# Introduction To Rigid Body Physics

## Modules 3 and 4: Rigid Bodies

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 the motion of rigid bodies and how various forces affect them. Rigid bodies differ from zero-dimensional particles in that an extra degree of freedom, rotation, is introduced. The math in this chapter is a bit more involved than in previous lessons, so most of our time will be spent deriving equations. After learning the theory describing how rigid bodies behave, we will code a simulation that applies a rubber band force to a box, which will cause it to spin and translate. Less time will be spent on how to code our simulation than in previous chapters. The code simply implements the theory, so it is more important that you understand the theory in the first place. Throughout this reading, I try to be as language-agnostic as possible. if you have coding questions, be sure to ask them on the IntroToRBP subreddit.

# 2 Suggested reading

Like the last lesson, I again provide a reference to Chris Hecker's excellent series on rigid body physics. His next column covers most of what we will go over in this lesson, albeit in less detail as he was limited to 4 or 5 pages. The second link provides a simulation that is similar to what we will eventually develop by the end of this lesson. In addition, there is a brief overview of the math involved. We will not cover collisions in this module as that is the subject of the next two lessons. The final link is to Witkin et al.'s series on physical modelling.

- [PDF] http://chrishecker.com/images/c/c2/Gdmphys2.pdf
- http://www.myphysicslab.com/collision.html
- [PDF] http://www.cs.cmu.edu/afs/cs/user/baraff/www/pbm/rigid1.pdf

# 3 The equations of motion of rigid bodies

## 3.1 Overview

Up until now, we have been dealing with theoretical "zero-volume" particles. While point particles are very useful in theoretical physics[1] (and in some simulations), we'd like to simulate objects we are more familiar with, like blocks or vehicles. The difference between rigid bodies and point particles is that we can't describe a rigid body using only a position. For

point masses, simply giving the position (and mass) at a certain point in time allowed us to perfectly recreate the particle at that given point in time. However, with rigid bodies, we must introduce a new concept, orientation.

Figure 1: Some rigid bodies. Not to scale.

In the real world, if you were to throw a three-dimensional object in the air it would probably spin around. At any point in time, you could describe how much it has spun since you first let go of it. We call this measure of rotation from some arbitrary starting point *orientation*. In three dimensions, we need at least three values to describe the orientation of an object. However, in two dimensions, our object can only spin clockwise or counter-clockwise. We can quantify this rotation using degrees, and hence we only need a scalar! This fact will greatly simplify our angular equations of motion for two-dimensional rigid bodies.

Although we certainly need to describe a rigid body's orientation, we may be getting ahead of ourselves. How do we mathematically define a rigid body's linear displacement, velocity and acceleration? How do we update each quantity? Well as it turns out, the equations we derived in the previous lessons aren't useless - the linear motion of rigid bodies can be calculated in the same way we calculate the linear motion of zero-dimensional particles!

We'll discover in this section that orientation is the *angular* analogue to linear displacement. Are there angular properties that are similar to linear velocity and acceleration? There are! We call these angular velocity and acceleration (original, right?). And are orientation, angular velocity and angular acceleration all related, as they are with particles? We will determine the answer to that question shortly.

### 3.2 Linear motion of rigid bodies

For now, let's ignore the fact that a rigid body has an orientation in space. In fact, for the sake of example, let's imagine the simplest rigid body - a simple circle. Orientation *doesn't matter*. Therefore, we can ignore it and focus on the *linear* (as opposed to *angular*) motion

of the circle. Linear motion concerns the change in location of an object, and is not related to the orientation or rotation of that object.

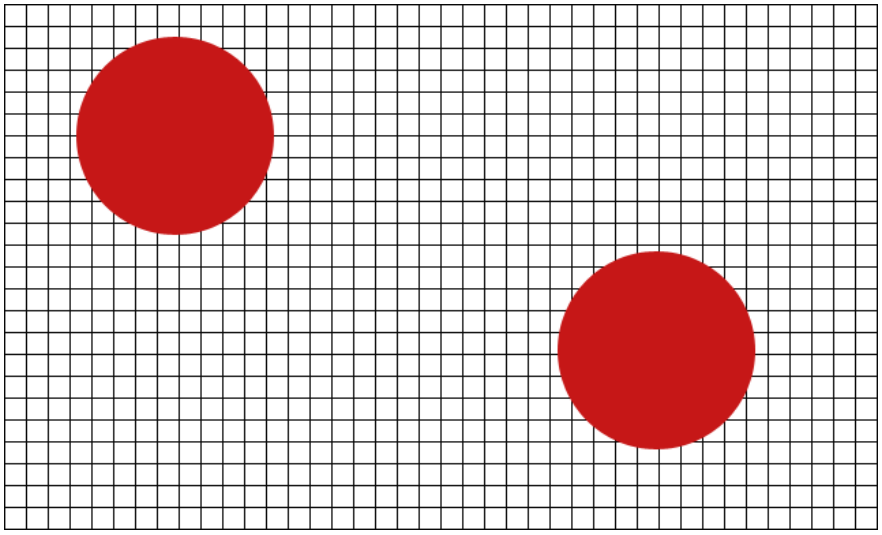

Figure 2: Can you tell the difference between these two objects? In fact, the second one is rotated 180 degrees!

The main difference between our circular rigid bodies and the particles we examined earlier is the concept of volume (or area in the 2D case). Rigid bodies take up "space." That is, we can't describe a rigid body by its position alone. Rather, we need to include additional information about its shape. In the case of our circular objects, that extra dimension can be described by just specifying its radius. If we know our body is a circle, merely knowing its **radius and position** is enough to describe it *spatially*. More complicated shapes obviously require more information to describe them. A piece of a broken wall could be a complicated polygon with 30 vertices, each of which needs to be stored individually. Knowing how to efficiently describe rigid bodies is an important skill we will develop as we begin simulating rigid body dynamics.

But what does this have to do with the linear motion of rigid bodies? Well, one way we could describe the body is by stating that it is simply a collection of small, bound particles. The object, $b$, is a sum of parts, $p_i$. Mathematically, this is shown in Equation 1.

$$b = \sum_{i}^{N} p_i \tag{1}$$

But since the component particles themselves have no dimension, how can their sum take up any space? If we sum an infinite number of them, we still have an object with 0 volume. What we really mean to say is that our body is comprised of a *continuum* of particles. It is a **distribution** of a **mass** ($m$) over a **space** ($\int dA$):

$$b = \int m \, dA$$
$$b = m \int dA$$

| ***Integration?!*** |
| --- |
| Don't worry if integration isn't a familiar concept - we're not going to be using integrals directly in our code. In fact, you're not going to see them again in this chapter! However, it's important to have a formal mathematical definition of our bodies. |

It's here that we introduce an important concept - the center of mass. The center of mass greatly reduces the complexity of our calculations and allows us to treat the linear motion of a rigid body like a particle. **The linear motion of a rigid body, no matter how it is oriented, is *exactly the same* as a particle with the same mass located at the body's center of mass**. That is, if we know the acceleration of a rigid body's center of mass, we can update the linear velocity and position of the center of mass accordingly.

The center of mass is the mean location of all the mass in a system.[3] . If we were to treat our body like a sum of particles as we did in Equation 1, then the center of mass (remember, it's a location and can be represented by a vector), given by $\mathbf{s_{cm}}$ would be mathematically defined by Equation 2 (where $M$ is the total mass of the body).

| ***Mass Continuum vs. Sum of Masses*** |
| --- |
| We treat the body as a sum of parts instead of a mass continuum because that is how we will represent them in our code. Our bodies don't have infinite resolution as that would require infinite memory. Instead, we can think of each individual point mass as a single pixel on our simulation screen, because that is the highest visual resolution we are capable of producing. More importantly, it is much simpler to carry out summations in our code than it is to find closed form solutions to (sometimes complicated) integrals. |

$$\mathbf{s_{cm}} = \frac{1}{N}\sum_{i}^{N} m_i \mathbf{s_i} \tag{2}$$

Each particle $i$ is simply defined by a weighted vector; that is, its position $\mathbf{s_i}$ scaled by its mass $m_i$. We divide by the total number of particles $N$ in order to normalize the sum. In our case, we will be dealing with bodies with uniform density, so our center of mass equation simplifies to:

$$\mathbf{s_{cm}} = \frac{1}{N}\sum_{i}^{N} \mathbf{s_i} \tag{3}$$

This is simply the average position of the particles in the body. An interesting result appears if we take the derivative of both sides of Equation 3 with respect to time. Remember that vector addition is a linear operation so we don't have to do anything tricky when we take the derivative. We could explicitly write out every vector in the summation in Equation 3, derive them, and then recombine them, but we would end up with the same result as simply deriving whatever is in the summation outright. After taking the derivative, we arrive at:

$$\mathbf{v_{cm}} = \frac{1}{N}\sum_{i}^{N} \mathbf{v_i} \tag{4}$$

What Equation 4 reveals is that the *total velocity of our system of particles* (the rigid body) is equal to the *velocity of the center of mass*! This is not entirely surprising, as we specifically chose to define the center of mass in such a way that our equations of motion for a rigid body are greatly simplified. We can represent this finding graphically in Figure 3.

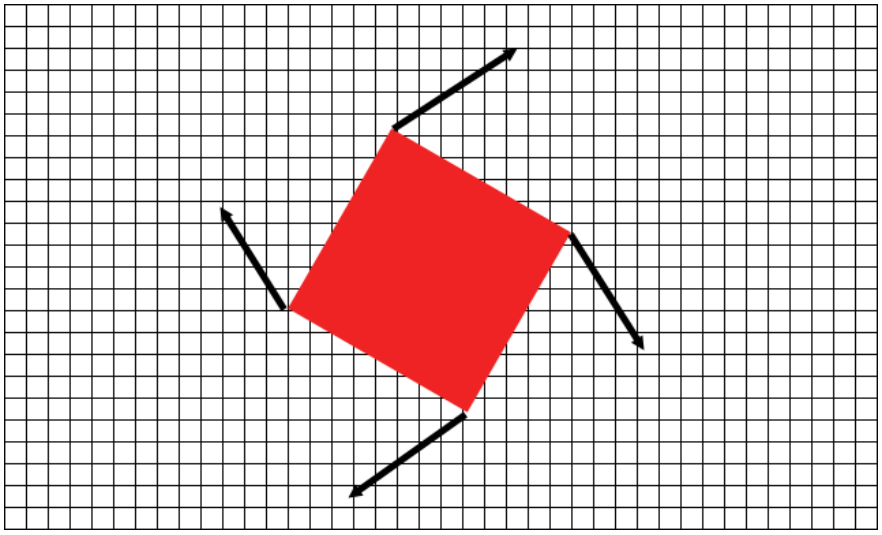

Figure 3: The individual velocity vectors of each corner of the rotating square.

Let's say the square shown in Figure 3 was stationary, but rotating. If we were to add the velocity components of the opposite corners of the square, they would add to 0. In fact, for every point on that square, there is a point moving with an equal speed, but in the opposite direction. So if you were to sum the velocity vectors of all the points on the square, you would get 0. This confirms our result, as the velocity of the center of mass is also 0.

| ***Is symmetry required?*** |
| --- |
| You might think that this would not work for non-symmetric objects. However, this is not true! If you want to take a sneak peak at the proof, see Section 4.3.2. |

If the square were not only rotating, but translating as well, each "particle" in the body would have a rotational velocity (like those shown in Figure 3) and a translational velocity. The rotational velocities would all add to 0, while the translational velocities would sum to the velocity of the center of mass.

If we take the derivative of Equation 4 again we get:

$$\mathbf{a_{cm}} = \frac{1}{N}\sum_{i}^{N}\mathbf{a_i} \tag{5}$$

The total acceleration of our body is equal to the acceleration of the center of mass. This is an important finding that we will revisit when we examine the effect of forces on our rigid body.

We define the position of a rigid body as the location of its center of mass. In what is hopefully apparent from the preceding equations, using the center of mass simplifies a lot of the linear and angular equations of a rigid body. This will become clearer soon.

These equations prove that we can describe the *linear* motion of our rigid body in the same way we describe the *linear* motion of a particle. (Remember, angular motion is a separate concept which we will cover shortly.) Since the center of mass is simply a location, and doesn't contain any spatial information, it is equivalent to a zero-dimensional particle. Our previous equations of motion still apply. Therefore, if we know the total acceleration of our body, we know the acceleration of our center of mass, and in turn, the translational aspects of our body. In case you've forgotten the previous equations of linear motion, they are reproduced here:

$$\begin{aligned}
\mathbf{s} &= \mathbf{v}t + \mathbf{s_0} \\
\mathbf{v} &= \mathbf{a}t + \mathbf{v_0} \\
\dot{\mathbf{s}} &= \mathbf{v} \\
\dot{\mathbf{v}} &= \mathbf{a}
\end{aligned}$$

### 3.3 Angular motion of rigid bodies

Remember that the property of a rigid body that differentiates it a zero-dimensional particle is the concept that it occupies space. The rigid body occupies two dimensions, so in order to completely describe a body's instantaneous state, we need to know more than just its linear position, velocity and acceleration. Rigid bodies have *angular* (orientation) and *spatial* properties (size, location of vertices) in addition to *linear* properties (position, velocity, etc.).

For now we will ignore the spatial properties of a rigid body. When we learn about the *moment of inertia* we will need to know the dimensions of our body. For now, we will only

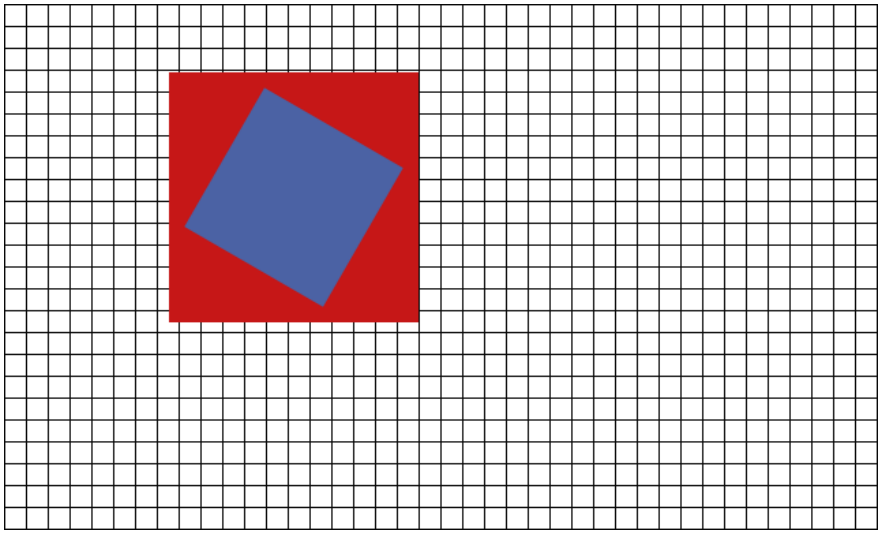

Figure 4: Although these squares have the same position and may have the same velocities and accelerations, physically they cannot be fully described by the same set of numbers, as they have different orientations and dimensions.

examine angular properties.

As I stated earlier, there is an angular analogue for each of the linear properties (position, velocity and acceleration). They are, respectively, orientation, angular velocity and angular acceleration. In the following subsections we will cover each.

#### 3.3.1 Orientation

The orientation of a body is the angular equivalent of position. Remember that the orientation of a body is how much it has rotated from some arbitrary starting point, which is similar to linear displacement. How we represent this property is important. We want to use the least amount of information possible to describe orientation for our simulation in order to be efficient. One way we could describe the rotation of a square is to list the current position of each of its vertices:

However, this is a waste of resources for a number of reasons. First, if the body were indeed a square, we wouldn't need to store four vertices. We could store three and determine the fourth dynamically. In addition, as our objects grow more complex, as we update each vertex individually, we are prone to numeric drift. Our object would become skewed over time leading to distorted bodies. We need a way to store our orientation simply and in a

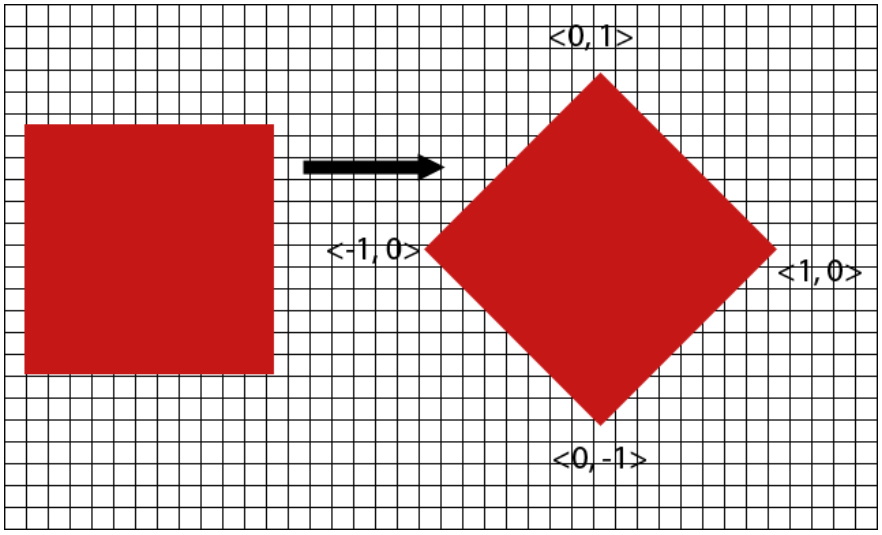


Figure 5: As the square rotates, we must update each vertex individually.

way that is largely immune to numeric drift.

Because we are dealing with rigid bodies, the points of a body don't change in position relative to each other. If one point were to rotate, all points rotate with it. There is only one dimensional configuration of points for all conceivable rotations of our body. That is, the shape of the rigid body *doesn't change* no matter how much we rotate it. Consequently, it is sufficient to store the rotation only.

Because we are building a two-dimensional simulator, there are only two ways we can rotate our body, clockwise and counter-clockwise. You can think of these rotations as positive and negative instead of clockwise or counter-clockwise. This means that we can represent orientation as a scalar! The symbol we use for the orientation of a rigid body is upper case omega, $\Omega$.

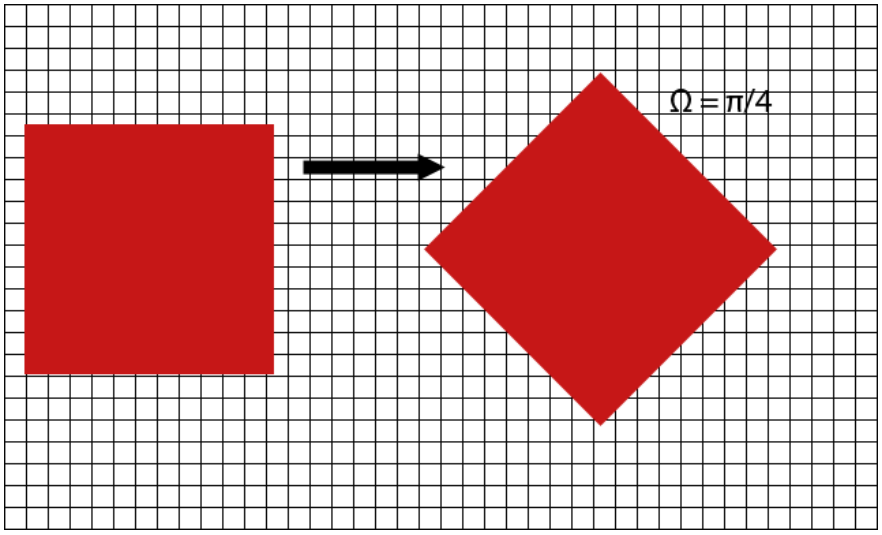


Figure 6: As the square rotates, we simply update a scalar.

| ***Note on radians*** |
| --- |
| If you are confused by the use of $\pi$ in Figure 6 instead of degrees, it is because many trigonometric functions in Python (and most other languages) use *radians* as input instead of *degrees*. A radian is simply another unit of angular measure.<br><br>One full rotation of an object is equivalent to $2\pi$ radians, as opposed to 360 degrees. Therefore, a half-rotation is simply $\pi$ radians instead of 180 degrees, and a quarter rotation is $\pi/2$ radians. The use of radians is rooted in the definition of $\pi$ itself, the ratio of a circle's circumference to its radius. One radian equals $180/\pi$ degrees. |

#### 3.3.2 Angular velocity

The definition of linear velocity is simply the rate of change of linear position, or:

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

It would make sense, then, that the definition of angular velocity is simply the rate of change of orientation. We represent angular velocity with a lower case omega, $\omega$. The mathematical

definition is:

$$\omega = \dot{\Omega}$$

The derivative of a one-dimensional quantity is also a one-dimensional quantity, so angular velocity is also a scalar.

#### 3.3.3 Angular acceleration

Angular acceleration is also similar to linear acceleration, in that angular acceleration, denoted by a lower case alpha, $\alpha$, is the rate of change of angular velocity. Mathematically, the three angular properties are related by the following:

$$\alpha = \dot{\omega} = \ddot{\Omega}$$

Again, angular acceleration is a scalar. Because angular orientation, velocity and acceleration are related by the derivative, we can use Euler's method to update each. If we know a body's angular acceleration, we can update its velocity and orientation. However, like in the last module, we need to figure out what influences a rigid body's angular acceleration.

# 4 Forces and rigid bodies

## 4.1 Overview

Most introductory physics courses end before showing how forces affect rigid bodies. While the concepts of angular rotation (and perhaps torque) might be glossed over, the general dynamics of rigid bodies in reaction to outside forces usually isn't covered. This is where the course begins to add on to what you might have already learned in high school and college. The mathematics become a little more complex than what we have experienced, but if you've made it this far then you shouldn't have any problem with the proofs.

This is one of the most important sections in the entire course. Although we all qualitatively know how a box might react if you were to push the corner of it, we will learn how to quantitatively describe that reaction. Implementing body dynamics takes our simulator from a flashy toy that can only simulate fireworks to a real (albeit rudimentary) physics engine. Therefore it is important you understand the concepts covered in this and the preceding section so that you can correctly implement dynamics (especially if you are developing your engine in a language other than Python).

A rigid body has **two** types of reactions to a force: a **linear** reaction (a change in position, velocity and acceleration) and an **angular** reaction (a change in orientation, angular velocity and angular acceleration). Remember that the linear reaction of a body is exactly the same as the linear reaction of a particle. In fact, we simply sum all the forces acting on a rigid body (no matter where on the body they are applied) and update the body as if it were a particle located at the body's center of mass. The angular effects are a bit more complicated as they rely not only the magnitude and direction of the applied forces, but also *where* on the body they act. In the following subsections I will go over each reaction type.

## 4.2 Linear reaction

I just made a rather bold statement: a rigid body reacts to force in the same way a particle would react to that force (if that particle was located at the body's center of mass, and had the same mass as the body). That is, to find out how to update the body's linear acceleration, we simply sum all the forces acting on the body. Recall Newton's second law:

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

To determine an object's resultant acceleration, we divide the force by the object's mass:

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

To get a body's acceleration in a certain time frame, we sum all the forces acting on the body and divide by its mass:

$$\mathbf{a} = \frac{1}{M}\sum_{i}^{N}\mathbf{f_i} \tag{6}$$

Another way we can write this is by individually determining the accelerations due to each force, and summing the accelerations themselves:

$$\mathbf{a_b} = \sum_{i}^{N}\mathbf{a_i}$$

To finish this proof, we will use another concept, momentum. Let's first outline what momentum is, and why it is useful.

### 4.2.1 Detour: linear momentum

The linear momentum $\mathbf{p}$ of a particle is a property we will use in our calculations. It is defined as:

$$\mathbf{p} = m\mathbf{v} \tag{7}$$

From the definition you can see that the momentum of an object is its velocity vector scaled by its mass. Why is momentum important?

You can think of momentum as the ability of an object to give or absorb energy. In layman's term, the more momentum an object hitting you has, the more it will hurt. A small particle moving *very* quickly has a lot of momentum. Chips of paint in orbit around the Earth certainly don't have a lot of mass, but they are moving extremely fast and will easily put a hole in a satellite or an astronaut. Similarly, although I can't toss a bowling ball very quickly, if I were to throw one at your head it would probably leave a nasty bruise.

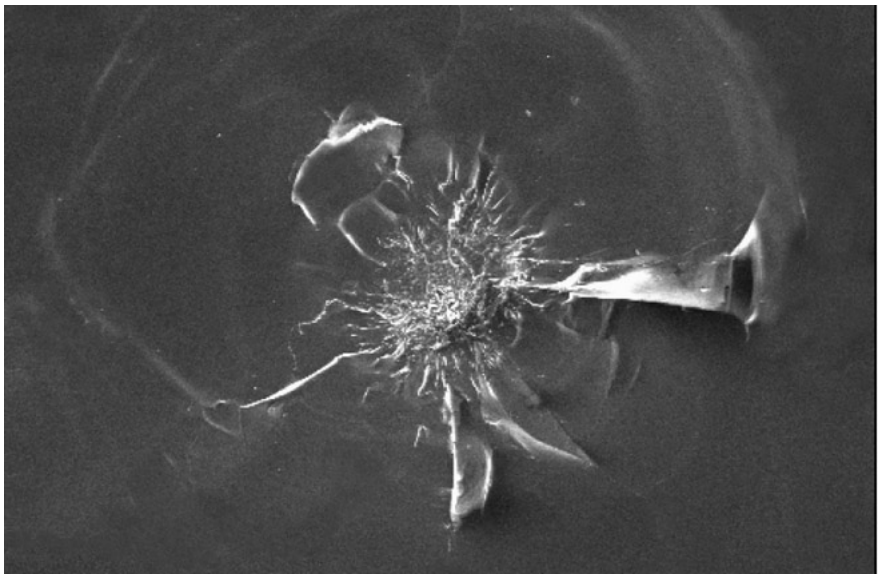

Figure 7: A scanning electron microscope image of a space shuttle window that was hit by a "particle of spacecraft paint." *source: http://ston.jsc.nasa.gov/collections/TRS/_techrep/TP-2006-213716.pdf*

We can tie momentum into the other concepts we have developed thus far by noting that the derivative of momentum is force:

$$\begin{aligned} \mathbf{F} &= m\mathbf{a} \\ \dot{\mathbf{p}} &= m\mathbf{a} \end{aligned}$$

The momentum of an object is useful in determining what happens when it collides with something. However, we are not going to cover collisions until the next lesson. So why are we covering momentum now? We will use it to finish the above proof.

#### 4.2.2 Further exploration of the linear reaction of a rigid body

Recall that the momentum of an object can be described by a vector. Therefore, the rules of summing vectors apply here. If we wanted to determine the total *linear* momentum of a rigid body ($\mathbf{p_b}$), we could sum the velocities of each particle scaled by the mass of each particle:

$$\mathbf{p_b} = \sum_i^N m_i \mathbf{v_i}$$

If we were to sum the masses of each particle, we would arrive at the total mass of the rigid body itself. Therefore:

$$\begin{aligned} \mathbf{p_b} &= \sum_i^N m_i \sum_i \mathbf{v_i} \\ \mathbf{p_b} &= M \sum_i^N \mathbf{v_i} \end{aligned}$$

Remember that the derivative of momentum is force. The derivative of the total momentum of a body is the total force acting on that body. Let's derive both sides of the above equation:

$$\mathbf{F_b} = M \sum_i^N \mathbf{a_i} \tag{8}$$

Also remember from the last section that the sum of the accelerations of all the particles in a body is equivalent to the acceleration of the center of mass:

$$\mathbf{F_b} = M\mathbf{a_{cm}}$$

Look at that! We've figured out what the linear reaction of a rigid body is in response to all the forces acting on it. Formally:

$$\mathbf{a_{cm}} = \frac{\mathbf{F_b}}{M} \tag{9}$$

It doesn't matter where the forces are applied to the body. For instance, forces with equal magnitude and direction will affect the position of a rigid body in the same manner, whether they are applied to the edge of the object or somewhere in the middle. A bullet hitting the side of a box causes the box to translate the same as gravity, as long as both applied forces are equal in magnitude and direction.

Now that we know the linear reaction of a rigid body to applied force, we need to derive the equations that govern the angular reaction. The angular reaction is a bit trickier than the linear reaction, as the location of the applied force *does* matter.

## 4.3 Angular reaction

We are going to use several building blocks to derive the angular reaction of a body due to an applied force. However, since this is an introductory course, I am assuming that you are not intimately familiar with many of the terms and concepts I am going to be using. Therefore, I need to describe each building block separately, and then use them to finally derive the angular reaction. The building blocks I keep talking about are perpendicular vectors, the velocity of a point on a rotating body, torque, the moment of inertia and angular momentum.

### 4.3.1 Perpendicular vectors

Vectors are perpendicular if the angle between them is 90 degrees.[4] We'll be using perpendicular vectors a lot when we derive the equations governing the angular reaction of rigid bodies. It is relatively simple to define a vector perpendicular to a given vector in two dimensions. A vector ($\mathbf{b}$) that is rotated 90 degrees counter clockwise from a given starting vector ($\mathbf{a}$) can be found in the following manner:

$$\begin{aligned} b_x &= -a_y \\ b_y &= a_x \end{aligned}$$

And to find a perpendicular vector that is rotated 90 degrees clockwise, do the following:

$$\begin{aligned} b_x &= a_y \\ b_y &= -a_x \end{aligned}$$

We denote perpendicular vectors using the following mathematical symbols:

$$\mathbf{b} = \mathbf{a}_\perp$$

Graphically, perpendicular vectors are shown in Figure 8.

Unless explicitly mentioned, if $\mathbf{b} = \mathbf{a}_\perp$ we assume that $\mathbf{b}$ is rotated 90 degrees counter-clockwise from $\mathbf{a}$.

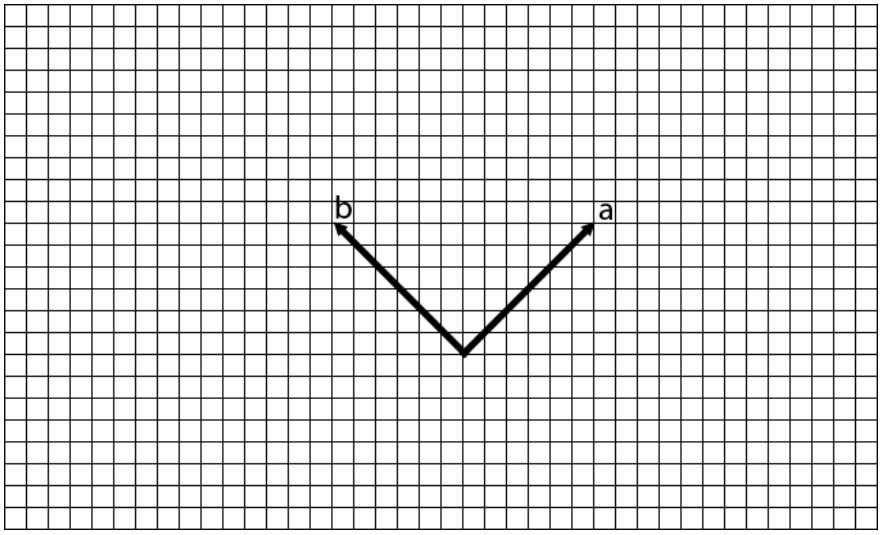


Figure 8: The vector $\mathbf{b}$ is perpendicular to $\mathbf{a}$, or $\mathbf{b} = \mathbf{a}_\perp$

***Perp dot product***

A common operation which we will use later in this section is the perp dot product. The operations used in this process are given in the name itself. If we wish to take the perp dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$, we simply find the dot product between $\mathbf{a}_\perp$ and $\mathbf{b}$. The symbol we use for the perp dot product operation is $\perp \cdot$. Graphically, taking the perp dot product between two vectors results is shown in Figure 9.

Remember that the dot product produces a *scalar*, not a vector. The red vector shown in Figure 9 is simply the magnitude produced by the dot product projected along the length of $\mathbf{a}_\perp$.

#### 4.3.2 Linear velocity of a point on a rotating body

Later when we're dealing with specific locations on a rotating rigid body, specifically for collisions, it will be important to know the instantaneous linear velocity of that particular point. This is illustrated in Figure 10.

In deriving our equation for the linear velocity of a point on a rotating body, it is impor-

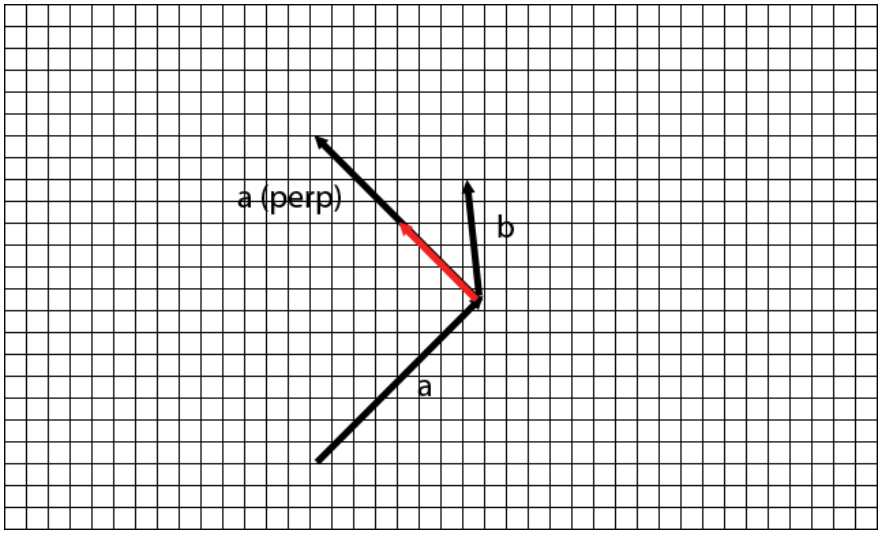


Figure 9: The perp dot product, projected onto $\mathbf{a}_\perp$, is in red.

tant to note several qualitative factors. First, the instantaneous linear velocity of a point on a body is *not* a scalar. Linear velocity is different from angular velocity in that is has both a magnitude and a direction. Secondly, although a rigid body can only have one angular velocity, every point on that body has a different instantaneous linear velocity. The horses that are near the edges of a merry-go-round move quicker than those closer to the center. Likewise, points farther from the center of mass will rotate more quickly than those closer to the center of the mass. We need to consider the point's location when we are calculating its linear velocity.

Consider the point from Figure 10. As the square rotates, the point will travel a circular path of length $2\pi r$ (the length of a circle's circumference), if $r$ is the distance from the square's center of mass. The path might look something like Figure 11.

For the sake of example, let's say the angular velocity of the square in Figure 11 is $2\pi$ rad/s, or 1 rotation per second. In addition, let's say that the length of the radius $r$ is 1 meter. The circumference of the path traveled by the point $\mathbf{p}$ has a length of $2\pi r$ or $2\pi$ meters. Therefore, the point has to travel a distance of $2\pi$ meters every second, giving it an instantaneous speed of $2\pi$ meters per second.

In addition, this point must move tangentially to the circle, because it can't get closer or

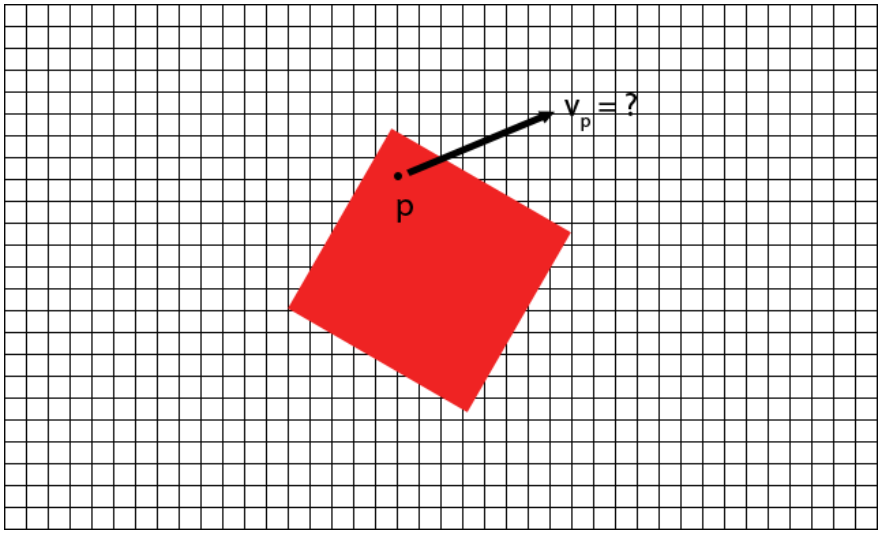


Figure 10: If the square is rotating in a clockwise direction, what is the linear velocity of a particular point **p** on that square?

farther to the center of mass (if it could, this wouldn't be a rigid body). In other words, it has to have a direction perpendicular to the radius vector. Any tangent line that touches a circle is perpendicular to the radius of the circle that extends that point.[5] We can therefore define the instantaneous linear velocity of a point on a rotating body with Equation 10.

$$\mathbf{v_p} = \omega \mathbf{r}_\perp \tag{10}$$

In addition, if the object in question is translating as well as rotating, we simply add the velocity of the center of mass ($\mathbf{v_b}$). Therefore, the complete equation for the velocity of a point on a rotating body is given in Equation 11.

$$\mathbf{v_p} = \omega \mathbf{r}_\perp + \mathbf{v_b} \tag{11}$$

Earlier I had mentioned that the velocities of a stationary-but-rotating rigid body cancel each other out, giving a velocity of 0 for the center of mass of that body. Now that we know how to calculate the velocity of arbitrary points on a rotating body, we can prove this. Let's examine the "L" Tetris piece, and treat each individual square of the piece as a point mass.

If the upper-left block (the corner of the "L") were at the origin, the four locations of the point masses would be (0, 0), (0, -1), (1, 0) and (2, 0). Using Equation 3, we find that the center of mass is located at (3/4, -1/4). To make things easy, we'll say the angular velocity is 1 rad/s and the linear velocity is 0 m/s. The vectors from the center of mass to each of the

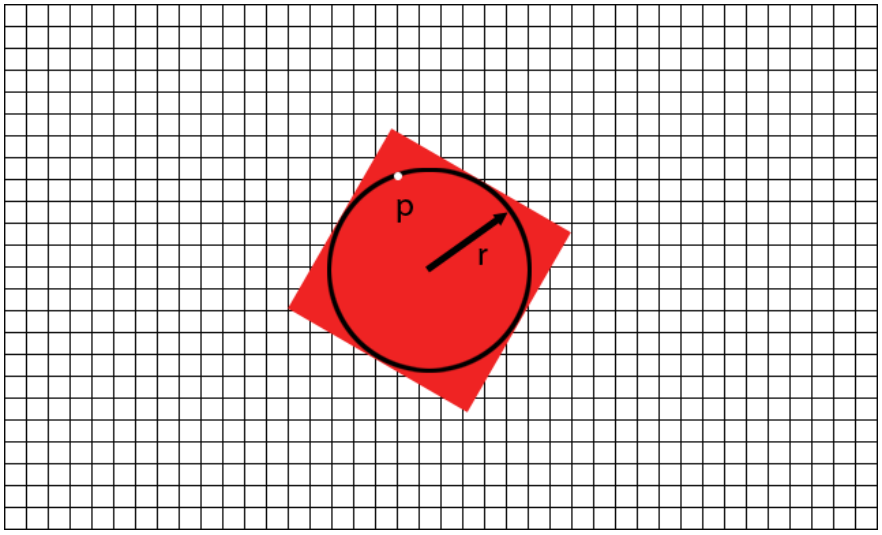


Figure 11: The circular path traced by a point on a rotating square.

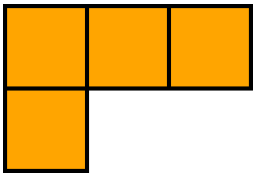

Figure 12: L block - the perfect first piece.

point masses are, respectively, (-3/4, 1/4), (-3/4, -3/4), (1/4, 1/4) and (1 1/4, 1/4). These vectors are shown in Figure 13.

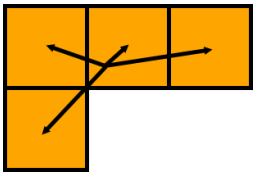

Figure 13: The radii $r_i$, used to calculate the linear velocity of a point on the rotating body.

By using Equation 11 we can determine the linear velocity of each point. The velocities are (-1/4, -3/4), (3/4, -3/4), (-1/4, 1/4) and (-1/4, 1 1/4). If we sum the velocities of each point mass, we end up with a total velocity of (0, 0), which proves our earlier point.

#### 4.3.3 Torque

Torque is the "tendency of a force to rotate an object about its axis."[6] The greater a torque induced by an applied force, the more an object will spin. At the risk of getting ahead of ourselves, we can think of torque as the angular analogue to force. There are two main components that define the magnitude of torque: the applied force and the *location* of the applied force. Mathematically, we define torque using the cross product:

$$\tau = \mathbf{r} \times \mathbf{F} \tag{12}$$

In the above equation, the vector $\mathbf{F}$ is simply the applied force vector, while $\mathbf{r}$ denotes a vector from the object's center of mass to the location of the applied force. Graphically, when an object is subject to an applied force, $\mathbf{r}$ and $\mathbf{F}$ are shown in Figure 14.

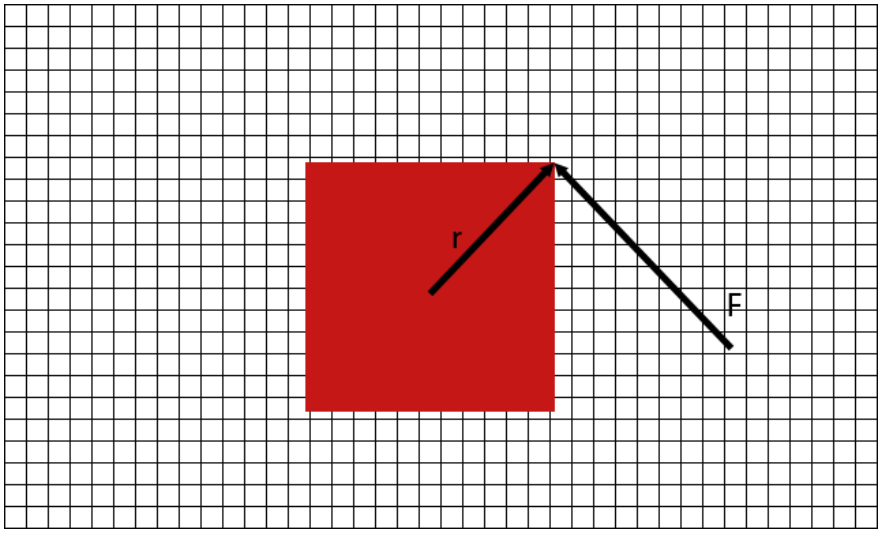


Figure 14: Remember that $\mathbf{F}$ is the applied force, while $\mathbf{r}$ is a vector from the square's center of mass to the location of the applied force.

In two dimensions, torque can be represented efficiently by a scalar. Technically, it is a 3D vector that points straight into or out of the screen. Taking the cross product of two vectors limited to a single plane will result in this. However, the resultant torque only has a component in the z dimension (the direction into or out of the screen), so its magnitude is simply the z component as the x and y components are both 0. Therefore we can store torque as a scalar when working in two dimensions.

Equation 12 makes intuitive sense. As we increase the magnitude of the applied force, the greater the torque of that force. If we hit the corner of a box with a large force, it will spin more than if we hit it with a smaller force. In addition, as we increase the magnitude of $\mathbf{r}$, or we increase the *distance* from the center of mass to the location of the applied force, the more the box will rotate. The same concept applies to a basic lever. As we apply a force at distances farther and farther away from the fulcrum of a lever, the resultant force at the same location on the other side increases in magnitude. Finally, the cross product between two vectors increases as the angle between them gets closer and closer to 90 degrees. According to Equation 12, a force that makes a perpendicular angle with the location vector $\mathbf{r}$ will induce the most torque. Again, this makes intuitive sense, as it is easier to rotate an object by pushing on the end of it sideways instead of pushing straight towards the center of the object.

#### 4.3.4 Moment of inertia

In the same way we have the linear concept of inertial mass (resistance to a change in motion given an applied force), there is the angular equivalent: the moment of inertia. The moment of inertia is defined so that it simplifies the angular equations of reaction.[7] Qualitatively, it is a mathematical description of the distribution of matter around a body's center of mass. This tells us how easy (or hard) it is to rotate the body.

If we have two objects (call them A and B) of equal mass, but the distribution of the mass in object A is more spread out than the distribution in object B, A will be harder to rotate.[8] This is because the outlying point masses in A must rotate farther than the outlying point masses in B to achieve the same rotational speed.

We're going to use kinetic energy to derive an expression for the moment of inertia. Kinetic energy is the "energy an object posseses due to its motion."[9] The kinetic energy an object has is defined by Equation 13. Note that we take the magnitude of the velocity in the following equation. Kinetic energy is a scalar quantity.

$$KE = \frac{1}{2}m|\mathbf{v}|^2 \tag{13}$$

If we define a body $b$ as a collection of point masses, and our body is rotating with an

angular velocity $\omega$, the total kinetic energy of our body would be defined as:

$$KE_b = \sum_i^N \frac{1}{2} m_i |\mathbf{v_i}|^2$$

We already determined how to find the linear velocity of a point on a rotating body (see Equation 10 for reference). Therefore, we can substitute Equation 10 into the above equation to get:

$$KE_b = \sum_i^N \frac{1}{2} m_i (\omega \, |\mathbf{r_i}|)^2$$

Because the angular velocity is a constant for the object, we take it out of the summation. We therefore arrive at:

$$KE_b = \frac{\omega^2}{2} \sum_i^N m_i \, |\mathbf{r_i}|^2$$

Because the summation in the above equation shows up a lot when dealing with rotational kinematics, we give it a special name, the moment of inertia. Formally, we give it the symbol $I$ and define it like so:

$$I = \sum_i^N m_i \, |\mathbf{r_i}|^2 \tag{14}$$

Remember that the moment of inertia is the rotational analogue to mass. This can be seen when we compare the kinetic energy of a translating (but not rotating) object with the kinetic energy of a rotating (but not translating) object:

$$\begin{aligned} KE &= \frac{1}{2} m \mathbf{v}^2 \\ KE &= \frac{1}{2} I \omega^2 \end{aligned}$$

| ***Alternative definition of the center of mass*** |
| --- |
| Because the moment of inertia is directly proportional to the distances between some reference location and all the point masses in a body, minimizing the moment of inertia means minimizing those distances. The center of mass of an object is located at the point where the object's moment of inertia (as measured from said point) is minimal. |

#### 4.3.5 Angular momentum

Although you may think that angular momentum is simply the angular equivalent of linear momentum, it is different because it includes the added complexity of a reference point. This is similar to the moment of inertia, which is also non-absolute and must include information about a reference point. Angular momentum can be observed in the physical world when a figure skater spins more quickly as she pulls her arms towards her body (her moment of inertia decreases, which causes her angular velocity to increase).[8]

We can find the angular momentum (represented by the symbol $L$) of a system by summing the momentums of all the point masses in that system around a chosen point.[7] Mathematically, this process, which uses the perp dot product, is given in Equation 15.

$$L = \sum_{i}^{N} \mathbf{r_{i\perp}} \cdot \mathbf{p_i} \tag{15}$$

We can substitute $m\mathbf{v}$ into Equation 15 since $\mathbf{p} = m\mathbf{v}$:

$$L = \sum_{i}^{N} \mathbf{r_{i\perp}} \cdot m\mathbf{v_i} \tag{16}$$

Recall from Equation 10 that $\mathbf{v_i} = \omega \mathbf{r}_\perp$. Substituting this into Equation 16 gives us Equation 17.

$$L = \omega \sum_{i}^{N} m\mathbf{r_{i\perp}} \cdot \mathbf{r_{i\perp}} \tag{17}$$

A property of the dot product operation is that if we take the dot product of a vector with itself, we end up with a value equal to the magnitude of the vector squared. Exploiting this fact gives us Equation 18.

$$L = \omega \sum_{i}^{N} m \left|\mathbf{r_i}\right|^2 \tag{18}$$

The value inside the summation looks familiar - in fact, it is the moment of inertia! Checking Equation 14 confirms this. Now you can see why introducing the moment of inertia simplifies equations. We can calculate it once at the beginning of our simulation, and use it in all subsequent operations involving angular quantities. Formally, the definition for angular momentum is given in Equation 19.

$$L = I\omega \tag{19}$$

You may notice that this is very similar to the linear definition of momentum, $\mathbf{p} = m\mathbf{v}$. Angular momentum is the angular analogue to linear momentum.

#### 4.3.6 Derivation of the angular reaction

We can now derive the angular reaction of a rigid body due to an external force. What we're going to derive is similar to the definition of the linear reaction due to force, which is simply $\mathbf{F} = m\mathbf{a}$.

Remember that linear force is the derivative of linear momentum:

$$\begin{aligned} \mathbf{p} &= m\mathbf{v} \\ \mathbf{F} &= m\mathbf{a} \end{aligned}$$

Although linear force equals the rate of change of linear momentum, what is the rate of change of angular momentum? Remember that torque can be thought of as the angular analogue to linear force. Indeed, torque is simply the rate of change of angular momentum:

$$\begin{aligned} L &= I\omega \\ \tau &= I\alpha \end{aligned}$$

The preceding equation is important. We now have a way to find the angular acceleration of an object due to an outside force:

$$\alpha = \frac{\tau}{I} \tag{20}$$

Because we need to actually find the torque at each step (it is dependent on not only force direction and magnitude, but also location), we will be doing calculations more comparable to the formula given in Equation 21.

$$\alpha = \frac{\sum(\mathbf{r_i} \times \mathbf{F_i})}{I} \tag{21}$$

### 4.4 Total reaction

We have learned that an applied force will induce both a linear and an angular reaction on a rigid body. To sum up the last results of the last three modules, we now have a pseudo-algorithm to determine the new position, velocity, acceleration, orientation, angular

velocity and angular acceleration after a force acts upon it. We can find the total reaction by following these steps:

1. Sum all forces acting on body to obtain the total force vector.

$$\mathbf{F} = \sum_{i}^{N} \mathbf{F_i}$$

2. Apply the total force vector to the body's center of mass and update the body's linear acceleration accordingly.

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

3. Find the total torque on the body by summing the cross products between all applied forces and their locations.

$$\tau = \sum_{i}^{N} (\mathbf{r_i} \times \mathbf{F_i})$$

4. Determine the new angular acceleration based on the total torque.

$$\alpha = \frac{I}{\tau}$$

5. Using the new linear and angular accelerations, integrate to find the new linear and angular velocities, as well as the new position and orientation.

## 5 Putting it all together in Python

In the previous Module, we laid the foundation for an organized simulator. If you need a refresher on the general flow we use to start and update a particular simulation, take a look at Module 2. The good news is that we don't have to significantly alter the structure of our simulator - to add rigid bodies, we simply have to extend our previous model.

A significant amount of error checking was also added. Previously, we simply checked to make sure we were receiving the right objects at the right times and ignoring errors. This is not only bad coding, but could introduce subtle errors that are hard to debug. For instance, if we do not receive a vector when we expect one, and instead use the default `Vector` value (which is $\langle 0, 0 \rangle$, in case you forgot), it would be really hard to figure out where we went wrong. Therefore, some strict error checking is used, which kills the program if we don't get what we expect.

## 5.1 Implementing rigid bodies

Recall from Module 2 that we already added point masses to our physics simulator with the `Body` class. From the theory that we derived earlier in this chapter, we know that a rigid body is a natural extension of a point mass, so it makes sense to simply subclass `Body`.

For reference, here is the `Body` class from the last Module. Note that there is some extra material in here, including a body ID which can be used to identify particular objects, and some error checking.

```
class Body(object):
    id = 0
    def __init__(self, mass=0.0, position=Vector(), velocity=Vector(),
            acceleration=Vector()):
        self.id = Body.id
        Body.id = Body.id+1

        self.mass = mass

        if isNumber(mass) and mass > 0:
            self.inverseMass = 1.0/mass
        elif isNumber(mass) and mass == 0.0:
            self.inverseMass = None
            condition = "Object " + str(self.id) +
                            " instantiated with mass of 0."
            rbpInstabilityWarning(condition)

        elif mass == float('inf'):
            self.inverseMass = 0
        else:
            self.inverseMass = None

        if isVector(position):
            self.pos = position
        else:
            rbpTypeError("Vector", type(position), self.id)

        if isVector(velocity):
            self.vel = velocity
        else:
            rbpTypeError("Vector", type(velocity), self.id)

        if isVector(acceleration):
```

```
            self.acc = acceleration
        else:
            rbpTypeError("Vector", type(acceleration), self.id)

        # A list of all forces being applied to this body in the current time
        # step. Note that this list is erased after all forces have been
        # applied.
        self.forces=[]

        # Whether or not we should physically update this body.
        self.active = True

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

    # Add a force to this object for the current time step
    def addForce(self, force):
        if isVector(force):
            self.forces.append(force.scale(self.inverseMass))
        else:
            rbpTypeError("Vector", type(force), self.id)

    # Apply all acting forces and update the body's acceleration, velocity and
    # position accordingly.
    def update(self):
        if self.active:
            for force in self.forces:
                self.acc += force

            self.vel += self.acc
            self.pos += self.vel

            self.acc.zero()

            self.forces = []
```

In addition to the linear components of a regular point mass, a rigid body has an angular orientation $\Omega$, velocity $\omega$, and acceleration $\alpha$. A rigid body exhibits a linear response due to force and an angular response due to torque. Finally, and most importantly, we need to define the rigid body itself. One way we can do this is by keeping a vertex list, or a list of

vectors that correspond to the position (in body coordinates) of each vertex. We need to keep track of all these values in our new `RigidBody` class.

```
class RigidBody(Body):
    def __init__(self, mass=0.0, vertices=[], position=Vector(),
            velocity=Vector(), acceleration=Vector(), orientation=0,
            angularVelocity=0, angularAcceleration=0):

        # A rigid body has all the properties of a normal body
        Body.__init__(self, mass, position, velocity, acceleration)

        self.verts = []     # The body's vertices when orientation=0

        self.ang_ori = orientation
        self.ang_vel = angularVelocity
        self.ang_acc = angularAcceleration

        self.torques = []   # A list of all torques currently acting on the
                            # body. Torque is the angular analogue of force
```

Remember that when we apply a force to a point mass, it only exhibits a linear response. In other words, only its position, velocity or acceleration is modified. However, in addition to a linear response, a rigid body undergoes some angular modification. Remember that the torque induced by a force $\mathbf{F}$ applied to a point $\mathbf{r}$ on a body, first shown in Equation 12, is:

$$\tau = \mathbf{r} \times \mathbf{F} \tag{22}$$

In addition, the angular response of an object to a set of applied forces can be calculated using Euler's method and by noting that angular acceleration is related to induced torque with the following relationship:

$$\alpha = \frac{\sum_i \tau_i}{I} \tag{23}$$

Therefore, we will have to modify the original body's `addForce` and `update` methods. When we add a force to a rigid body, the force will induce a torque on the object (unless the force is applied at the center of mass). See if you can fill in the appropriate code in the following stub. If you're having trouble, check out the full code in the Appendix.

```
    def addForce(self, force, location=None):
        # Add force to center of mass
```

```
        if isVector(force):
            Body.addForce(self, force)

            # Append the calculated torque to the torque list here

    def update(self):
        if self.active:
            # Update the angular acceleration, velocity and orientation based on
            # the torques in the torque list here

            self.torques = []

            # Update linear position, velocity and acceleration
            Body.update(self)
```

And that's pretty much it! We now have a way to store, apply forces to, and update rigid bodies. Let's make a nice little simulation using our new simulator!

### 5.2 The Rubber Band Box Simulation

Now that we have added rigid bodies to our simulator, let's do something with them! Currently, we don't have any way to recognize or handle colliding bodies, so we need some other way to show off our rigid body dynamics.

One of the first resources I used when learning about rigid body dynamics was the wonderful website myphysicslab.com. The link listed in the suggested reading section has a rigid body simulation where rectangular bodies are pulled by rubber bands formed between the body and the position of a mouse click. This is a fantastic simulation to replicate with our simulator. In case you want to know what the final product looks like, it's shown in Figure 15.

There are a few things we still have to add:

- A way to represent a "box" rigid body
- Drawing code to display rigid bodies
- A spring force

Each of these are covered in the following sections.

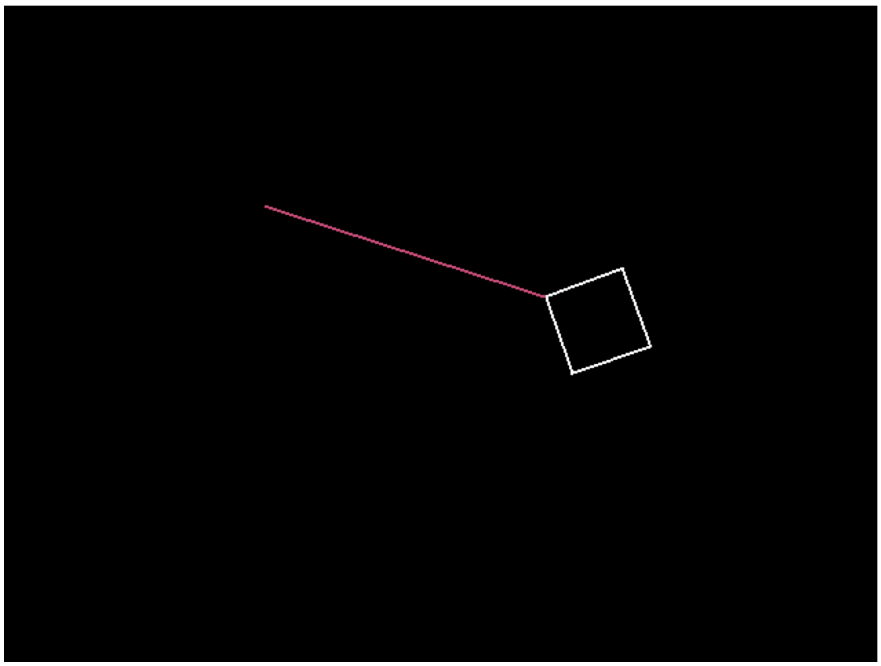

Figure 15: The Rubber Band Box Simulation in action.

#### 5.2.1 Box class

The `Box` class is going to be a simple extension of the `RigidBody` class. We're not going to limit our `Box` class to just squares. Therefore, we instantiate a box with a vector specifying the top-left corner of the box (give in body coordinates) and a height and a width. We also give it a mass and a way to define its initial position in the constructor.

The way we extended our `Box` class makes it easy to update it every step in the simulator. We simply have to call the superclass's `update` method at every step and the generic rigid body dynamics will take care of all the torques and forces on the object. To see the full definition of the `Box` class, refer to the file `rbpBodies.py` or the Appendix.

#### 5.2.2 Rigid body drawing code

With every simulation we create, we need to keep the simulator and the code used to display the numbers generated by the simulator separate. The simulator (i.e., the `World` class), doesn't contain any drawing code, and it shouldn't! As we will see later, we might want to represent complex objects with very simple rigid bodies to cut down on computation time. Therefore, drawing should be an auxiliary layer separate from the number crunching going on in the simulator. We first saw this in the Fireworks simulation, where we extended a simple

`Body` into a firework particle. We handled drawing the firework particles in the `Firework` class, *not* the body class.

With this in mind, we should extend the `Box` class for the box used in the Rubber Band Box simulation. We call this new class `SimpleBox`, and it can be found in the file `simpleBodies.py`. The drawing code in the `SimpleBox` class is very simple. We simply draw a line between each vertex of the box, accounting for the box's rotation and location in the world.

#### 5.2.3 A spring force

For our spring force, we add a simple massless spring to our `forces.py` file. Our spring force is based on Hooke's law, which basically states that for a spring, the force applied to objects attached to either end of the spring increases with the spring's length, or "strain is directly proportional to stress"[10]. Mathematically, Hooke's law is:

$$\mathbf{F} = -k\mathbf{x} \tag{24}$$

The spring constant $k$ is a measure of how strong the spring is. The larger $k$, the greater the pull exerted by the spring. Note that $\mathbf{x}$ is given in body coordinates relative to the spring's center. The reason the force is negative is that an object is *pulled* towards the center of the spring. Think of a situation where the center of the spring is at $\langle 0, 0\rangle$ (remember that this system is defined in relation to the body coordinates of the spring), and an object is located at position $\langle 10, 10\rangle$. The force exerted on the object is $\langle -10k, -10k\rangle$, which will move the object closer to the spring's origin.

Therefore, we need a way to attach a spring to two objects, and to exert a spring force on those objects at every step based on the length of the spring. And perhaps we may only want to attach one end of the spring to an object, leaving the other one statically attached to the world. We need this for our Rubber Band Box simulation, as one end of the spring will be "attached" to the mouse location.

Taking all of this into account, we can add a `Spring` force class to our `rbpForces.py` file. Note that a fully commented version is available in the `src/` directory and in the Appendix.

```
class Spring(Force):
    def __init__(self, k, attachedToBody1, attachedToBody2, loc1, loc2,
                 body1=None, body2=None):
        Force.__init__(self)
```

```
        self.loc1 = loc1
        self.loc2 = loc2

        self.attachedToBody1 = attachedToBody1
        self.attachedToBody2 = attachedToBody2
        self.body1 = body1
        self.body2 = body2

        self.k = k

    def applyForce(self):
        # Get the transformed position of end1 if it is attached to a rigid body
        if self.attachedToBody1 and isinstance(self.body1, RigidBody):
            loc1 = self.loc1.rotateCopy(self.body1.ang_ori)+self.body1.pos
        # If it's not a rigid body, just attach it to the center of the generic body
        elif self.attachedToBody1:
            loc1 = self.body1.pos
        # Otherwise, this end is hooked to the world
        else:
            loc1 = self.loc1

        # Do the same thing for end2
        if self.attachedToBody2 and isinstance(self.body2, RigidBody):
            loc2 = self.loc2.rotateCopy(self.body2.ang_ori)+self.body2.pos
        elif self.attachedToBody2:
            loc2 = self.body2.pos
        else:
            loc2 = self.loc2

        # This is the force vector pointing from body1 to body2, i.e. the one
        # pulling on body1
        force1 = (loc2 - loc1)*self.k
        # This is the force vector pointing from body2 to body1, i.e. the one
        # pulling on body2
        force2 = (loc1 - loc2)*self.k

        if self.attachedToBody1 and isinstance(self.body1, RigidBody):
            self.body1.addForce(force1, loc1-self.body1.pos)
        elif self.attachedToBody1:
            self.body1.addForce(force1)

        if self.attachedToBody2 and isinstance(self.body2, RigidBody):
```

```
            self.body2.addForce(force2, loc2-self.body2.pos)
        elif self.attachedToBody2:
            self.body2.addForce(force2)
```

We use the `Spring` force class within a class called `RubberBand` that adds a spring force between the mouse position and a corner of a `SimpleBox`.

#### 5.2.4 Combining everything

Now that we have our basic building blocks (a box and a spring and a way to draw everything), we need to combine everything into an interactive simulation. Like the simulation from myphysicslab.com mentioned earlier, the user will click somewhere on the screen. A spring will be created between the mouse click and the closest corner of the box. Just like a real spring, the spring (or rubber band) in the simluation will pull on the corner with a varying force depending on the distance between the mouse pointer and the box.

The interaction code that captures mouse clicks, etc., is not relevant to the physics material being covered here (see the Appendix for the full simulation code). However, I'd like to go over the `SimpleBox` subclass as it provides a good blueprint for how to use this physics engine in an actual application.

As mentioned earlier, the classes in `rbpBodies.py` do not have any drawing code. Therefore, we need to subclass them if we want them to actually appear on-screen. The `SimpleBox` class used in the Rubber Band Simulation is a good example of this.

```
class SimpleBox(Box):
    def __init__(self, screen, world):
        # Create a Box entity with the following properties:
        #   mass: 5
        #   height: 60
        #   width: 60
        #   position: <320, 240> (i.e. the center of the world)
        Box.__init__(self, 5, Vector(-30, -30), 60, 60, Vector(640/2, 480/2))

        self.screen = screen
        self.world = world

    def update(self):
        Box.update(self)
```

```
    def draw(self):
        # Draw a white box with lines of width 2 on the simulation screen. The
        # corners should be drawn in world coordinates, rotated by the box's
        # orientation.
        pygame.draw.polygon(self.screen, [255, 255, 255],
                            self.rotVertWorldComponents(), 2)
```

To draw an object to the screen, we add a `draw` method and specify what needs to be drawn to the screen at every step. The actual drawing is already handled by the engine. We could theoretically draw anything here, but it should be obvious that we should draw an accurate representation of the object that the draw method is attached to.

One final behavior that we add is some linear and angular damping. This is mainly to provide aesthetic benefit and to prevent the box from shooting off-screen too easily.

So that's everything for now! Remember, to start the simulation run `python rbp.py` from the directory that contains this file. The code is already organized correctly in the included distribution, so try running it before you make any modifications.

## 6 Final Comments

That's it for this lesson. Here's a quick summary of what was covered in this module:

1. Unlike point masses, rigid bodies take up space. Therefore, we need several more quantities to define a rigid body's state, including its **vertices**, **orientation**, **angular velocity** and **angular acceleration**.

2. A force applied to a rigid body causes that body to undergo both a **linear** and **angular** response. The linear response of a rigid body is the same as if the force acted on the body's **center of mass**, but the angular response is proprotional to the force's location and direction. In other words, a force exerts **torque** on a rigid body. We sum all torques and apply the total torque on an body each step. Torque is defined as:

$$\tau = \mathbf{r} \times \mathbf{F} \tag{25}$$

   The angular response due to torque is:

$$\alpha = \frac{\sum(\mathbf{r_i} \times \mathbf{F_i})}{I} \tag{26}$$

3. Physics code and drawing code should be kept separately. One way to do this is to subclass a physical object, such as a `Box`, and put the drawing code in this subclass.

# 7 Keyword definitions

***orientation***

The description of how an object is aligned in the space it occupies.[2] Like displacement, orientation is a measure of how much an object has deviated from some initial frame of reference. In our engine, the orientation of a rigid body describes how much it has rotated about its center of mass.

***center of mass***

The mean location of all the mass in a body.[3] ADD MORE TO THIS

# 8 Appendix

Because the amount of code we are writing is growing large, there is not enough space in the text to go over every class and function. Therefore, all of the code is reproduced here with documentation in the code comments.

## 8.1 Physics Engine

rbpMath.py

```
import math

class Vector(object):
    # Initializes 2d vector.  Default is the 0 vector.
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = 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. Overloads the ==
    # operator.
    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. Overloads the != operator.
    def __ne__(self, other):
        eq = self.__eq__(other)
        if eq is NotImplemented:
            return not eq
        return NotImplemented

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

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

# Vector subtraction and assignment. Overloads the -= operator.
def __isub__(self, other):
    if isinstance(other, Vector):
        self.x -= other.x
        self.y -= other.y
        return self
    else:
        return NotImplemented

# Dot product. Overloads the * operator.
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. Overloads the *= operator.
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 the cross product can be represented by a scalar.
    def __mod__(self, other):
        if isinstance(other, Vector):
            return (self.x*other.y - self.y*other.x)
        else:
            return NotImplemented

    def zero(self):
        self.x = 0
        self.y = 0

    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)

    def angle(self):
        return math.atan2(self.y, self.x)

    # Rotates the vector through an angle of "angle" radians.
    # Based on the following matrix rotation:
    # |x'| = |cos(angle) -sin(angle)||x|
    # |y'| = |sin(angle)  cos(angle)||y|
    def rotate(self, angle):
        self.x = math.cos(angle)*self.x - math.sin(angle)*self.y
        self.y = math.sin(angle)*self.x + math.cos(angle)*self.y

    def rotateCopy(self, angle):
        x = math.cos(angle)*self.x - math.sin(angle)*self.y
        y = math.sin(angle)*self.x + math.cos(angle)*self.y
        return Vector(x, y)

def isNumber(obj):
    return isinstance(obj, int) or isinstance(obj, float)

def isVector(obj):
    return isinstance(obj, Vector)

def euclideanDistance(v1, v2):
    return math.sqrt((v1.x-v2.x)**2 + (v1.y-v2.y)**2)
```

rbpForces.py

```
from rbpMath import *
from rbpBodies import *
import sys

# The basic superclass for all forces.
class Force(object):
    id = 0
    def __init__(self):
        # This is a unique ID that is given to every Force. It is used to keep
        # track of specific forces in the World object.
        self.id = Force.id
        Force.id = Force.id + 1

        self.active = True
        pass

    def applyForce(self, bodyList):
        pass

# Unary gravitational force that acts in a direction specified by forceVector.
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.itervalues():
            body.acc += self.forceVector

# Unary force that dampens an object's linear motion.
class LinearDamping(Force):
    def __init__(self, b):
        Force.__init__(self)
        self.b = b

    def applyForce(self, bodyList):
        for body in bodyList.itervalues():
            body.addForce(body.vel.scale(-self.b))
```

```
# Force that dampens an object's angular motion.
class AngularDamping(Force):
    def __init__(self, b):
        Force.__init__(self)
        self.b = b

    def applyForce(self, bodyList):
        for body in bodyList.itervalues():
            if isinstance(body, RigidBody):
                body.ang_vel = body.ang_vel*self.b

# A spring that can be attached to the world or a body in the world.
# If attachedToBody1 or 2 is true, the appropriate body must be passed to this
# function.
class Spring(Force):
    def __init__(self, k, attachedToBody1, attachedToBody2, loc1, loc2,
                 body1=None, body2=None):
        Force.__init__(self)

        # If either end is attached to a body, loc1 and loc2 should be Vectors
        # specifying the location of the spring in BODY coordinates.  Otherwise,
        # they should specify the coordinates of the spring in WORLD
        # coorindates.
        self.loc1 = loc1
        self.loc2 = loc2

        self.attachedToBody1 = attachedToBody1
        self.attachedToBody2 = attachedToBody2
        self.body1 = body1
        self.body2 = body2

        if attachedToBody1:
            if not isinstance(body1, Body):
                print "Error in spring force: expected a body at end 1."
                sys.exit(1)
        if attachedToBody2:
            if not isinstance(body2, Body):
                print "Error in spring force: expected a body at end 2."
                sys.exit(1)

        self.k = k

    def applyForce(self):
```

```
        # Get the transformed position of end1 if it is attached to a rigid
        # body.
        if self.attachedToBody1 and isinstance(self.body1, RigidBody):
            loc1 = self.loc1.rotateCopy(self.body1.ang_ori)+self.body1.pos
        # If it's not a rigid body, just attach it to the center of the generic
        # body.
        elif self.attachedToBody1:
            loc1 = self.body1.pos
        # Otherwise, this end is hooked to the world
        else:
            loc1 = self.loc1

        # Do the same thing for end2
        if self.attachedToBody2 and isinstance(self.body2, RigidBody):
            loc2 = self.loc2.rotateCopy(self.body2.ang_ori)+self.body2.pos
        elif self.attachedToBody2:
            loc2 = self.body2.pos
        else:
            loc2 = self.loc2

        # This is the force vector pointing from body1 to body2, i.e. the one
        # pulling on body1
        force1 = (loc2 - loc1)*self.k
        # This is the force vector pointing from body2 to body1, i.e. the one
        # pulling on body2
        force2 = (loc1 - loc2)*self.k

        if self.attachedToBody1 and isinstance(self.body1, RigidBody):
            self.body1.addForce(force1, loc1-self.body1.pos)
        elif self.attachedToBody1:
            self.body1.addForce(force1)

        if self.attachedToBody2 and isinstance(self.body2, RigidBody):
            self.body2.addForce(force2, loc2-self.body2.pos)
        elif self.attachedToBody2:
            self.body2.addForce(force2)
```

rbpBodies.py

```
from rbpMath import *
from rbpErrors import *
```

```
# This is the basic superclass for all bodies.
class Body(object):
    id = 0
    def __init__(self, mass=0.0, position=Vector(), velocity=Vector(),
            acceleration=Vector()):

        # Assign a static ID for this body. This ID is unique for every instance
        # that derives from this class.
        self.id = Body.id
        Body.id = Body.id+1

        self.mass = mass

        if isNumber(mass) and mass > 0:
            self.inverseMass = 1.0/mass
        elif isNumber(mass) and mass == 0.0:
            self.inverseMass = None
            # Creating an object with zero mass is a potentially unstable
            # operation, so raise a warning.
            condition = "Object", str(self.id), "instantiated with mass of 0."
            rbpInstabilityWarning(condition)

        # The inverse mass is used in the collision response equation, and an
        # inverse mass of 0 essentially creates an unmoveable object.
        elif isNumber(mass) and mass == float('inf'):
            self.inverseMass = 0
        elif not isNumber(mass):
            rbpTypeError("Number", type(mass), self.id)
        # We got a negative or otherwise illegal mass.
        else:
            self.inverseMass = None
            condition = "Object", str(self.id),\
                    "instantiated with illegal mass of", str(self.mass)
            rbpGeneralWarning(condition)

        if isVector(position):
            self.pos = position
        else:
            rbpTypeError("Vector", type(position), self.id)

        if isVector(velocity):
            self.vel = velocity
        else:
```

```
            rbpTypeError("Vector", type(velocity), self.id)

        if isVector(acceleration):
            self.acc = acceleration
        else:
            rbpTypeError("Vector", type(acceleration), self.id)

        # A list of all forces being applied to this body in the current time
        # step. Note that this list is erased after all forces have been
        # applied.
        self.forces=[]

        # Whether or not we should physically update this body every step.
        self.active = True

        # When print is called on this class, this output is produced
    def __str__(self):
                return  "Position: " + str(self.pos) + "\n"\
                "Velocity: " + str(self.vel) + "\n"\
                "Acceleration: " + str(self.acc) + "\n"

    # Add a force to this object for the current time step.
    def addForce(self, force):
        if isVector(force):
            self.forces.append(force.scale(self.inverseMass))
        else:
            rbpTypeError("Vector", type(force), self.id)

    # Apply all acting forces and update the body's acceleration, velocity and
    # position accordingly.
    def update(self):
        if self.active:
            for force in self.forces:
                self.acc += force

            self.vel += self.acc
            self.pos += self.vel

            self.acc.zero()

            self.forces = []

# A rigid body extension of the base Body class.
```

```
# The main difference between a body and a rigid body is that a rigid body is
# two-dimensional - it occupies space and has an angular orientation (i.e. a
# certain amount of rotation).
class RigidBody(Body):
    def __init__(self, mass=0.0, vertices=[], position=Vector(),
            velocity=Vector(), acceleration=Vector(), orientation=0,
            angularVelocity=0, angularAcceleration=0):

        # A rigid body has all the properties of a normal body
        Body.__init__(self, mass, position, velocity, acceleration)

        # The body's vertices when orientation=0. The list of vertices is a list
        # of vectors corresponding to each vertex, in body coordinates.
        self.verts = []
        # The vertices rotated by the body's orientation
        self.rotVerts = []

        for vertex in vertices:
            if isVector(vertex):
                self.verts.append(vertex)
                self.rotVerts.append(vertex)

        if isNumber(orientation):
            self.ang_ori = orientation
        else:
            rbpTypeError("Number", type(orientation), self.id)

        if isNumber(angularVelocity):
            self.ang_vel = angularVelocity
        else:
            rbpTypeError("Number", type(angularVelocity), self.id)

        if isNumber(angularAcceleration):
            self.ang_acc = angularAcceleration
        else:
            rbpTypeError("Number", type(angularAcceleration), self.id)

        # A list of all torques currently acting on the body. Torque is the
        # angular analogue of force
        self.torques = []

    def __str__(self):
        return Body.__str__(self) + "\n"\
```

```
            "Orientation: " + str(self.ang_ori) + "\n"\
            "Angular velocity: " + str(self.ang_vel) + "\n"\
            "Angular acceleration: " + str(self.ang_acc) + "\n"

    def addForce(self, force, location=None):
        # Add force to center of mass. A force induces both a linear and angular
        # response on a rigid body.
        if isVector(force):
            Body.addForce(self, force)

            # Add torque to object.
            # Recall that we overloaded the "%" operator to perform the cross
            # product between two Vectors. Also recall that the cross product of
            # two vectors in a plane results in a vector orthogonal to that
            # plane. Therefore, we can represent the torque as a scalar, since
            # its direction is redundant.
            if isVector(location):
                self.torques.append((location % force)/self.momentOfInertia)

    def update(self):
        if self.active:
            if self.momentOfInertia is not None:
                totalTorque = 0
                for t in self.torques:
                    totalTorque += t

                self.ang_acc = totalTorque
                self.ang_vel += self.ang_acc
                self.ang_ori += self.ang_vel

                self.torques = []

            # Update linear position, velocity and acceleration
            Body.update(self)

            # Update vertices based on orientation
            for i in range(0, len(self.verts)):
                self.rotVerts[i] = self.verts[i].rotateCopy(self.ang_ori)

    # Returns a list of vertices in body coordinates, with no rotation
    def vertsBody(self):
        return self.verts
```

```
    # Returns a list of rotated vertices in body coordinates
    def rotVertsBody(self):
        return self.rotVerts

    # Returns a list of rotated vertices in world coordinates
    def rotVertsWorld(self):
        verts = []
        for vertex in self.rotVerts:
            verts.append(vertex+self.pos)
        return verts

    # Returns a list of vertex component tuples. Useful for Pygame, which
    # requires vertices in the form [[x1, y1], [x2, y2], ... , [xn, yn]]
    def rotVertComponents(self):
        comps = []
        for vertex in self.rotVerts:
            comps.append(vertex.components())
        return comps

    # Returns a list of rotated vertex components in world coordinates
    def rotVertWorldComponents(self):
        comps = []
        for vertex in self.rotVerts:
            comps.append((vertex+self.pos).components())
        return comps

# A specific type of RigidBody - a rectangular Box.
# Create one by specifying the top-left corner as a Vector in body coordinates,
# followed by the box's height and width.
class Box(RigidBody):
    def __init__(self, mass, corner, height, width, position=Vector()):
        if isVector(corner):
            cc = corner.components()
        else:
            rbpTypeError("Vector", type(corner), self.id)

        # Initializes vertexes in clockwise direction, starting at upper-left
        v1 = corner
        v2 = Vector(cc[0]+width, cc[1])
        v3 = Vector(cc[0]+width, cc[1]+height)
        v4 = Vector(cc[0], cc[1]+height)

        self.height = height
```

```
        self.width = width

        RigidBody.__init__(self, mass, [v1, v2, v3, v4], position)

        # This is the equation for the moment of inertia for a 2D rectangle
        self.momentOfInertia = self.mass*(self.width**2+self.height**2)/12

    def update(self):
        if self.active:
            RigidBody.update(self)
```

## 8.2 World Code and Error Catching

rbpWorld.py

```
from rbpBodies import *
from rbpMath import *
from rbpForces import *
import sys

class World(object):
    def __init__(self):
        # Instead of keeping generic object lists, we instead store every object
        # and force in a dictionary. The dictionary keys are the
        # object's/force's static ID. This makes it possible to quickly remove
        # or change a specific entity instead of having to linearly search an
        # array every time we need access to that entity.
        self.bodyList = {}
        self.unaryForces = {}
        self.forces = {}

        self.drawOffScreen = True

    def addBody(self, body):
        if isinstance(body, Body):
            self.bodyList[body.id] = body

    def removeBody(self, body):
        if isinstance(body, Body):
            # TODO: call dictionary-specific delete here
            try:
                self.bodyList.index(body)
                self.bodyList.remove(body)
            except ValueError:
                print "Could not find and delete body", body.id

    def clearBodyList(self):
        self.bodyList = {}

    def addUnaryForce(self, force):
        if isinstance(force, Force):
            self.unaryForces[force.id] = force
        else:
            rbpTypeError("Force", type(force), "World")
```

```
    def addForce(self, force):
        if isinstance(force, Force):
            self.forces[force.id] = force
        else:
            rbpTypeError("Force", type(force), "World")

    def update(self):
        for force in self.unaryForces.itervalues():
            if force.active:
                force.applyForce(self.bodyList)

        for force in self.forces.itervalues():
            if force.active:
                force.applyForce()

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

rbpErrors.py

```
import sys
import traceback

# This is an unexpected entity (type) error - i.e. getting a Number when we
# expected a Vector.
def rbpTypeError(expectedType, receivedType, id, additionalInfo=None):
    print "****************************************"
    tb = traceback.format_stack()
    # Remove the last two stack entries that refer to this function
    tb = tb[0:-2]
    for line in tb:
        print line.strip()

    print "----------------------------------------"
    print "RBP error: expected " + str(expectedType) + ", but got "\
                + str(receivedType) + ". Aborting."
    print "In object " + str(id) + "."
    if additionalInfo:
        print additionalInfo
```

```
    print "****************************************"

    # Because our code can't handle incorrect data types, we abort.
    sys.exit(1)

# This is a general warning that does not necessitate an abort.
def rbpGeneralWarning(condition, descriptiveInfo=None, additionalInfo=None):
    print "****************************************"
    tb = traceback.format_stack()
    # Remove the last two stack entries that refer to this function
    tb = tb[0:-2]
    for line in tb:
        print line.strip()

    print "----------------------------------------"
    if descriptiveInfo is not None:
        print descriptiveInfo
    print condition
    if additionalInfo is not None:
        print additionalInfo
    print "****************************************"

# This warning is called when an unstable condition, such as an object with 0
# mass, is detected.
def rbpInstabilityWarning(condition, additionalInfo=None):
    rbpGeneralWarning(condition, "RBP warning: unstable condition detected.")
```

## 8.3 Simulation Code

rbp.py

```
import pygame

import sys
sys.path.append('./sims/')
from simpleBodies import SimpleBodiesSim

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

simpleBodiesSim = SimpleBodiesSim(screen)

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

    # Pass mouse clicks to the simulation object
    if pygame.mouse.get_pressed()[0] == 1:
        simpleBodiesSim.handleMousePressed( pygame.mouse.get_pos() )
    else:
        simpleBodiesSim.handleMouseNotPressed()


    screen.fill(C_BLACK)

    simpleBodiesSim.update()

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

```
pygame.quit()
```

simpleBodies.py

```
import pygame, random, math
from rbpBodies import Body
from rbpMath import *
from rbpWorld import *
from rbpForces import *

class SimpleBodiesSim(object):
    def __init__(self, screen):
        self.world = World()
        self.screen = screen
        self.mouseHasClicked = False
        # The position of the closest box corner, in world coordinates.
        self.closestCorner = None
        self.cci = -1

        self.simpleBox = SimpleBox(self.screen, self.world)
        self.world.addBody(self.simpleBox)

        self.rubberBand = RubberBand(self.screen, self.world, 0.002,
                                     self.simpleBox)
        self.world.addBody(self.rubberBand)

        # Add some damping to the simulation to prevent the box from flying
        # off-screen too easily.
        angularDamping = AngularDamping(0.98)
        self.world.addUnaryForce(angularDamping)
        drag = LinearDamping(0.05)
        self.world.addUnaryForce(drag)

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

    def handleMousePressed(self, pos):
        mPos = Vector(float(pos[0]), float(pos[1]))

        rvw = self.simpleBox.rotVertsWorld()
        vb = self.simpleBox.vertsBody()
        # Find the closest box corner relative to the mouse position.
```

```
        if not self.mouseHasClicked:
            minDist = 99999
            self.closestCorner = None
            self.cci = -1

            for i in range(0, len(rvw)):
                d = euclideanDistance(mPos, rvw[i])
                if d < minDist:
                    self.closestCorner = vb[i]
                    self.cci = i
                    minDist = d

            self.mouseHasClicked = True
        else:
            self.closestCorner = vb[self.cci]

        # Attach the rubber band to the closest corner of the box and to the
        # world at the mouse pointer's position.
        self.rubberBand.setEnd1Pos(self.closestCorner)
        self.rubberBand.setEnd2Pos(mPos)
        self.rubberBand.setActive(True)

    def handleMouseNotPressed(self):
        self.rubberBand.setActive(False)
        self.mouseHasClicked = False

class SimpleBox(Box):
    def __init__(self, screen, world):
        Box.__init__(self, 5, Vector(-30, -30), 60, 60, Vector(640/2, 480/2))

        self.screen = screen
        self.world = world

    def update(self):
        Box.update(self)

    def draw(self):
        pygame.draw.polygon(self.screen, [255, 255, 255],
                            self.rotVertWorldComponents(), 2)

class RubberBand(Body):
    def __init__(self, screen, world, k, attachedBody):
        Body.__init__(self)
```

```
        self.force = Spring(k, True, False, Vector(), Vector(), attachedBody)
        world.addForce(self.force)
        self.active = False
        self.force.active = self.active

        self.screen = screen
        self.world = world

        self.attachedBody = attachedBody
        self.len = euclideanDistance(self.force.loc1, self.force.loc2)

    # This sets the end of the rubber band that is attached to the body.
    # Note that pos must be in the body coordinates of the attached object.
    def setEnd1Pos(self, pos):
        self.force.loc1 = pos

    # This sets the end of the rubber band that is attached to the world.
    # Note that pos must be in world coordinates.
    def setEnd2Pos(self, pos):
        self.force.loc2 = pos

    def setActive(self, active):
        if active:
            self.active = True
            self.force.active = True
        else:
            self.active = False
            self.force.active = False

    # Override adding forces to this object. The rubber band should not be
    # affected by world or other forces.
    def addForce(self, force):
        pass

    def update(self):
        self.pos = (self.force.loc1+self.force.loc2).scale(0.5)
        if self.active:
            self.len = euclideanDistance(self.force.loc1, self.force.loc2)

    def draw(self):
        if self.active:
            pygame.draw.polygon(self.screen, [255, 0, min(self.len/2, 255)],
```

```
                    [(self.force.loc1.rotateCopy(self.attachedBody.ang_ori)
                     +self.attachedBody.pos).components(),
                      self.force.loc2.components()], 2)
```

# 9 References


1. http://en.wikipedia.org/wiki/Point_particle
2. http://en.wikipedia.org/wiki/Orientation_(geometry)
3. http://en.wikipedia.org/wiki/Center_of_mass
4. http://mathworld.wolfram.com/PerpendicularVector.html
5. http://en.wikipedia.org/wiki/Circle#Tangent_lines
6. http://en.wikipedia.org/wiki/Torque
7. http://chrishecker.com/images/c/c2/Gdmphys2.pdf
8. http://en.wikipedia.org/wiki/Moment_of_inertia
9. http://en.wikipedia.org/wiki/Kinetic_energy
10. http://en.wikipedia.org/wiki/Hooke's_law