $t = 0$. For example we'll look at the blue curve in the above figure. At $t = 0$, $v = 10$. It's certainly possible to solve for acceleration *at time 0*, since we know

what velocity is *at time 0*. If we make the assumption that velocity doesn't change (which is a bad assumption), we can then solve for velocity by integrating acceleration:

$$\begin{aligned} \dot{v}(0) &= -\frac{1}{5}v(0) \\ \dot{v}(0) &= -2 \\ \int \dot{v}\ dt &= v = -2t + C \end{aligned}$$

(*Note:* We use C because the constant we add might not necessarily be the initial velocity itself. The correct constant C needs to be added to $v(t)$ to obtain the correct *initial value*.) Let's plot the solution we just came up with. It's shown in Figure 8. You can see that, overall, it's a pretty bad approximation. However, close to the point where we seeded our approximation, at $t = 0$, we come pretty close to the actual velocity. But overall, our initial assumption, that v was unchanging, is totally wrong. However, can we *recalculate* for velocity at a future time step? Remember, to come up with an approximate solution for $v(t)$ for all time, we need to know the value of $v(t)$ at some point in time. We can use our previous approximation to figure out $v(t)$ when $t = 2$:

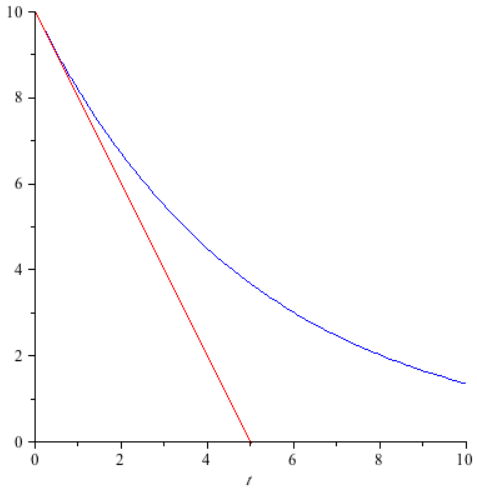


Figure 8: Plotting actual particle velocity versus approximated velocity.

$$\begin{aligned}
\dot{v}(2) &= -\frac{1}{5}v(2) \\
v(2) &= -2(2) + 10 = 6 \\
\dot{v}(2) &= -\frac{6}{5} \\
\int \dot{v}\ dt &= v = -\frac{6}{5}t + C
\end{aligned}$$

We know the velocity at $t = 2$ is 6 from the above plot. Thus, if we plot the updated approximation, we arrive at Figure 9. This approximation looks a little better. Let's keep doing the same thing for all multiples of $t = 2$. We get something that looks like Figure 10. (I switched from plotting in Maple, which does symbolic math, to Matlab, which I like better for numerical analysis).

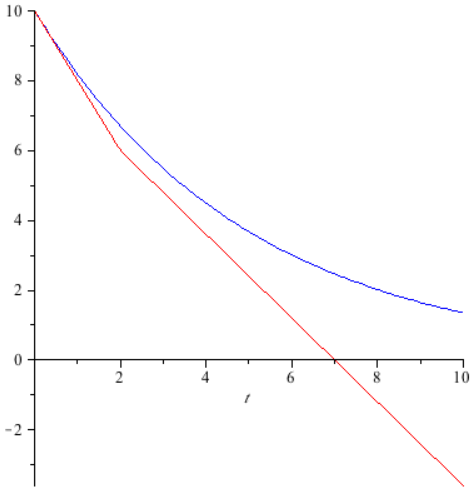


Figure 9: Plotting actual particle velocity versus approximated velocity, with two approximations instead of just one.

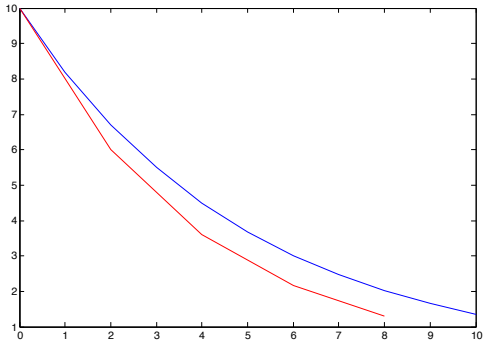


Figure 10: Plotting actual particle velocity versus approximated velocity, with an approximation step size of 2.

If you notice the caption to Figure 10, I mention the term step size. This is simply how often we skip ahead to approximate. In the above two figures, we used a step size of 2. This provided an okay approximation. But what if we wanted better resolution? What happens when we decrease the step size? We can see the results of using a step size of 1 in Figure 11.

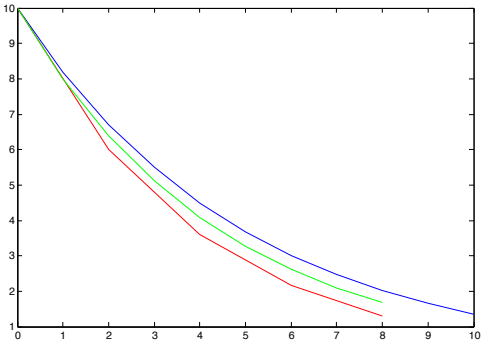


Figure 11: Plotting actual particle velocity versus approximated velocity, with an approximation step size of 2 (red) and 1 (green).

We can see that as we decrease the step size, the accuracy of our numerical integrator increases. However, as we increase our resolution (decrease the step size), the amount of calculations increases. This is an engineering tradeoff that must be experimented with further as we start to build our simulator.

Believe it or not, I did not invent the process I just detailed. It is actually a well known approximation technique called Euler's Method. Let's come up with a more rigorous definition

of Euler's Method, one that can be translated directly into code.

4.2 Euler's Method

At its core, Euler's Method is based on the geometric definition of the derivative of a curve: *the slope of its tangent line at a specific point.* It's a first order procedure for solving differential equations,[4] and it's probably the most basic numerical integration technique. We use it to solve for future values of a curve for which we know an initial value and something about the curve's derivative. Using the derivative, we know the slope of the curve at the initial point. Therefore we can come up with an equation for a line, starting at the initial value and with a slope calculated from the derivative. We can then do this same process for all subsequent points.

The basic iterative process behind Euler's Method is very simple. The steps can be outlined:

1. Get the value of the curve at the current point.
2. Calculate the slope of the tangent line at the current point using the derivative.
3. The value of the next point is the current point value, plus the slope times the step size.
4. Move the current point up 1 step.
5. Repeat.

The formula defining this process is given in Equation 6, with h being the step size, and n being the current step.

$$y(n+1) = y(n) + h\dot{y}(n) \tag{6}$$

4.3 Problems with Euler

That's it! Euler's method is very simple, but it does suffer from a lack of accuracy. As we saw in the previous section, we will always suffer some numerical drift when calculating curves using Euler's Method. The smaller the step size we use, the more accuracy we gain. However, no matter how small our step size, we will never arrive at the actual answer. We would have to use an infinitely small step size to do that.

In addition, Euler is prone to instability. That is, when we encounter certain types of forces, our solutions will blow up to infinity. For instance, think of a simple spring-mass system. Say our numerical integrator told us the position of the weight was a little bit farther than it actually was. Over time, these errors would add up. And since the force exerted by a spring is directly proportional to how far it has extended, the force would grow larger and larger, eventually causing the mass to shoot off into space (or into the center of the Earth)!

One way we can overcome this is by using damping forces. That is, we can suck a little bit of the energy out of our system each time step to make sure that nothing blows up. However, the forces we're going to be dealing with for the time being don't cause much instability. In addition, it would be more straightforward to use a better integrator in the first place. But we'll cross that bridge when we come to it. For now, Euler is good enough.

5 Putting it all together

Correctly coding a physics engine is just as important as understanding the theory behind it. In this section, we will go over one possible implementation of the ideas we have covered so far. First we will update our Body class to incorporate a force accumulator and a Euler integrator. Then, we will create a "World" class where our objects will live and die. In addition, we will create several unary Force classes, namely Gravity and Drag, that we can apply to all objects in our simulation. Finally, we will put together an example that simulates a fireworks show.

5.1 Updating the Body class

When we finished Module 1, we only had a bare-bones Body class that stored a position, velocity and acceleration vector. However, we now know that a body has another property, mass. This is simply a scalar. Appropriately enough, we will call it `mass`. In addition, since many operations include dividing something by the body's mass (specifically when applying a force), we also create an `inverseMass` property. In the future, we will be able to create immovable objects by making the `inverseMass` 0 (i.e., infinite mass).

In addition, we need to keep track of all the forces being applied to a body in a single time step. We create a `forceList` list to take care of this.

Finally, we need to incorporate Euler's method into our simulation. A good place to put this numerical integration is within the Body class itself, in a method called `update`, so that we can simply tell the Body to update itself without having to know its acceleration, velocity or acceleration.

This is probably the most important part of our code in this lesson. It is critical you understand the rather simplistic nature of the integrator in our body class. Remember the relation between acceleration and velocity:

$$\mathbf{a} = \dot{\mathbf{v}}$$

If we want to find the velocity of a particle one step from now, we can use Euler:

$$\begin{aligned} \mathbf{v}(n+1) &= \mathbf{v}(n) + \dot{\mathbf{v}}(n) \\ \mathbf{v}(n+1) &= \mathbf{v}(n) + \mathbf{a}(n) \end{aligned}$$

What this means is that to integrate velocity, all we have to do is add the current acceleration! To update position, all we have to do is add the current velocity. This is a very simple operation, and one that we implement in our Body class. This class then takes the following form:

```
from rbpmath import Vector

# This is the basic superclass for all bodies
class Body(object):
    def __init__(self, mass, position, velocity, acceleration):
        self.mass = mass
        self.inverseMass = 1.0/self.mass

        # Make sure position, velocity and acceleration are vectors
        if isinstance(position, Vector):
            self.pos = position
        if isinstance(velocity, Vector):
            self.vel = velocity
        if isinstance(acceleration, Vector):
            self.acc = acceleration

        self.forces=[]

    # Prints information about the body.
```

```
    def __str__(self):
                return  "Position: " + str(self.pos) + "\n"\
                "Velocity: " + str(self.vel) + "\n"\
                "Acceleration: " + str(self.acc) + "\n"

    def addForce(self, force):

    def update(self):
```

See if you can fill in the method stubs. Remember, when applying a force to an object, we simply sum the applied force vectors:

$$\mathbf{F} = \sum_i \mathbf{f_i}$$

When we know all the forces, we can divide by the mass to get a body's acceleration:

$$\mathbf{v} = \frac{1}{m}\mathbf{F}$$

The full Body class can be found in the code section and in the Module's `src/` directory.

5.2 A whole new World

Although we now have a body class that can represent simple zero-dimensional objects, we still need a place for them to reside. We need a controller object to update and apply forces to bodies, and something to destroy objects to limit memory usage. We will call this controller class `World`. Our `World` class has three main functions:

1. Adding/removing bodies
2. Adding unary forces to all objects
3. Updating/drawing all objects

Adding and removing bodies is rather trivial. We can simply use Python data structures to store bodies, and use the `index()` method to look up objects and remove them. We will use a simple list to keep track of bodies for now, but as we begin to refine our engine, we may switch to more complicated structures to increase performance.

We also have to add unary forces to all objects. Because each unary force has a different effect on objects, and may depend on a body's velocity or position, we will abstract the

actual force application into another class and call it through a generic method. We will keep track of our unary forces in another list as well.

Finally, we need to update our bodies. Remember, we put the actual update implementation in the bodies themselves, so all we have to do is iterate through our body list, and tell each body to update itself.

Our `World` class might look something like the following. See if you can fill in the method stubs. Again, the full listing is given in the Code section.

```
from rbpbodies import *
from rbpmath import *
from rbpforces import *

class World(object):
    def __init__(self):
        self.bodyList = []
        self.unaryForces = []

    def addBody(self, body):

    def removeBody(self, body):

    def clearBodyList(self):

    def addUnaryForce(self, force):

    def update(self):
```

5.3 Forcing the issue

In the last section I said that we will abstract unary force implementations away from the `World` class, because those implementations can vary widely. To do this, we will create a `Force` superclass that simply has an `applyForce` method that adds a force to all objects in the world.

The stub class is given below. For the `applyForce()` method, we pass a reference to the world's body list. Remember, unary forces are applied to all objects. However, the actual `applyForce()` implementation will differ between classes.

```
from rbpmath import *

class Force(object):
    def __init__(self):
        pass

    def applyForce(self, bodyList):
        pass
```

5.4 Ending with a bang

We now have enough to create a simple simulation that showcases Euler's method and differential forces. Our example will be a simple fireworks particle effect. When creating any simulation, it is important that we first spec out what exactly we want to model.

In this case, we will model a firework shell being launched into the air. After a certain period of time, it will burst, shooting many fireworks sparks in random directions. Affecting both the shells and the sparks are the forces due to gravity and drag. Remember Galileo's experiment - gravity affects both the firework shell and sparks equally. However, the force due to air resistance should affect each object differently, in an amount inversely proportional to their respective masses.

5.4.1 Model forces

Let's model gravity and drag. We will subclass the Force class and create different `applyForce()` implementations for each. In Gravity's constructor, we specify a vector, which as we already know, consists of a direction and a magnitude. Also remember that the definition for the force of gravity is:

$$\mathbf{F_g} = m\mathbf{g}$$

Here's a hint on how to implement gravity: since m appears in the force definition, we can ignore a body's mass in our calculation.

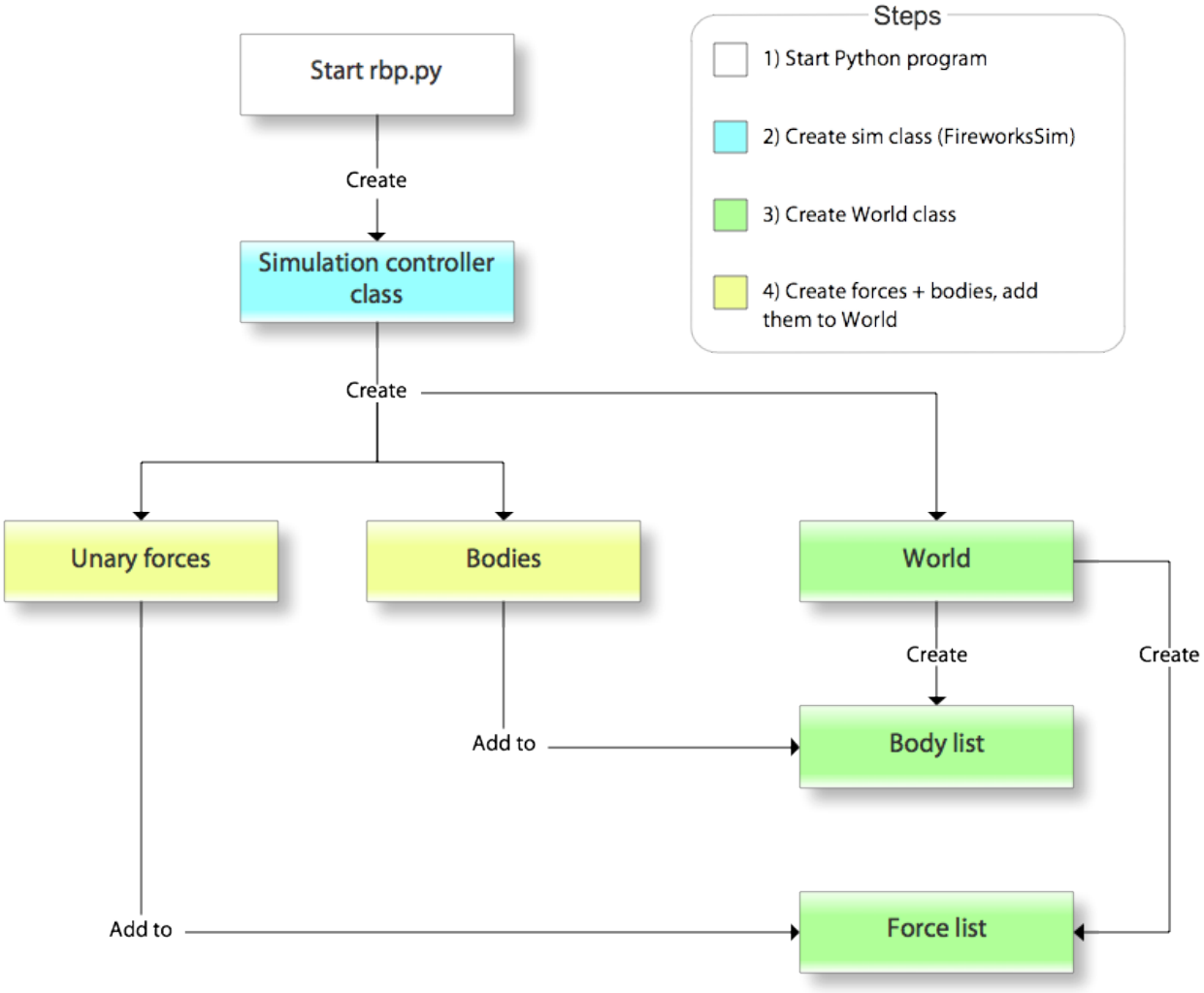
Next we want to implement drag. The definition for acceleration due to drag is as follows:

$$\mathbf{F_d} = -\frac{b}{m}\mathbf{v}$$

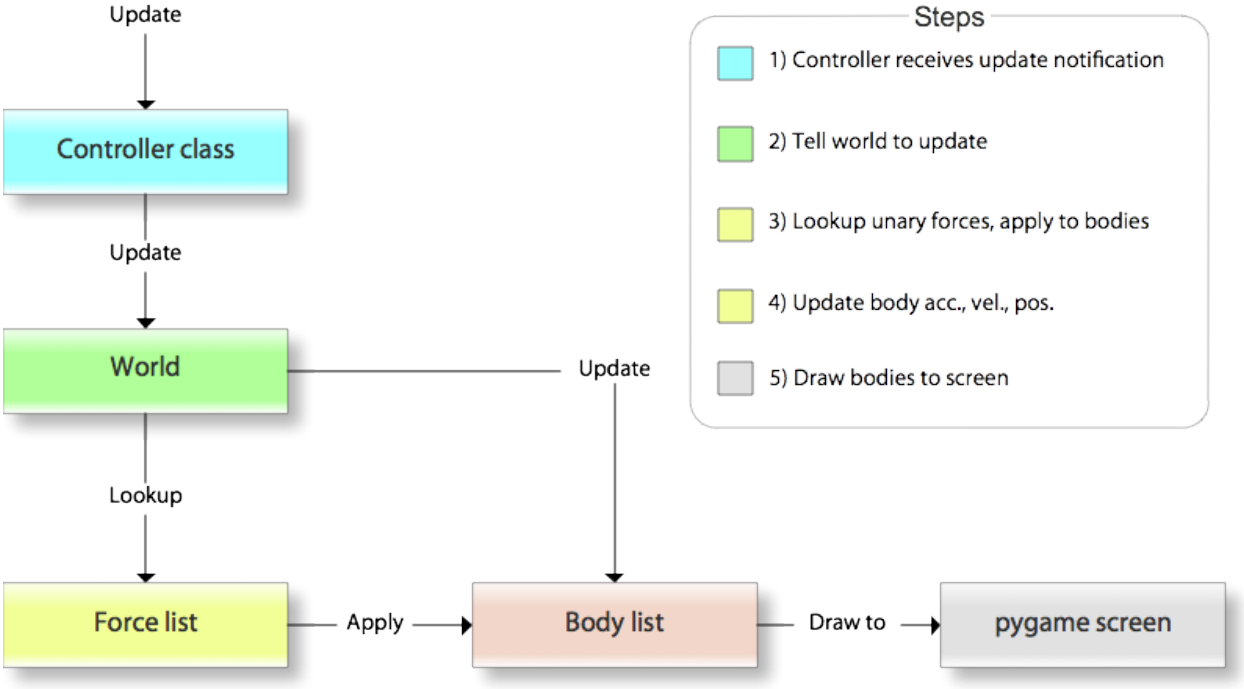
This is why we will specify a b constant in our contructor for the drag force. In addition, we can use the ideas of Euler to set the acceleration each step. We simply take the current velocity and scale it to figure out the drag force.

5.4.2 Model overview

Here's the basic dataflow model of our simulator which we'll be referring to in future lessons. These are the steps we go through when starting our simulation:



Next let's look at how we'll be updating our simulation every time step:



Hopefully these diagrams give you an overall picture of how we'll be running our simulation.

5.4.3 Final model

I'm not going to go through the rest of the code, as it is more game programming than physics programming. All the sources are listed at the end of this lesson. If you have all of them in one directory, and you run `python rbp.py`, the simulation should start. The code runs correctly, but there may be ways I can improve it. If you come across any bugs, be sure to let me know!

One thing that I want to point out is that in order to draw primitives to the screen, we need to have a reference to the pygame screen. Therefore I pass this to all objects so that their drawing methods can be self-contained.

6 Conclusion

We've covered a lot in this lesson, from forces and how they affect zero dimensional particles, to how we can model these in a computer simulation. The simulation that I've gone over here is just a simple example. See if you can add a more complex force, such as multiple gravity wells or some type of fluid dynamics. Go wild!

In the next lesson, we will move to full two dimensional objects and the rules that describe their motion.

7 Chapter summary

Review

Forces

$$\begin{aligned} \mathbf{F} &= m\mathbf{a} \\ \mathbf{a} &= \frac{1}{m}\sum_i \mathbf{f_i} \\ \mathbf{F_g} &= m\mathbf{g} \\ \mathbf{F_d} &= -\frac{b}{m}\mathbf{v} \end{aligned}$$

Euler's method

$$\begin{aligned} y(n+1) &= y(n) + h\dot{y}(n) \\ \mathbf{v}(n+1) &= \mathbf{v}(n) + \mathbf{a}(n) \\ \mathbf{s}(n+1) &= \mathbf{s}(n) + \mathbf{v}(n) \end{aligned}$$

8 Keyword Definitions

Euler's Method

A first-order numerical procedure used to solve differential equations, given an initial value.[4]

force

Any influence that causes a free body to undergo acceleration.[5]

mass

Inertial mass is the property that defines an object's resistance to force.

n-ary force

A force that acts on some particles. One particle may exert an n-ary force on another, or many other, particles.

numerical integrator

A system that finds the integral of a curve by dividing the curve into small segments, approximates the area under each segment, and sums the results. Integrates functions with hard-to-find closed-form solutions.

step size

In numerical analysis, how often a numerical solver approximates a solution. The smaller the step size, the more accurate the approximation. However, a smaller step size uses more computing power.

unary force

A force that acts on all particles. Think of a unary force as a force exerted by the universe on all particles within it. Unary forces can vary from particle to particle, but all particles experience unary forces to some degree.

9 Code

`rbp.py`

```
import pygame
from fireworks import FireworkSim

SPEED = 100             # The speed at which our simulation is run
SCREEN_WIDTH = 640      # The dimensions of the display window
SCREEN_HEIGHT = 480
C_BLACK = 0, 0, 0
```

```
pygame.init()
screen = pygame.display.set_mode( (SCREEN_WIDTH, SCREEN_HEIGHT) )
pygame.display.set_caption('I2RBP')

clock = pygame.time.Clock()

running = 1

fireworkSim = FireworkSim(screen)

while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = 0

    screen.fill(C_BLACK)

    fireworkSim.update()

    pygame.display.flip()
    clock.tick(SPEED)

pygame.quit()
```

rbpmath.py

```
import math

class Vector(object):
    # Initializes 2d vector.  Default is 0.
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def components(self):
        return (self.x, self.y)

    # Prints components of vector.
    def __str__(self):
```

```
    return "X: " + str(self.x) + " Y: " + str(self.y)

# Checks to see if the components of two vectors are equal.
def __eq__(self, other):
    if isinstance(other, Vector):
        return ((self.x == other.x) and (self.y == other.y))
    else:
        return NotImplemented

# Checks to see if the components are not equal.
def __ne__(self, other):
    eq = self.__eq__(other)
    if eq is NotImplemented:
        return not eq
    return NotImplemented

# Vector addition
def __add__(self, other):
    if isinstance(other, Vector):
        return Vector(self.x+other.x, self.y+other.y)
    else:
        return NotImplemented

# Vector addition and assignment
def __iadd__(self, other):
    if isinstance(other, Vector):
        self.x += other.x
        self.y += other.y
        return self
    else:
        return NotImplemented

# Vector subtraction
def __sub__(self, other):
    return Vector(self.x-other.x, self.y+other.y)

# Vector subtraction and assignment
def __isub__(self, other):
    if isinstance(other, Vector):
        self.x -= other.x
```

```
            self.y -= other.y
            return self
        else:
            return NotImplemented

    # Dot product
    def __mul__(self, other):
        if isinstance(other, Vector):
            return (self.x*other.x + self.y*other.y)
        elif isinstance(other, int) or isinstance(other, float):
            return Vector(self.x * other, self.y * other)
        else:
            return NotImplemented

    # Scalar multiplication and assignment
    def __imul__(self, a):
        self.x *= a
        self.y *= a
        return self

    # Scalar multiplication
    def scale(self, a):
        return Vector(self.x*a, self.y*a)

    # Cross product
    # In 2d it is a scalar
    def __mod__(self, other):
        if isinstance(other, Vector):
            return (self.x*other.y - self.y*other.x)
        else:
            return NotImplemented

    # Zeros the vector
    def zero(self):
        self.x = 0
        self.y = 0

    # Returns the magnitude of the vector.
    def magnitude(self):
        return math.sqrt(self.x*self.x + self.y*self.y)
```

```
    # Returns tuple of components for drawing functions
    def components(self):
        return (self.x, self.y)
```

rbpbodies.py

```
from rbpmath import Vector

# This is the basic superclass for all bodies
class Body(object):
    def __init__(self, mass, position, velocity, acceleration):
        self.mass = mass
        self.inverseMass = 1.0/self.mass
        # Make sure position, velocity and acceleration are vectors
        if isinstance(position, Vector):
            self.pos = position
        if isinstance(velocity, Vector):
            self.vel = velocity
        if isinstance(acceleration, Vector):
            self.acc = acceleration

        self.forces=[]

        # Prints information about the body.
    def __str__(self):
                return  "Position: " + str(self.pos) + "\n"\
                "Velocity: " + str(self.vel) + "\n"\
                "Acceleration: " + str(self.acc) + "\n"

    def addForce(self, force):
        if isinstance(force, Vector):
            self.forces.append(force.scale(self.inverseMass))

    def update(self):
        for force in self.forces:
            self.acc += force

        self.vel += self.acc
```

```
        self.pos += self.vel

        self.acc.zero()
        self.forces = []
```

rbpworld.py

```
from rbpbodies import *
from rbpmath import *
from rbpforces import *

class World(object):
    def __init__(self):
        self.bodyList = []
        self.unaryForceList = []
        self.unaryForces = []

        self.drawOffScreen = True

    def addBody(self, body):
        if isinstance(body, Body):
            self.bodyList.append(body)

    def removeBody(self, body):
        if isinstance(body, Body):
            try:
                self.bodyList.index(body)
                self.bodyList.remove(body)
            except ValueError:
                print "Could not delete body " + str(body)

    def clearBodyList(self):
        self.bodyList = []

    def addUnaryForce(self, force):
        if isinstance(force, Force):
            force.bodyList = self.bodyList
            self.unaryForces.append(force)
```

```
    def update(self):
        for force in self.unaryForces:
            force.applyForce(self.bodyList)

        for body in self.bodyList:
            body.update()
            if body.pos.x>=0 and body.pos.x<=640:
                if body.pos.y>=0 and body.pos.y<=480:
                    body.draw()
```

rbpforces.py

```
from rbpmath import *

class Force(object):
    def __init__(self):
        pass

    def applyForce(self, bodyList):
        pass

class Gravity(Force):
    def __init__(self, forceVector):
        Force.__init__(self)
        if isinstance(forceVector, Vector):
            self.forceVector = forceVector
        else:
            self.forceVector = Vector(0, 0)

    def applyForce(self, bodyList):
        for body in bodyList:
            body.acc += self.forceVector

class Drag(Force):
    def __init__(self, b):
        Force.__init__(self)
        self.b = b

    def applyForce(self, bodyList):
```

```
        for body in bodyList:
            print body.vel
            body.addForce(body.vel.scale(-self.b))
```

fireworks.py

```
import pygame, random, math
from rbpbodies import Body
from rbpmath import *
from rbpworld import *
from rbpforces import *

class FireworkSim(object):
    def __init__(self, screen):
        self.world = World()
        gravity = Gravity(Vector(0, 0.02))
        drag = Drag(.0002)
        self.world.addUnaryForce(gravity)
        self.world.addUnaryForce(drag)

        random.seed()

        self.screen = screen
        self.fTimer = 0

    def update(self):
        self.world.update()

        if self.fTimer > 0:
            self.fTimer -= 1
        else:
            self.world.clearBodyList()

            fireworkShell = FireworkShell(  self.screen, self.world, 1,
                                            Vector(random.randint(0, 640), 490),
                                            Vector(random.random()*2-1, -4),
                                            Vector(0, 0),
                                            200)
            self.world.addBody(fireworkShell)
```

```
            self.fTimer = 400

class FireworkShell(Body):
    color = [255, 255, 255]
    size = 5

    def __init__(self, screen, world,
            mass, position, velocity, acceleration, lifeTimer):
        Body.__init__(self, mass, position, velocity, acceleration)
        self.screen = screen
        self.world = world
        self.lifeTimer = lifeTimer
        self.blownUp = False

    def update(self):
        Body.update(self)

        if self.lifeTimer > 0:
            self.lifeTimer -= 1
        else:
            if not self.blownUp:
                newColor = [random.randint(0, 255),
                            random.randint(0, 255),
                            random.randint(0, 255)]
                for i in range(1,50):
                    randDir = random.random()*2*math.pi
                    randMag = random.random()*2
                    fireworkSpark = FireworkSpark(self.screen, self.world,
                                        newColor, .01,
                                        Vector(self.pos.x, self.pos.y),
                                        Vector(randMag*math.cos(randDir),
                                                randMag*math.sin(randDir)),
                                        Vector(0, 0),
                                        random.random()*50+100)
                    self.world.addBody(fireworkSpark)

                self.blownUp = True

    def draw(self):
        if not self.blownUp:
```

```
            pygame.draw.circle(self.screen, self.color,
                    self.pos.components(), self.size)

class FireworkSpark(Body):
    def __init__(self, screen, world, color,
                    mass, position, velocity, acceleration, lifeTimer):
        Body.__init__(self, mass, position, velocity, acceleration)
        self.screen = screen
        self.world = world
        self.color = color
        self.lifeTimer = lifeTimer
        self.size = 2

    def update(self):
        Body.update(self)

        if self.lifeTimer > 0:
            self.lifeTimer -= 1
        else:
            self.world.removeBody(self)

    def draw(self):
        pygame.draw.circle(self.screen, self.color,
                self.pos.components(), self.size)
```

10 References

1. http://csep10.phys.utk.edu/astr161/lect/history/newton3laws.html
2. http://chrishecker.com/images/d/df/Gdmphys1.pdf
3. http://www.cs.cmu.edu/afs/cs/user/baraff/www/pbm/particles.pdf
4. http://en.wikipedia.org/wiki/Euler_method
5. http://en.wikipedia.org/wiki/Force