An Introduction to Physically Based Modeling:
Rigid Body Simulation I—Unconstrained Rigid Body Dynamics

David Baraff
Robotics Institute
Carnegie Mellon University

Rigid Body Simulation

David Baraff
Robotics Institute
Carnegie Mellon University

Introduction

This portion of the course notes deals with the problem of rigid body dynamics. To help get you started simulating rigid body motion, we've provided code fragments that implement most of the concepts discussed in these notes. This segment of the course notes is divided into two parts. The first part covers the motion of rigid bodies that are completely *unconstrained* in their allowable motion; that is, simulations that aren't concerned about collisions between rigid bodies. Given any external forces acting on a rigid body, we'll show how to simulate the motion of the body in response to these forces. The mathematical derivations in these notes are meant to be fairly informal and intuitive.

The second part of the notes tackles the problem of *constrained* motion that arises when we regard bodies as solid, and need to disallow inter-penetration. We enforce these non-penetration constraints by computing appropriate contact forces between contacting bodies. Given values for these contact forces, simulation proceeds exactly as in the unconstrained case: we simply apply all the forces to the bodies and let the simulation unfold as though the motions of bodies are completely unconstrained. If we have computed the contact forces correctly, the resulting motion of the bodies will be free from inter-penetration. The computation of these contact forces is the most demanding component of the entire simulation process.[1]

[1]Collision detection (i.e. determining the points of contact between bodies) runs a close second though!

Part I. Unconstrained Rigid Body Dynamics

1 Simulation Basics

This portion of the course notes is geared towards a full implementation of rigid body motion. In this section, we'll show the basic structure for simulating the motion of a rigid body. In section 2, we'll define the terms, concepts, and equations we need to implement a rigid body simulator. Following this, we'll give some code to actually implement the equations we need. Derivations for some of the concepts and equations we will be using will be left to appendix A.

The only thing you need to be familiar with at this point are the basic concepts (but not the numerical details) of solving ordinary differential equations. If you're not familiar with this topic, you're in luck: just turn back to the beginning of these course notes, and read the section on "Differential Equation Basics." You also might want to read the next section on "Particle Dynamics" as well, although we're about to repeat some of that material here anyway.

Simulating the motion of a rigid body is almost the same as simulating the motion of a particle, so let's start with particle simulation. The way we simulate a particle is as follows. We let a function $x(t)$ denote the particle's location in world space (the space all particles or bodies occupy during simulation) at time t. The function $v(t) = \dot{x}(t) = \frac{d}{dt}x(t)$ gives the velocity of the particle at time t. The *state* of a particle at time t is the particle's position and velocity. We generalize this concept by defining a *state vector* $\mathbf{Y}(t)$ for a system: for a single particle,

$$\mathbf{Y}(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}. \tag{1–1}$$

When we're talking about an actual implementation, we have to "flatten" out $\mathbf{Y}(t)$ into an array. For a single particle, $\mathbf{Y}(t)$ can be described as an array of six numbers: typically, we'd let the first three elements of the array represent $x(t)$, and the last three elements represent $v(t)$. Later, when we talk about state vectors $\mathbf{Y}(t)$ that contain matrices as well as vectors, the same sort of operation is done to flatten $\mathbf{Y}(t)$ into an array. Of course, we'll also have to reverse this process and turn an array of numbers back into a state vector $\mathbf{Y}(t)$. This all comes down to pretty simple bookkeeping though, so henceforth, we'll assume that we know how to convert any sort of state vector $\mathbf{Y}(t)$ to an array (of the appropriate length) and vice versa. (For a simple example involving particles, look through the "Particle System Dynamics" section of these notes.)

For a system with n particles, we enlarge $\mathbf{Y}(t)$ to be

$$\mathbf{Y}(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ \vdots \\ x_n(t) \\ v_n(t) \end{pmatrix} \tag{1–2}$$

where $x_i(t)$ and $v_i(t)$ are the position and velocity of the ith particle. Working with n particles is no harder than working with one particle, so we'll let $\mathbf{Y}(t)$ be the state vector for a single particle for now (and when we get to it later, a single rigid body).

To actually simulate the motion of our particle, we need to know one more thing—the force acting on the particle at time t. We'll define $F(t)$ as the force acting on our particle at time t. The function $F(t)$ is the sum of all the forces acting on the particle: gravity, wind, spring forces, etc. If the particle has mass m, then the change of $\mathbf{Y}$ over time is given by

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ F(t)/m \end{pmatrix}. \tag{1–3}$$

Given any value of $\mathbf{Y}(t)$, equation (1–3) describes how $\mathbf{Y}(t)$ is instantaneously changing at time t. A simulation starts with some initial conditions for $\mathbf{Y}(0)$, (i.e. values for $x(0)$ and $v(0)$) and then uses a numerical equation solver to track the change or "flow" of $\mathbf{Y}$ over time, for as long as we're interested in. If all we want to know is the particle's location one second from now, we ask the solver to compute $\mathbf{Y}(1)$, assuming that time units are in seconds. If we're going to animate the motion of the particle, we'd want to compute $\mathbf{Y}(\frac{1}{30})$, $\mathbf{Y}(\frac{2}{30})$ and so on.

The numerical method used by the solver is relatively unimportant with respect to our actual implementation. Let's look at how we'd actually interact with a numerical solver, in a C++-like language. Assume we have access to a numerical solver, which we'll generically write as a function named `ode`. Typically, `ode` has the following specification:

```
typedef void (*dydt_func)(double t, double y[], double ydot[]);

void    ode(double y0[], double yend[], int len, double t0,
            double t1, dydt_func dydt);
```

We pass an initial state vector to `ode` as an array `y0`. The solver `ode` knows nothing about the inherent structure of `y0`. Since solvers can handle problems of arbitrary dimension, we also have to pass the length `len` of `y0`. (For a system of n particles, we'd obviously have `len` $= 6n$.) We also pass the solver the starting and ending times of the simulation, `t0` and `t1`. The solver's goal is to compute the state vector at time `t1` and return it in the array `yend`.

We also pass a function `dydt` to `ode`. Given an array `y` that encodes a state vector $\mathbf{Y}(t)$ and a time t, `dydt` must compute and return $\frac{d}{dt}\mathbf{Y}(t)$ in the array `ydot`. (The reason we must pass `t` to `dydt` is that we may have time-varying forces acting in our system. In that case, `dydt` would have to know "what time it is" to determine the value of those forces.) In tracing the flow of $\mathbf{Y}(t)$ from `t0` to `t1`, the solver `ode` is allowed to call `dydt` as often as it likes. Given that we have such a routine `ode`, the only work we need to do is to code up the routine `dydt` which we'll give as a parameter to `ode`.

Simulating rigid bodies follows exactly the same mold as simulating particles. The only difference is that the state vector $\mathbf{Y}(t)$ for a rigid body holds more information, and the derivative $\frac{d}{dt}\mathbf{Y}(t)$ is a little more complicated. However, we'll use exactly the same paradigm of tracking the movement of a rigid body using a solver `ode`, which we'll supply with a function `dydt`.

2 Rigid Body Concepts

The goal of this section is to develop an analogue to equation (1–3), for rigid bodies. The final differential equation we develop is given in section 2.11. In order to do this though, we need to define

a lot of concepts first and relations first. Some of the longer derivations are found in appendix A. In the next section, we'll show how to write the function `dydt` needed by the numerical solver `ode` to compute the derivative $\frac{d}{dt}\mathbf{Y}(t)$ developed in this section.

2.1 Position and Orientation

The location of a particle in space at time t can be described as a vector $x(t)$, which describes the translation of the particle from the origin. Rigid bodies are more complicated, in that in addition to translating them, we can also rotate them. To locate a rigid body in world space, we'll use a vector $x(t)$, which describes the translation of the body. We must also describe the rotation of the body, which we'll do (for now) in terms of a 3×3 rotation matrix $R(t)$. We will call $x(t)$ and $R(t)$ the *spatial variables* of a rigid body.

A rigid body, unlike a particle, occupies a volume of space and has a particular shape. Because a rigid body can undergo only rotation and translation, we define the shape of a rigid body in terms of a fixed and unchanging space called *body space.* Given a geometric description of the body in body space, we use $x(t)$ and $R(t)$ to transform the body-space description into world space (figure 1). In order to simplify some equations we'll be using, we'll require that our description of the rigid body in body space be such that the *center of mass* of the body lies at the origin, $(0, 0, 0)$. We'll define the center of mass more precisely later, but for now, the center of mass can be thought of as a point in the rigid body that lies at the geometric center of the body. In describing the body's shape, we require that this geometric center lie at $(0, 0, 0)$ in body space. If we agree that $R(t)$ specifies a rotation of the body about the center of mass, then a fixed vector r in body space will be rotated to the world-space vector $R(t)r$ at time t. Likewise, if p_0 is an arbitrary point on the rigid body, in body space, then the world-space location $p(t)$ of p_0 is the result of first rotating p_0 about the origin and then translating it:

$$p(t) = R(t)\,p_0 + x(t). \tag{2–1}$$

Since the center of mass of the body lies at the origin, the world-space location of the center of mass is always given directly by $x(t)$. This lets us attach a very physical meaning to $x(t)$ by saying that $x(t)$ is the location of the center of mass in world space at time t. We can also attach a physical meaning to $R(t)$. Consider the x axis in body space i.e. the vector $(1, 0, 0)$. At time t, this vector has direction

$$R(t)\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

in world space. If we write out the components of $R(t)$ as

$$R(t) = \begin{pmatrix} r_{xx} & r_{yx} & r_{zx} \\ r_{xy} & r_{yy} & r_{zy} \\ r_{xz} & r_{yz} & r_{zz} \end{pmatrix}, \tag{2–2}$$

then

$$R(t)\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \tag{2–3}$$

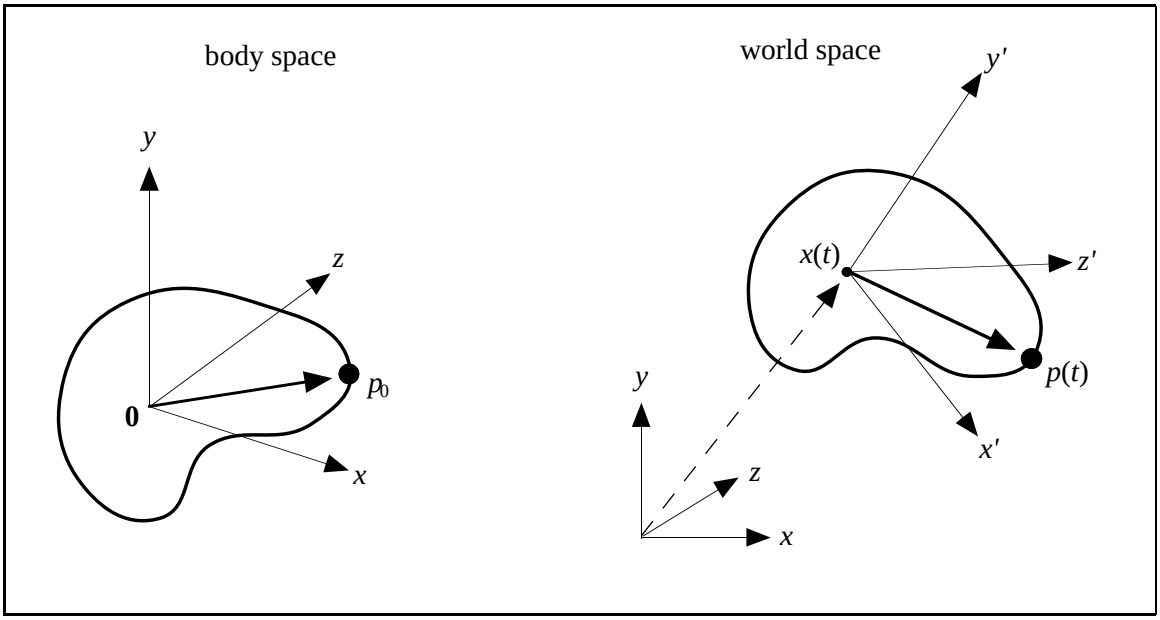

Figure 1: The center of mass is transformed to the point $x(t)$ in world space, at time t. The fixed x, y, and z axes of the body in body space transform to the vectors $\dot{x} = R(t)x$, $y' = R(t)y$ and $z' = R(t)z$. The fixed point p_0 in body space is transformed to the point $p(t) = R(t)p_0 + x(t)$.

which is the first column of $R(t)$. The physical meaning of $R(t)$ is that $R(t)$'s first column gives the direction that the rigid body's x axis points in, when transformed to world space at time t. Similarly, the second and third columns of $R(t)$,

$$\begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix}$$

are the directions of the y and z axes of the rigid body in world space at time t (figure 2).

2.2 Linear Velocity

For simplicity, we'll call $x(t)$ and $R(t)$ the *position* and *orientation* of the body at time t. The next thing we need to do is define how the position and orientation change over time. This means we need expressions for $\dot{x}(t)$ and $\dot{R}(t)$. Since $x(t)$ is the position of the center of mass in world space, $\dot{x}(t)$ is the velocity of the center of mass in world space. We'll define the *linear velocity* $v(t)$ as this velocity:

$$v(t) = \dot{x}(t). \tag{2–4}$$

If we imagine that the orientation of the body is fixed, then the only movement the body can undergo is a pure translation. The quantity $v(t)$ gives the velocity of this translation.

2.3 Angular Velocity

In addition to translating, a rigid body can also spin. Imagine however that we freeze the position of the center of mass in space. Any movement of the points of the body must therefore be due to the

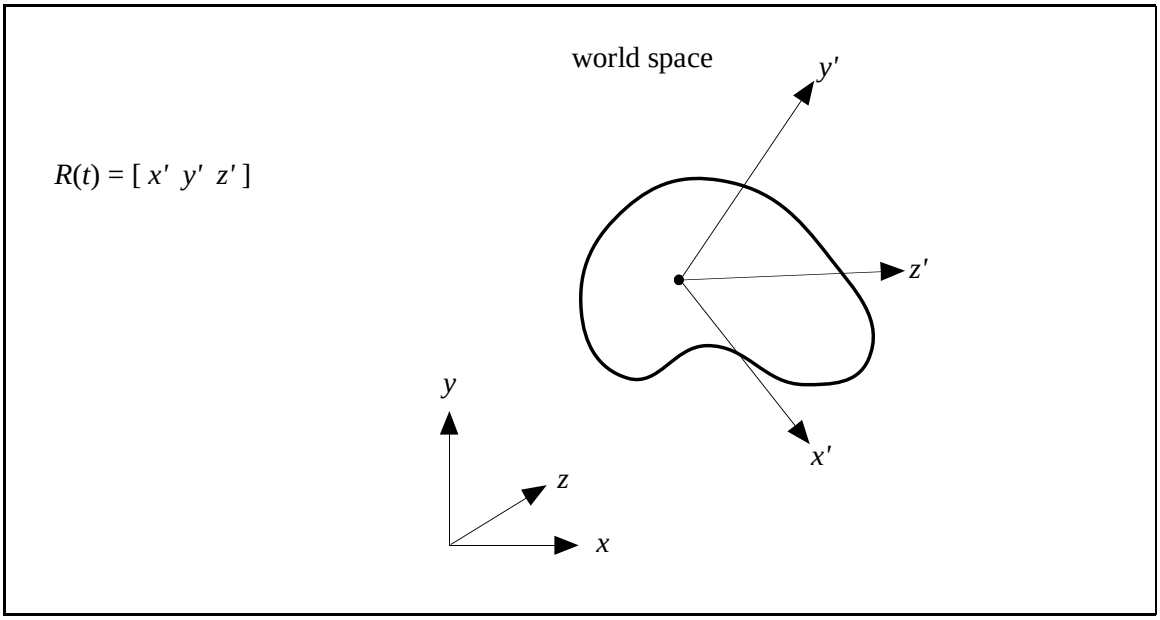

Figure 2: Physical interpretation of the orientation matrix $R(t)$. At time t, the columns of $R(t)$ are the world-space directions that the body-space x, y, and z axes transform to.

body spinning about some axis that passes through the center of mass. (Otherwise the center of mass would itself be moving). We can describe that spin as a vector $\omega(t)$. The *direction* of $\omega(t)$ gives the direction of the axis about which the body is spinning (figure 3). The magnitude of $\omega(t)$, $|\omega(t)|$, tells how fast the body is spinning. $|\omega(t)|$ has the dimensions of revolutions/time; thus, $|\omega(t)|$ relates the angle through which the body will rotate over a given period of time, if the angular velocity remains constant. The quantity $\omega(t)$ is called the *angular velocity.*

For linear velocity, $x(t)$ and $v(t)$ are related by $v(t) = \frac{d}{dt}x(t)$. How are $R(t)$ and $\omega(t)$ related? (Clearly, $\dot{R}(t)$ cannot be $\omega(t)$, since $R(t)$ is a matrix, and $\omega(t)$ is a vector.) To answer this question, let's remind ourselves of the physical meaning of $R(t)$. We know that the columns of $R(t)$ tell us the directions of the transformed x, y and z body axes at time t. That means that the columns of $\dot{R}(t)$ must describe the *velocity* with which the x, y, and z axes are being transformed. To discover the relationship between $\omega(t)$ and $R(t)$, let's examine how the change of an arbitrary vector in a rigid body is related to the angular velocity $\omega(t)$.

Figure 4 shows a rigid body with angular velocity $\omega(t)$. Consider a vector $r(t)$ at time t specified in world space. Suppose that we consider this vector fixed to the body; that is, $r(t)$ moves along with the rigid body through world space. Since $r(t)$ is a direction, it is independent of any translational effects; in particular, $\dot{r}(t)$ is independent of $v(t)$. To study $\dot{r}(t)$, we decompose $r(t)$ into vectors a and b, where a is parallel to $\omega(t)$ and b is perpendicular to $\omega(t)$. Suppose the rigid body were to maintain a constant angular velocity, so that the tip of $r(t)$ traces out a circle centered on the $\omega(t)$ axis (figure 4). The radius of this circle is $|b|$. Since the tip of the vector $r(t)$ is instantaneously moving along this circle, the instantaneous change of $r(t)$ is perpendicular to both b and $\omega(t)$. Since the tip of $r(t)$ is moving in a circle of radius b, the instantaneous velocity of $r(t)$ has magnitude $|b||\omega(t)|$. Since b and $\omega(t)$ are perpendicular, their cross product has magnitude

$$|\omega(t) \times b| = |\omega(t)|\,|b|. \tag{2–5}$$

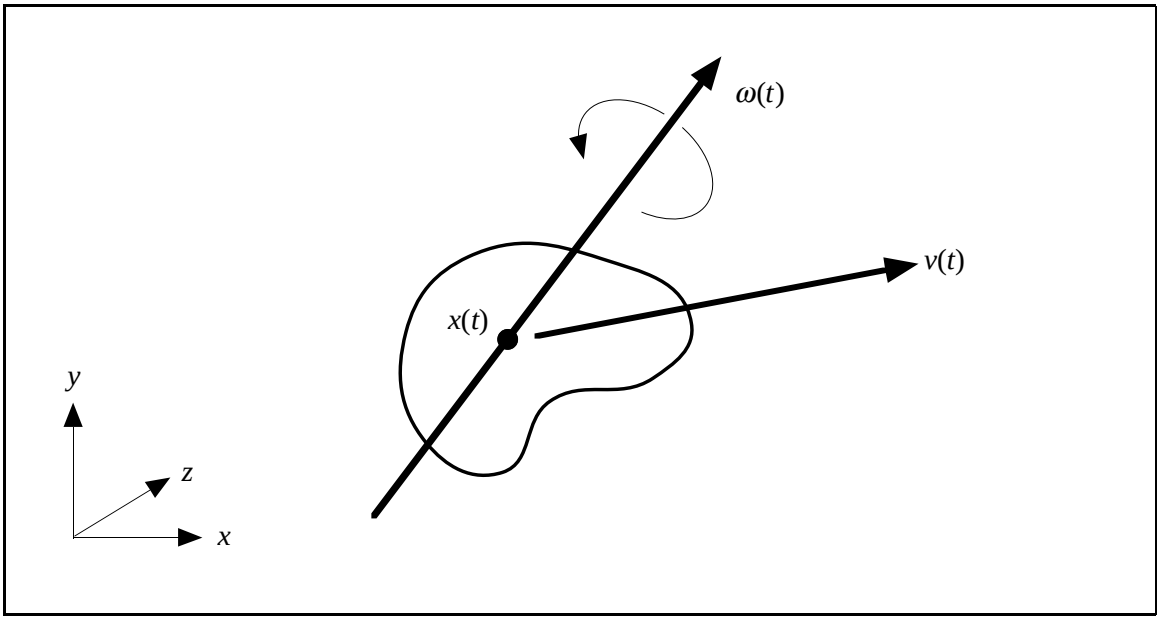

Figure 3: Linear velocity $v(t)$ and angular velocity $\omega(t)$ of a rigid body.

Putting this together, we can write $\dot{r}(t) = \omega(t) \times (b)$. However, since $r(t) = a + b$ and a is parallel to $\omega(t)$, we have $\omega(t) \times a = 0$ and thus

$$\dot{r}(t) = \omega(t) \times b = \omega(t) \times b + \omega(t) \times a = \omega(t) \times (b + a). \tag{2–6}$$

Thus, we can simply express the rate of change of a vector as

$$\dot{r}(t) = \omega(t) \times r(t). \tag{2–7}$$

Let's put all this together now. At time t, we know that the direction of the x axis of the rigid body in world space is the first column of $R(t)$, which is

$$\begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix}.$$

At time t, the derivative of the first column of $R(t)$ is just the rate of change of this vector: using the cross product rule we just discovered, this change is

$$\omega(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix}.$$

The same obviously holds for the other two columns of $R(t)$. This means that we can write

$$\dot{R} = \begin{pmatrix} \omega(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} & \omega(t) \times \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} & \omega(t) \times \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \end{pmatrix}. \tag{2–8}$$

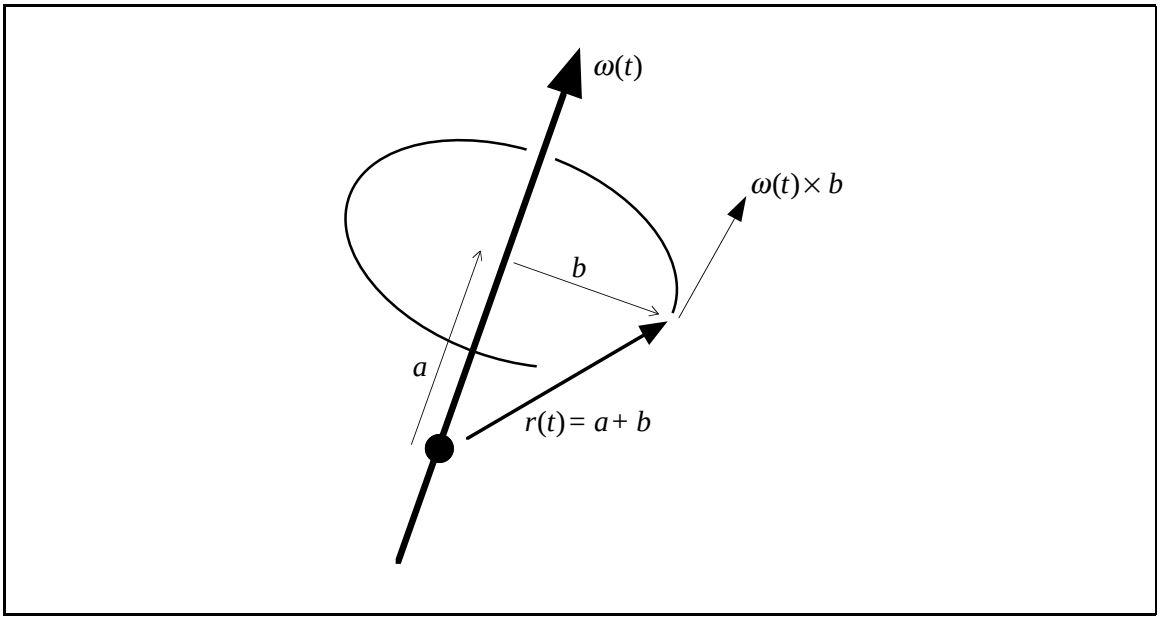

Figure 4: The rate of change of a rotating vector. As the tip of $r(t)$ spins about the $\omega(t)$ axis, it traces out a circle of diameter $|b|$. The speed of the tip of $r(t)$ is $|\omega(t)||b|$.

This is too cumbersome an expression to tote around though. To simplify things, we'll use the following trick. If a and b are 3-vectors, then $a \times b$ is the vector

$$\begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix}.$$

Given the vector a, let us define a^* to be the matrix

$$\begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix}.$$

Then[2]

$$a^* b = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix} = a \times b. \qquad (2\text{–}9)$$

Using the "$*$" notation, we can rewrite $\dot{R}(t)$ more simply as

$$\dot{R}(t) = \left(\omega(t)^* \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \quad \omega(t)^* \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \omega(t)^* \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right). \qquad (2\text{–}10)$$

[2]This looks a little too "magical" at first. Did someone discover this identity accidentally? Is it a relation that just happens to work? This construct can be derived by considering what's known as *infinitesimal* rotations. The interested reader might wish to read chapter 4.8 of Goldstein[10] for a more complete derivation of the a^* matrix.

By the rules of matrix multiplication, we can factor this into

$$\dot{R}(t) = \omega(t)^* \left(\begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} \quad \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} \quad \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \right) \tag{2–11}$$

which is a matrix-matrix multiplication. But since the matrix on the right is $R(t)$ itself, we get simply that

$$\dot{R}(t) = \omega(t)^* R(t). \tag{2–12}$$

This, at last, gives us the relation we wanted between $\dot{R}(t)$ and $\omega(t)$. Note the correspondence between $\dot{r}(t) = \omega(t) \times r(t)$ for a vector, and $\dot{R}(t) = \omega(t)^* R(t)$ for the rotation matrix.

2.4 Mass of a Body

In order to work out some derivations, we'll need to (conceptually) perform some integrations over the volume of our rigid body. To make these derivations simpler, we're going to temporarily imagine that a rigid body is made up of a large number of small particles. The particles are indexed from 1 to N. The mass of the ith particle is m_i, and each particle has a (constant) location r_{0i} in body space. The location of the ith particle in world space at time t, denoted $r_i(t)$, is therefore given by the formula

$$r_i(t) = R(t) r_{0i} + x(t). \tag{2–13}$$

The total mass of the body, M, is the sum

$$M = \sum_{i=1}^{N} m_i. \tag{2–14}$$

(Henceforth, summations are assumed to be summed from 1 to N with index variable i.)

2.5 Velocity of a Particle

The *velocity* $\dot{r}_i(t)$ of the ith particle is obtained by differentiating equation (2–13): using the relation $\dot{R}(t) = \omega^* R(t)$, we obtain

$$\dot{r}_i(t) = \omega^* R(t) r_{0i} + v(t). \tag{2–15}$$

We can rewrite this as

$$\begin{aligned} \dot{r}_i(t) &= \omega(t)^* R(t) r_{0i} + v(t) \\ &= \omega(t)^* \left(R(t) r_{0i} + x(t) - x(t) \right) + v(t) \\ &= \omega(t)^* \left(r_i(t) - x(t) \right) + v(t) \end{aligned} \tag{2–16}$$

using the definition of $r_i(t)$ from equation (2–13). Recall from the definition of the "$*$" operator that $\omega(t)^* a = \omega(t) \times a$ for any vector a. Using this, we can simply write

$$\dot{r}_i(t) = \omega(t) \times \left(r_i(t) - x(t) \right) + v(t). \tag{2–17}$$

Note that this separates the velocity of a point on a rigid body into two components (figure 5): a linear component $v(t)$, and an angular component $\omega \times (r_i(t) - x(t))$.

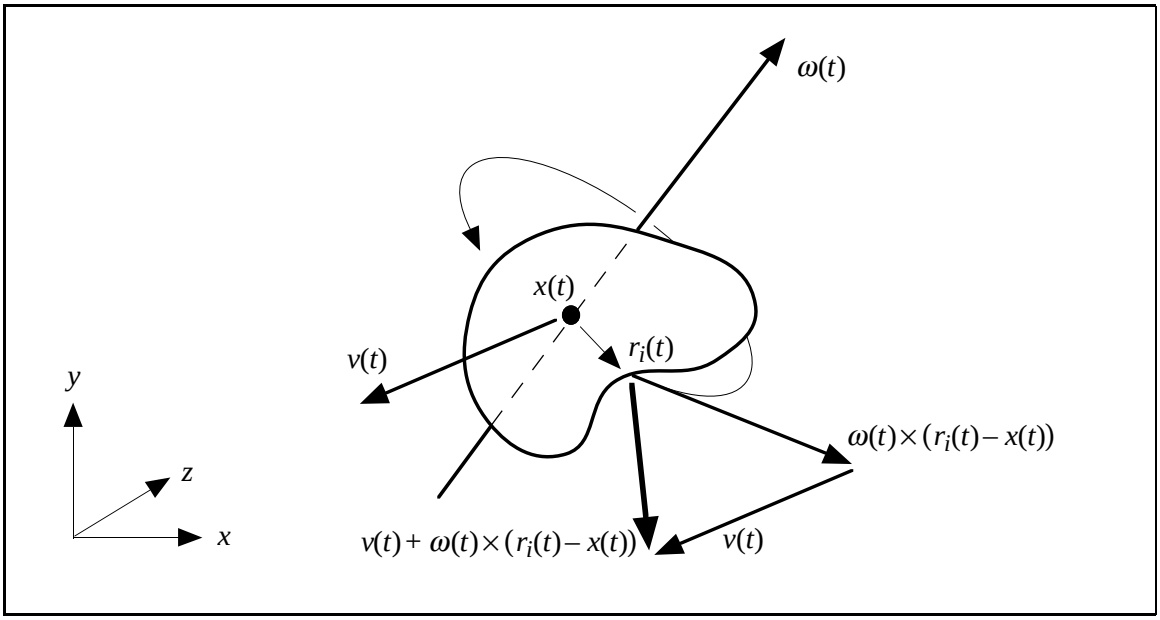

Figure 5: The velocity of the ith point of a rigid body in world space. The velocity of $r_i(t)$ can be decomposed into a linear term $v(t)$ and an angular term $\omega(t) \times (r_i(t) - x(t))$.

2.6 Center of Mass

Our definition of the center of mass is going to enable us to likewise separate the dynamics of bodies into linear and angular components. The center of mass of a body in world space is defined to be

$$\frac{\sum m_i r_i(t)}{M} \tag{2–18}$$

where M is the mass of the body (i.e. the sum of the individual masses m_i). When we say that we are using a center of mass coordinate system, we mean that in body space,

$$\frac{\sum m_i r_{0i}}{M} = \mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \tag{2–19}$$

Note that this implies that $\sum m_i r_{0i} = \mathbf{0}$ as well.

We have spoken of $x(t)$ as being the location of the center of mass at time t. Is this true? Yes: since the ith particle has position $r_i(t) = R(t)r_{0i} + x(t)$ at time t, the center of mass at time t is

$$\frac{\sum m_i r_i(t)}{M} = \frac{\sum m_i (R(t)r_{0i} + x(t))}{M} = \frac{R(t)\sum m_i r_{0i} + \sum m_i x(t)}{M} = x(t)\frac{\sum m_i}{M} = x(t).$$

Additionally, the relation

$$\sum m_i (r_i(t) - x(t)) = \sum m_i (R(t)r_{0i} + x(t) - x(t)) = R(t)\sum m_i r_{0i} = \mathbf{0} \tag{2–20}$$

is also very useful.

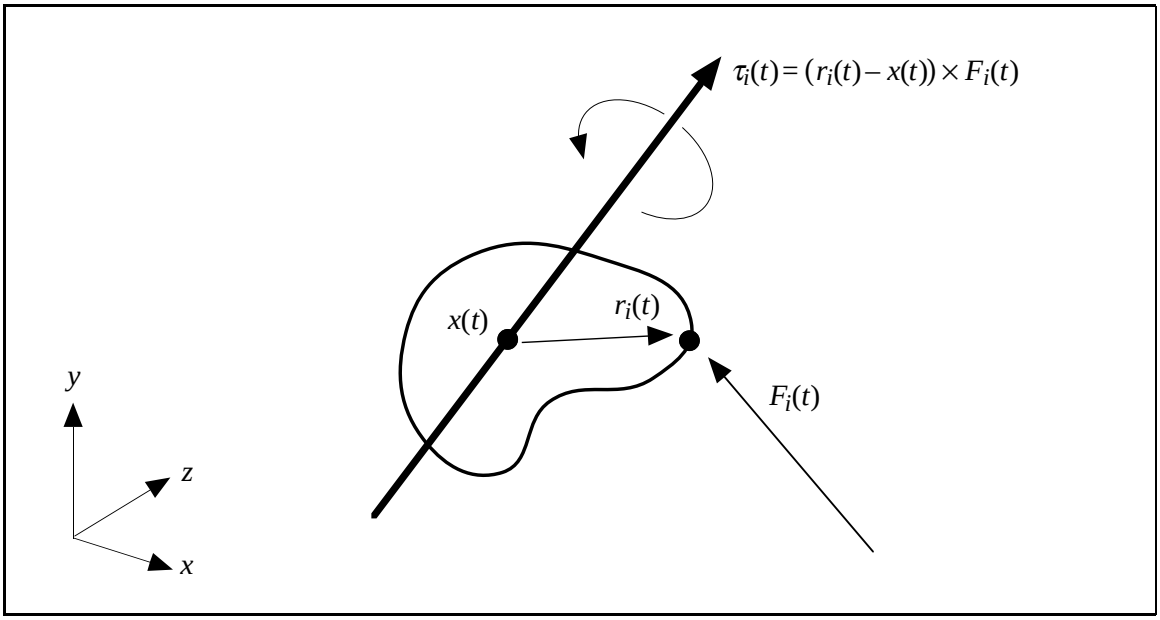

Figure 6: The torque $\tau_i(t)$ due to a force $F_i(t)$ acting at $r_i(t)$ on a rigid body.

2.7 Force and Torque

When we imagine a force acting on a rigid body due to some external influence (e.g. gravity, wind, contact forces), we imagine that the force acts on a particular particle of the body. (Remember that our particle model is conceptual only. We can have a force act at any geometrical location on or inside the body, because we can always imagine that there happens to be a particle at that exact location.) The location of the particle the force acts on defines the location at which the force acts. We will let $F_i(t)$ denote the total force from external forces acting on the ith particle at time t. Also, we define the external *torque* $\tau_i(t)$ acting on the ith particle as

$$\tau_i(t) = (r_i(t) - x(t)) \times F_i(t). \tag{2–21}$$

Torque differs from force in that the torque on a particle depends on the location $r_i(t)$ of the particle, relative to the center of mass $x(t)$. We can intuitively think of the direction of $\tau_i(t)$ as being the axis the body would spin about due to $F_i(t)$, if the center of mass were held firmly in place (figure 6).

The total external force $F(t)$ acting on the body is the sum of the $F_i(t)$:

$$F(t) = \sum F_i(t) \tag{2–22}$$

while the total external torque is defined similarly as

$$\tau(t) = \sum \tau_i(t) = \sum (r_i(t) - x(t)) \times F_i(t). \tag{2–23}$$

Note that $F(t)$ conveys no information about where the various forces acted on the body; however, $\tau(t)$ does tell us something about the distribution of the forces $F_i(t)$ over the body.

2.8 Linear Momentum

The linear momentum p of a particle with mass m and velocity v is defined as

$$p = mv. \tag{2–24}$$

The total linear momentum $P(t)$ of a rigid body is the sum of the products of the mass and velocity of each particle:

$$P(t) = \sum m_i \dot{r}_i(t). \tag{2–25}$$

From equation (2–17), the velocity $\dot{r}_i(t)$ of the ith particle is $\dot{r}_i(t) = v(t) + \omega(t) \times (r_i(t) - x(t))$. Thus, the total linear momentum of the body is

$$\begin{aligned} P(t) &= \sum m_i \dot{r}_i(t) \\ &= \sum \Big(m_i v(t) + m_i \omega(t) \times (r_i(t) - x(t)) \Big) \\ &= \sum m_i v(t) + \omega(t) \times \sum m_i \left(r_i(t) - x(t) \right). \end{aligned} \tag{2–26}$$

Because we are using a center of mass coordinate system, we can apply equation (2–20) and obtain

$$P(t) = \sum m_i v(t) = \Big(\sum m_i \Big) \, v(t) = M v(t). \tag{2–27}$$

This gives us the nice result that the total linear momentum of our rigid body is the same as if the body was simply a particle with mass M and velocity $v(t)$. Because of this, we have a simple transformation between $P(t)$ and $v(t)$: $P(t) = Mv(t)$ and $v(t) = P(t)/M$. Since M is a constant,

$$\dot{v}(t) = \frac{\dot{P}(t)}{M}. \tag{2–28}$$

The concept of linear momentum lets us express the effect of the total force $F(t)$ on a rigid body quite simply. Appendix A derives the relation

$$\dot{P}(t) = F(t) \tag{2–29}$$

which says that the change in linear momentum is equivalent to the total force acting on a body. Note that $P(t)$ tells us nothing about the rotational velocity of a body, which is good, because $F(t)$ also conveys nothing about the change of rotational velocity of a body!

Since the relationship between $P(t)$ and $v(t)$ is simple, we will be using $P(t)$ as a state variable for our rigid body, instead of $v(t)$. We could of course let $v(t)$ be a state variable, and use the relation

$$\dot{v}(t) = \frac{F(t)}{M}. \tag{2–30}$$

However, using $P(t)$ instead of $v(t)$ as a state variable will be more consistent with the way we will be dealing with angular velocity and acceleration.

2.9 Angular Momentum

While the concept of linear momentum is pretty intuitive ($P(t) = Mv(t)$), the concept of angular momentum (for a rigid body) is not. The only reason that one even bothers with the angular momentum of a rigid body is that it lets you write simpler equations than you would get if you stuck with angular velocity. With that in mind, it's probably best not to worry about attaching an intuitive physical explanation to angular momentum—all in all, it's a most unintuitive concept. Angular momentum ends up simplifying equations because it is conserved in nature, while angular *velocity* is not: if you have a body floating through space with no torque acting on it, the body's angular momentum is constant. This is not true for a body's angular velocity though: even if the angular momentum of a body is constant, the body's angular *velocity* may not be! Consequently, a body's angular velocity can vary even when no force acts on the body. Because of this, it ends up being simpler to choose angular momentum as a state variable over angular velocity.

For linear momentum, we have the relation $P(t) = Mv(t)$. Similarly, we define the total angular momentum $L(t)$ of a rigid body by the equation $L(t) = I(t)\omega(t)$, where $I(t)$ is a 3×3 matrix (technically a rank-two tensor) called the *inertia tensor*, which we will describe momentarily. The inertia tensor $I(t)$ describes how the mass in a body is distributed relative to the body's center of mass. The tensor $I(t)$ depends on the orientation of a body, but does not depend on the body's translation. Note that for both the angular and the linear case, momentum is a linear function of velocity—it's just that in the angular case the scaling factor is a matrix, while it's simply a scalar in the linear case. Note also that $L(t)$ is independent of any translational effects, while $P(t)$ is independent of any rotational effects.

The relationship between $L(t)$ and the total torque $\tau(t)$ is very simple: appendix A derives

$$\dot{L}(t) = \tau(t), \tag{2–31}$$

analogous to the relation $\dot{P}(t) = F(t)$.

2.10 The Inertia Tensor

The inertia tensor $I(t)$ is the scaling factor between angular momentum $L(t)$ and angular velocity $\omega(t)$. At a given time t, let r'_i be the displacement of the ith particle from $x(t)$ by defining $r'_i = r_i(t) - x(t)$. The tensor $I(t)$ is expressed in terms of r'_i as the symmetric matrix

$$I(t) = \sum \begin{pmatrix} m_i(r'^2_{iy} + r'^2_{iz}) & -m_i r'_{ix} r'_{iy} & -m_i r'_{ix} r'_{iz} \\ -m_i r'_{iy} r'_{ix} & m_i(r'^2_{ix} + r'^2_{iz}) & -m_i r'_{iy} r'_{iz} \\ -m_i r'_{iz} r'_{ix} & -m_i r'_{iz} r'_{iy} & m_i(r'^2_{ix} + r'^2_{iy}) \end{pmatrix} \tag{2–32}$$

For an actual implementation, we replace the finite sums with integrals over a body's volume in world space. The mass terms m_i are replaced by a density function. At first glance, it seems that we would need to evaluate these integrals to find $I(t)$ whenever the orientation $R(t)$ changes. This would be prohibitively expensive to do during a simulation unless the body's shape was so simple (for example, a sphere or cube) that that the integrals could be evaluated symbolically.

Fortunately, by using body-space coordinates we can cheaply compute the inertia tensor for any orientation $R(t)$ in terms of a precomputed integral in body-space coordinates. (This integral is typically computed before the simulation begins and should be regarded as one of the input parameters

describing a physical property of the body.) Using the fact that $r_i'^T r_i' = r_{ix}'^2 + r_{iy}'^2 + r_{iz}'^2$, we can rewrite $I(t)$ as the difference

$$I(t) = \sum m_i r_i'^T r_i' \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} m_i r_{ix}'^2 & m_i r_{ix}' r_{iy}' & m_i r_{ix}' r_{iz}' \\ m_i r_{iy}' r_{ix}' & m_i r_{iy}'^2 & m_i r_{iy}' r_{iz}' \\ m_i r_{iz}' r_{ix}' & m_i r_{iz}' r_{iy}' & m_i r_{iz}'^2 \end{pmatrix} \tag{2–33}$$

Taking the outer product multiplication of r_i' with itself, that is

$$r_i' r_i'^T = \begin{pmatrix} r_{ix}' \\ r_{iy}' \\ r_{iz}' \end{pmatrix} \begin{pmatrix} r_{ix}' & r_{iy}' & r_{iz}' \end{pmatrix} = \begin{pmatrix} r_{ix}'^2 & r_{ix}' r_{iy}' & r_{ix}' r_{iz}' \\ r_{iy}' r_{ix}' & r_{iy}'^2 & r_{ix}' r_{iz}' \\ r_{iz}' r_{ix}' & r_{iz}' r_{iy}' & r_{iz}'^2 \end{pmatrix} \tag{2–34}$$

and letting $\mathbf{1}$ denote the 3×3 identity matrix, we can express $I(t)$ simply as

$$I(t) = \sum m_i ((r_i'^T r_i')\mathbf{1} - r_i' r_i'^T) \tag{2–35}$$

How does this help?

Since $r_i(t) = R(t) r_{0i} + x(t)$ where r_{0i} is a constant, $r_i' = R(t) r_{0i}$. Then, since $R(t)R(t)^T = \mathbf{1}$,

$$\begin{aligned} I(t) &= \sum m_i ((r_i'^T r_i')\mathbf{1} - r_i' r_i'^T) \\ &= \sum m_i ((R(t)r_{0i})^T (R(t)r_{0i})\mathbf{1} - (R(t)r_{0i})(R(t)r_{0i})^T) \\ &= \sum m_i (r_{0i}^T R(t)^T R(t) r_{0i}\mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= \sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T). \end{aligned} \tag{2–36}$$

Since $r_{0i}^T r_{0i}$ is a scalar, we can rearrange things by writing

$$\begin{aligned} I(t) &= \sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= \sum m_i (R(t)(r_{0i}^T r_{0i}) R(t)^T \mathbf{1} - R(t) r_{0i} r_{0i}^T R(t)^T) \\ &= R(t) \left(\sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - r_{0i} r_{0i}^T) \right) R(t)^T. \end{aligned} \tag{2–37}$$

If we define I_{body} as the matrix

$$I_{body} = \sum m_i ((r_{0i}^T r_{0i})\mathbf{1} - r_{0i} r_{0i}^T) \tag{2–38}$$

then from the previous equation we have

$$I(t) = R(t) I_{body} R(t)^T. \tag{2–39}$$

Since I_{body} is specified in body-space, it is constant over the simulation. Thus, by precomputing I_{body} for a body before the simulation begins, we can easily compute $I(t)$ from I_{body} and the orientation matrix $R(t)$. Section 5.1 derives the body-space inertia tensor for a rectangular object in terms of an integral over the body's volume in body space.

Also, the inverse of $I(t)$ is given by the formula

$$\begin{aligned} I^{-1}(t) &= \left(R(t)I_{body}R(t)^T\right)^{-1} \\ &= \left(R(t)^T\right)^{-1} I_{body}^{-1}R(t)^{-1} \\ &= R(t)I_{body}^{-1}R(t)^T \end{aligned} \tag{2–40}$$

since, for rotation matrices, $R(t)^T = R(t)^{-1}$ and $\left(R(t)^T\right)^T = R(t)$. Clearly, I_{body}^{-1} is also a constant during the simulation.

2.11 Rigid Body Equations of Motion

Finally, we have covered all the concepts we need to define the state vector $\mathbf{Y}(t)$! For a rigid body, we will define $\mathbf{Y}(t)$ as

$$\mathbf{Y}(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix}. \tag{2–41}$$

Thus, the state of a rigid body is its position and orientation (describing spatial information), and its linear and angular momentum (describing velocity information). The mass M of the body and body-space inertia tensor I_{body} are constants, which we assume we know when the simulation begins. At any given time, the auxiliary quantities $I(t)$, $\omega(t)$ and $v(t)$ are computed by

$$v(t) = \frac{P(t)}{M}, \quad I(t) = R(t)I_{body}R(t)^T \qquad \text{and} \qquad \omega(t) = I(t)^{-1}L(t). \tag{2–42}$$

The derivative $\frac{d}{dt}\mathbf{Y}(t)$ is

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \omega(t)^*R(t) \\ F(t) \\ \tau(t) \end{pmatrix}. \tag{2–43}$$

The next section gives an implementation for the function `dydt` that computes $\frac{d}{dt}\mathbf{Y}(t)$.

One final note: rather than represent the orientation of the body as a matrix $R(t)$ in $\mathbf{Y}(t)$, it is better to use *quaternions.* Section 4 discusses using quaternions in place of rotation matrices. Briefly, a quaternion is a type of four element vector that can be used to represent a rotation. If we replace $R(t)$ in $\mathbf{Y}(t)$ with a quaternion $q(t)$, we can treat $R(t)$ as an auxiliary variable that is computed directly from $q(t)$, just as $\omega(t)$ is computed from $L(t)$. Section 4 derives a formula analogous to $\dot{R}(t) = \omega(t)^*R(t)$, that expresses $\dot{q}(t)$ in terms of $q(t)$ and $\omega(t)$.

3 Computing $\frac{d}{dt}\mathbf{Y}(t)$

Lets consider an implementation of the function `dydt` for rigid bodies. The code is written in C++, and we'll assume that we have datatypes (classes) called `matrix` and `triple` which implement, respectively, 3×3 matrices and points in 3-space. Using these datatypes, we'll represent a rigid

body by the structure

```
struct RigidBody {
        /* Constant quantities */
        double  mass;           /* mass M */
        matrix  Ibody,          /* I_body */
                Ibodyinv;       /* I_body^-1 (inverse of I_body) */

        /* State variables */
        triple  x;              /* x(t) */
        matrix  R;              /* R(t) */
        triple  P,              /* P(t) */
                L;              /* L(t) */

        /* Derived quantities (auxiliary variables) */
        matrix  Iinv;           /* I^-1(t) */
        triple  v,              /* v(t) */
                omega;          /* ω(t) */

        /* Computed quantities */
        triple  force,          /* F(t) */
                torque;         /* τ(t) */
};
```

and assume a global array of bodies

```
RigidBody Bodies[NBODIES];
```

The constant quantities `mass`, `Ibody` and `Ibodyinv` are assumed to have been calculated for each member of the array `Bodies`, before simulation begins. Also, the initial conditions for each rigid body are specified by assigning values to the state variables `x`, `R`, `P` and `L` of each member of `Bodies`. The implementation in this section represents orientation with a rotation matrix; section 4 describes the changes necessary to represent orientation by a quaternion.

We communicate with the differential equation solver `ode` by passing arrays of real numbers. Several bookkeeping routines are required:

```
/* Copy the state information into an array */
void State_to_Array(RigidBody *rb, double *y)
{
        *y++ = rb->x[0];            /* x component of position */
        *y++ = rb->x[1];            /* etc. */
        *y++ = rb->x[2];

        for(int i = 0; i < 3; i++)  /* copy rotation matrix */
                for(int j = 0; j < 3; j++)
                        *y++ = rb->R[i,j];
```

```
        *y++ = rb->P[0];
        *y++ = rb->P[1];
        *y++ = rb->P[2];

        *y++ = rb->L[0];
        *y++ = rb->L[1];
        *y++ = rb->L[2];
}
```

and

```
/* Copy information from an array into the state variables */
void Array_to_State(RigidBody *rb, double *y)
{
        rb->x[0] = *y++;
        rb->x[1] = *y++;
        rb->x[2] = *y++;

        for(int i = 0; i < 3; i++)
                for(int j = 0; j < 3; j++)
                        rb->R[i,j] = *y++;

        rb->P[0] = *y++;
        rb->P[1] = *y++;
        rb->P[2] = *y++;

        rb->L[0] = *y++;
        rb->L[1] = *y++;
        rb->L[2] = *y++;

        /* Compute auxiliary variables... */

        /* v(t) = P(t)/M */
        rb->v = rb->P / mass;

        /* I^-1(t) = R(t) I_body^-1 R(t)^T */
        rb->Iinv = R * Ibodyinv * Transpose(R);

        /* ω(t) = I^-1(t) L(t) */
        rb->omega = rb->Iinv * rb->L;
}
```

Note that `Array_to_State` is responsible for computing values for the auxiliary variables `Iinv`, `v` and `omega`. We'll assume that the appropriate arithmetic operations have been defined between real numbers, `triple`'s and `matrix`'s, and that `Transpose` returns the transpose of a matrix.

Examining these routines, we see that each rigid body's state is represented by $3+9+3+3=18$ numbers. Transfers between all the members of `Bodies` and an array `y` of size $18 \cdot$ `NBODIES` are implemented as

```
#define STATE_SIZE      18

void Array_to_Bodies(double y[])
{
        for(int i = 0; i < NBODIES; i++)
                Array_to_State(&Bodies[i], &y[i * STATE_SIZE]);
}
```

and

```
void Bodies_to_Array(double y[])
{
        for(int i = 0; i < NBODIES; i++)
                State_to_Array(&Bodies[i], &y[i * STATE_SIZE]);
}
```

Now we can implement `dydt`. Let's assume that the routine

```
void    Compute_Force_and_Torque(double t, RigidBody *rb);
```

computes the force $F(t)$ and torque $\tau(t)$ acting on the rigid body `*rb` at time `t`, and stores $F(t)$ and $\tau(t)$ in `rb->force` and `rb->torque` respectively. `Compute_Force_and_Torque` takes into account all forces and torques: gravity, wind, interaction with other bodies etc. Using this routine, we'll define `dydt` as

```
void dydt(double t, double y[], double ydot[])
{
        /* put data in y[] into Bodies[] */
        Array_to_Bodies(y);

        for(int i = 0; i < NBODIES; i++)
        {
                Compute_Force_and_Torque(t, &Bodies[i]);
                ddt_State_to_Array(&Bodies[i],
                                   &ydot[i * STATE_SIZE]);
        }
}
```

The numerical solver `ode` calls calls `dydt` and is responsible for allocating enough space for the arrays `y`, and `ydot` (`STATE_SIZE` $\cdot$ `NBODIES` worth for each). The function which does the real work of computing $\frac{d}{dt}\mathbf{Y}(t)$ and storing it in the array `ydot` is `ddt_State_to_Array`:

```
void ddt_State_to_Array(RigidBody *rb, double *ydot)
{
        /* copy d/dt x(t) = v(t) into ydot */
        *ydot++ = rb->v[0];
        *ydot++ = rb->v[1];
        *ydot++ = rb->v[2];

        /* Compute Rdot(t) = omega(t)*R(t) */
        matrix  Rdot = Star(rb->omega) * rb->R;

        /* copy Rdot(t) into array */
        for(int i = 0; i < 3; i++)
                for(int j = 0; j < 3; j++)
                        *ydot++ = Rdot[i,j];

        *ydot++ = rb->force[0];          /* d/dt P(t) = F(t) */
        *ydot++ = rb->force[1];
        *ydot++ = rb->force[2];

        *ydot++ = rb->torque[0];         /* d/dt L(t) = tau(t) */
        *ydot++ = rb->torque[1];
        *ydot++ = rb->torque[2];
}
```

The routine `Star`, used to calculate $\dot{R}(t)$ is defined as

```
matrix   Star(triple a);
```

and returns the matrix

$$\begin{pmatrix} 0 & -\texttt{a[2]} & \texttt{a[1]} \\ \texttt{a[2]} & 0 & -\texttt{a[0]} \\ -\texttt{a[1]} & \texttt{a[0]} & 0 \end{pmatrix}.$$

Given all of the above, actually performing a simulation is simple. Assume that the state variables of all `NBODIES` rigid bodies are initialized by a routine `InitStates`. We'll have our simulation run for 10 seconds, calling a routine `DisplayBodies` every $\frac{1}{30}$th of a second to display the bodies:

```
void RunSimulation()
{
        double  y0[STATE_SIZE * NBODIES],
                yfinal[STATE_SIZE * NBODIES];

        InitStates();
        Bodies_to_Array(yfinal);
```

```
        for(double t = 0; t < 10.0; t += 1./30.)
        {
                /* copy yfinal back to y0 */
                for(int i = 0; i < STATE_SIZE * NBODIES; i++)
                        y0[i] = yfinal[i];

                ode(y0, yfinal, STATE_SIZE * NBODIES,
                    t, t+1./30., dydt);

                /* copy d/dt Y(t + 1/30) into state variables */

                Array_to_Bodies(yfinal);
                DisplayBodies();
        }
}
```

4 Quaternions vs. Rotation Matrices

There is a better way to represent the orientation of a rigid body than using a 3×3 rotation matrix. For a number of reasons, *unit quaternions*, a type of four element vector normalized to unit length, are a better choice than rotation matrices[16].

For rigid body simulation, the most important reason to avoid using rotation matrices is because of numerical drift. Suppose that we keep track of the orientation of a rigid body according to the formula

$$\dot{R}(t) = \omega(t)^* R(t).$$

As we update $R(t)$ using this formula (that is, as we integrate this equation), we will inevitably encounter drift. Numerical error will build up in the coefficients of $R(t)$ so that $R(t)$ will no longer be precisely a rotation matrix. Graphically, the effect would be that applying $R(t)$ to a body would cause a skewing effect.

This problem can be alleviated by representing rotations with unit quaternions. Since quaternions have only four parameters, there is only one extra variable being used to describe the three freedoms of the rotation. In contrast, a rotation matrix uses nine parameters to describe three degrees of freedom; therefore, the degree of redundancy is noticeably lower for quaternions than rotation matrices. As a result, quaternions experience far less drift than rotation matrices. If it does become necessary to account for drift in a quaternion, it is because the quaternion has lost its unit magnitude[3]. This is easily correctable by renormalizing the quaternion to unit length. Because of these two properties, it is desirable to represent the orientation of a body directly as a unit quaternion $q(t)$. We will still express angular velocity as a vector $\omega(t)$. The orientation matrix $R(t)$, which is needed to compute $I^{-1}(t)$, will be computed as an auxiliary variable from $q(t)$.

We will write a quaternion $s + v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$ as the pair

$$[s, v].$$

[3] Any quaternion of unit length corresponds to a rotation, so quaternions deviate from representing rotations only if they lose their unit length. These notes will deal with that problem in a very simplistic way.

Using this notation, quaternion multiplication is

$$[s_1, v_1][s_2, v_2] = [s_1 s_2 - v_1 \cdot v_2, s_1 v_2 + s_2 v_1 + v_1 \times v_2]. \tag{4–1}$$

A rotation of θ radians about a unit axis u is represented by the unit quaternion

$$[\cos(\theta/2), \sin(\theta/2)u].$$

In using quaternions to represent rotations, if q_1 and q_2 indicate rotations, then $q_2 q_1$ represents the composite rotation of q_1 followed by q_2.[4] In a moment, we'll show how to change the routines of section 3 to handle the quaternion representation for orientation. Before we can make these changes though, we'll need a formula for $\dot{q}(t)$. Appendix B derives the formula

$$\dot{q}(t) = \tfrac{1}{2}\omega(t)q(t). \tag{4–2}$$

where the multiplication $\omega(t)q(t)$ is a shorthand for multiplication between the quaternions $[0, \omega(t)]$ and $q(t)$. Note the similarity between equation (4–2) and

$$\dot{R}(t) = \omega(t)^* R(t).$$

To actually use a quaternion representation, we'll need to redefine the type `RigidBody`:

```
struct RigidBody {
        /* Constant quantities */
        double  mass;           /* mass M */
        matrix  Ibody,          /* I_body */
                Ibodyinv;       /* I_body^-1 (inverse of I_body) */

        /* State variables */
        triple  x;              /* x(t) */
        quaternion q;           /* q(t) */
        triple  P,              /* P(t) */
                L;              /* L(t) */

        /* Derived quantities (auxiliary variables) */
        matrix  Iinv,           /* I^-1(t) */
                R;              /* R(t) */
        triple  v,              /* v(t) */
                omega;          /* omega(t) */

        /* Computed quantities */
        triple  force,          /* F(t) */
                torque;         /* tau(t) */
};
```

[4] This is according to the convention that the rotation of a point p by a quaternion q is qpq^{-1}. Be warned! This is *opposite* the convention for rotation in the original paper Shoemake[16], but it is in correspondence with some more recent versions of Shoemake's article. Writing a composite rotation as $q_2 q_1$ parallels our matrix notation for composition of rotations.

Next, in the routine `State_to_Array`, we'll replace the double loop

```
for(int i = 0; i < 3; i++)      /* copy rotation matrix */
        for(int j = 0; j < 3; j++)
                *y++ = rb->R[i,j];
```

with

```
/*
        Assume that a quaternion is represented in
        terms of elements 'r' for the real part,
        and 'i', 'j', and 'k' for the vector part.
*/

*y++ = rb->q.r;
*y++ = rb->q.i;
*y++ = rb->q.j;
*y++ = rb->q.k;
```

A similar change is made in `Array_to_State`. Also, since `Array_to_State` is responsible for computing the auxiliary variable $I^{-1}(t)$, which depends on $R(t)$, `Array_to_State` must also compute $R(t)$ as an auxiliary variable: in the section

```
/* Compute auxiliary variables... */

/* v(t) = P(t)/M */
rb->v = rb->P / mass;

/* I^-1(t) = R(t) I_body^-1 R(t)^T */
rb->Iinv = R * Ibodyinv * Transpose(R);

/* ω(t) = I^-1(t) L(t) */
rb->omega = rb->Iinv * rb->L;
```

we add the line

```
rb->R = quaternion_to_matrix(normalize(rb->q));
```

prior to computing `rb->Iinv`. The routine `normalize` returns `q` divided by its length; this unit length quaternion returned by `normalize` is then passed to `quaternion_to_matrix` which returns a 3×3 rotation matrix. Given a quaternion $q = [s, v]$, `quaternion_to_matrix` returns the matrix

$$\begin{pmatrix} 1-2v_y^2-2v_z^2 & 2v_xv_y-2sv_z & 2v_xv_z+2sv_y \\ 2v_xv_y+2sv_z & 1-2v_x^2-2v_z^2 & 2v_yv_z-2sv_x \\ 2v_xv_z-2sv_y & 2v_yv_z+2sv_x & 1-2v_x^2-2v_y^2 \end{pmatrix}.$$

In case you need to convert from a rotation matrix to a quaternion,

```
quaternion matrix_to_quaternion(const matrix &m)
{
        quaternion      q;
        double          tr, s;

        tr = m[0,0] + m[1,1] + m[2,2];

        if(tr >= 0)
        {
                s = sqrt(tr + 1);
                q.r = 0.5 * s;
                s = 0.5 / s;
                q.i = (m[2,1] - m[1,2]) * s;
                q.j = (m[0,2] - m[2,0]) * s;
                q.k = (m[1,0] - m[0,1]) * s;
        }
        else
        {
                int i = 0;

                if(m[1,1] > m[0,0])
                        i = 1;
                if(m[2,2] > m[i,i])
                        i = 2;

                switch (i)
                {
                case 0:
                        s = sqrt((m[0,0] - (m[1,1] + m[2,2])) + 1);
                        q.i = 0.5 * s;
                        s = 0.5 / s;
                        q.j = (m[0,1] + m[1,0]) * s;
                        q.k = (m[2,0] + m[0,2]) * s;
                        q.r = (m[2,1] - m[1,2]) * s;
                        break;
                case 1:
                        s = sqrt((m[1,1] - (m[2,2] + m[0,0])) + 1);
                        q.j = 0.5 * s;
                        s = 0.5 / s;
                        q.k = (m[1,2] + m[2,1]) * s;
                        q.i = (m[0,1] + m[1,0]) * s;
                        q.r = (m[0,2] - m[2,0]) * s;
                        break;
                case 2:
                        s = sqrt((m[2,2] - (m[0,0] + m[1,1])) + 1);
```

```
                        q.k = 0.5 * s;
                        s = 0.5 / s;
                        q.i = (m[2,0] + m[0,2]) * s;
                        q.j = (m[1,2] + m[2,1]) * s;
                        q.r = (m[1,0] - m[0,1]) * s;
                }

        }
        return q;
}
```

The matrix `m` is structured so that `m[0,0]`, `m[0,1]` and `m[0,2]` form the first row (not column) of `m`.

The routines `Array_to_Bodies` and `Bodies_to_Array` don't need any changes at all, but note that the constant `STATE_SIZE` changes from 18 to 13, since a quaternion requires five less elements than a rotation matrix. The only other change we need is in `ddt_State_to_Array`. Instead of

```
matrix  Rdot = Star(rb->omega) * rb->R;

/* copy Ṙ(t) into array */
for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)
                *ydot++ = Rdot[i,j];
```

we'll use

```
quaternion      qdot = .5 * (rb->omega * rb->q);
*ydot++ = qdot.r;
*ydot++ = qdot.i;
*ydot++ = qdot.j;
*ydot++ = qdot.k;
```

We're assuming here that the multiplication between the triple `rb->omega` and the quaternion `rb->q` is defined to return the quaternion product

$$[0, \texttt{rb->omega}]\texttt{q}.$$

5 Examples

5.1 Inertia Tensor of a Block

Let us calculate the inertia tensor I_{body} of the rectangular block in figure 7. The block has dimensions $x_0 \times y_0 \times z_0$. As required, the center of mass of the block is at the origin. Thus, the extent of the block is from $-\frac{x_0}{2}$ to $\frac{x_0}{2}$ along the x axis, and similarly for the y and z axes. To calculate the inertia tensor, we must treat the sums in equation (2–32) as integrals over the volume of the block. Let us assume that the block has constant unit density. This means that the density function $\rho(x, y, z)$ is always one. Since the block has volume $x_0 y_0 z_0$, the mass M of the block is $M = x_0 y_0 z_0$. Then, in

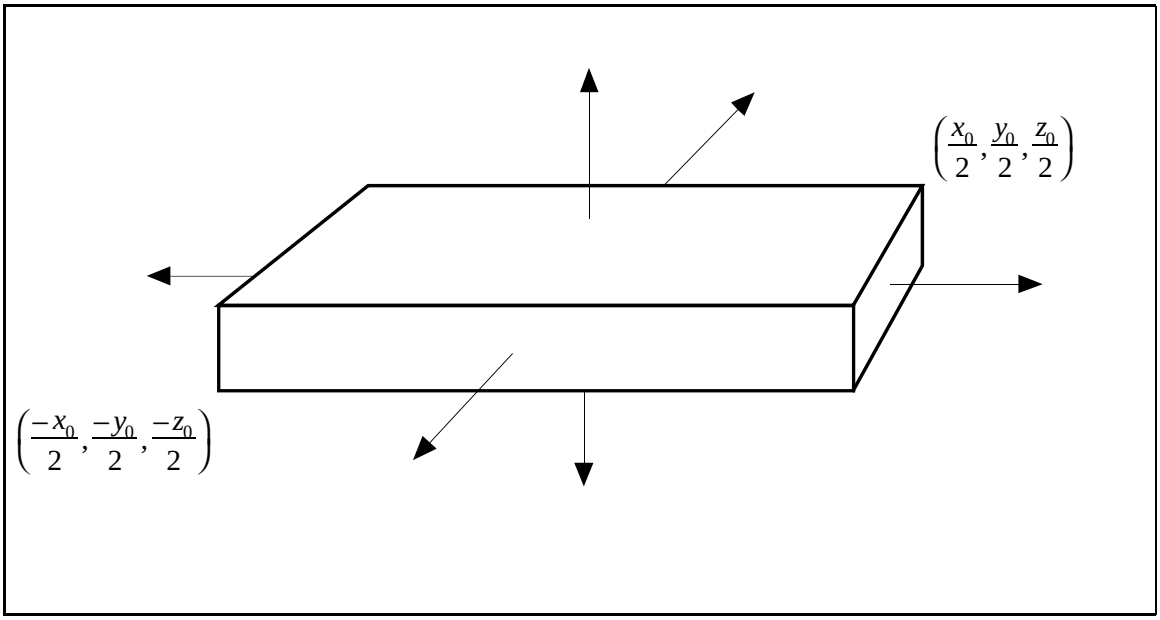

Figure 7: A rectangular block of constant unit density, with center of mass at (0,0,0).

body space,

$$
\begin{aligned}
I_{xx} &= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} \rho(x,y,z)(y^2+z^2)\,dx\,dy\,dz = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} y^2+z^2\,dx\,dy\,dz \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} y^2 z + \frac{z^3}{3}\Big|_{z=\frac{-z_0}{2}}^{z=\frac{z_0}{2}} dx\,dy \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} y^2 z_0 + \frac{z_0^3}{12}\,dx\,dy \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \frac{y_0^3}{3} z_0 + \frac{z_0^3}{12} y_0 \Big|_{y=\frac{-y_0}{2}}^{y=\frac{y_0}{2}} dx \\
&= \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \frac{y_0^3 z_0}{12} + \frac{z_0^3 y_0}{12}\,dx \\
&= \frac{y_0^3 z_0}{12} + \frac{z_0^3 y_0}{12}\Big|_{x=\frac{-x_0}{2}}^{x=\frac{x_0}{2}} \frac{y_0^3 z_0 x_0}{12} + \frac{z_0^3 y_0 x_0}{12} = \frac{x_0 y_0 z_0}{12}(y_0^2+z_0^2) = \frac{M}{12}(y_0^2+z_0^2).
\end{aligned}
\tag{5–1}
$$

Similarly, $I_{yy} = \frac{M}{12}(x_0^2 + z_0^2)$ and $I_{zz} = \frac{M}{12}(x_0^2 + y_0^2)$. Now, the off-diagonal terms, such as I_{xy}, are

$$
I_{xy} = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} \rho(x,y,z)(xy)\,dx\,dy\,dz = \int_{\frac{-x_0}{2}}^{\frac{x_0}{2}} \int_{\frac{-y_0}{2}}^{\frac{y_0}{2}} \int_{\frac{-z_0}{2}}^{\frac{z_0}{2}} xy\,dx\,dy\,dz = 0 \tag{5–2}
$$

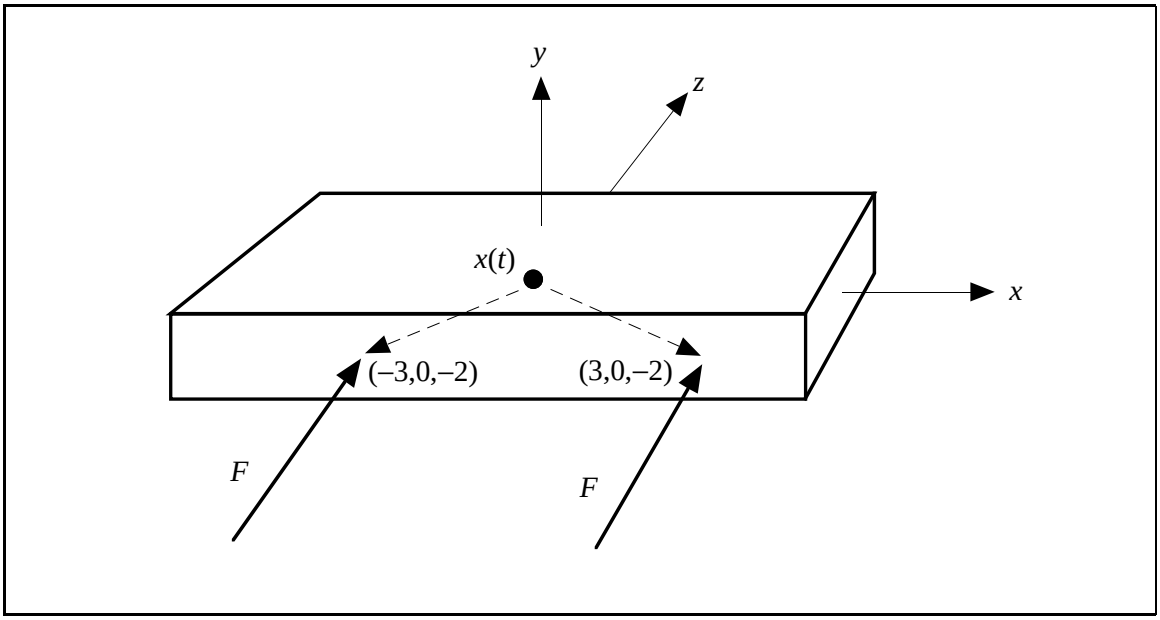

Figure 8: A block acted on by two equal forces F at two different points.

(and similarly for the others) because the integrals are all symmetric. Thus, the inertia tensor of the block is

$$I_{body} = \frac{M}{12}\begin{pmatrix} y_0^2 + z_0^2 & 0 & 0 \\ 0 & x_0^2 + z_0^2 & 0 \\ 0 & 0 & x_0^2 + y_0^2 \end{pmatrix}. \tag{5–3}$$

5.2 A Uniform Force Field

Suppose a uniform force acts on each particle of a body. For example, we typically describe a gravitational field as exerting a force $m_i g$ on each particle of a rigid body, where g is a vector pointing downwards. The net force F_g acting due to gravity on the body then is

$$F_g = \sum m_i g = Mg \tag{5–4}$$

which yields an acceleration of $\frac{Mg}{g} = g$ of the center of mass, as expected. What is the torque due to the gravitational field? The net torque is the sum

$$\sum (r_i(t) - x(t)) \times m_i g = \left(\sum m_i (r_i(t) - x(t))\right) \times g = \mathbf{0} \tag{5–5}$$

by equation (2–20). We see from this that a uniform gravitational field can have no effect on the angular momentum of a body. Furthermore, the gravitational field can be treated as a single force Mg acting on the body at its center of mass.

5.3 Rotation Free Movement of a Body

Now, let us consider some forces acting on the block of figure 8. Suppose that an external force $F = (0, 0, f)$ acts on the body at points $x(t) + (-3, 0, -2)$ and $x(t) + (3, 0, -2)$. We would expect

that this would cause the body to accelerate linearly, without accelerating angularly. The net force acting on the body is $(0, 0, 2f)$, so the acceleration of the center of mass is

$$\frac{2f}{M}$$

along the z axis. The torque due to the force acting at $x(t) + (-3, 0, -2)$ is

$$((x(t) + \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix}) - x(t)) \times F = \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix} \times F$$

while the torque due to the force acting at $x(t) + (3, 0, -2)$ is

$$((x(t) + \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix}) - x(t)) \times F = \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix} \times F.$$

The total torque τ is therefore

$$\tau = \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix} \times F + \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix} \times F = (\begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix} + \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix}) \times F = \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix} \times F.$$

But this gives

$$\tau = \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ f \end{pmatrix} = \mathbf{0}.$$

As expected then, the forces acting on the block impart no angular acceleration to the block.

5.4 Translation Free Movement of a Body

Suppose now that an external force $F_1 = (0, 0, f)$ acts on the body at point $x(t) + (-3, 0, -2)$ and an external force $F_2 = (0, 0, -f)$ acts on the body at point $x(t) + (3, 0, 2)$ (figure 9). Since $F_1 = -F_2$, the net force acting on the block is $F_1 + F_2 = \mathbf{0}$, so there is no acceleration of the center of mass. On the other hand, the net torque is

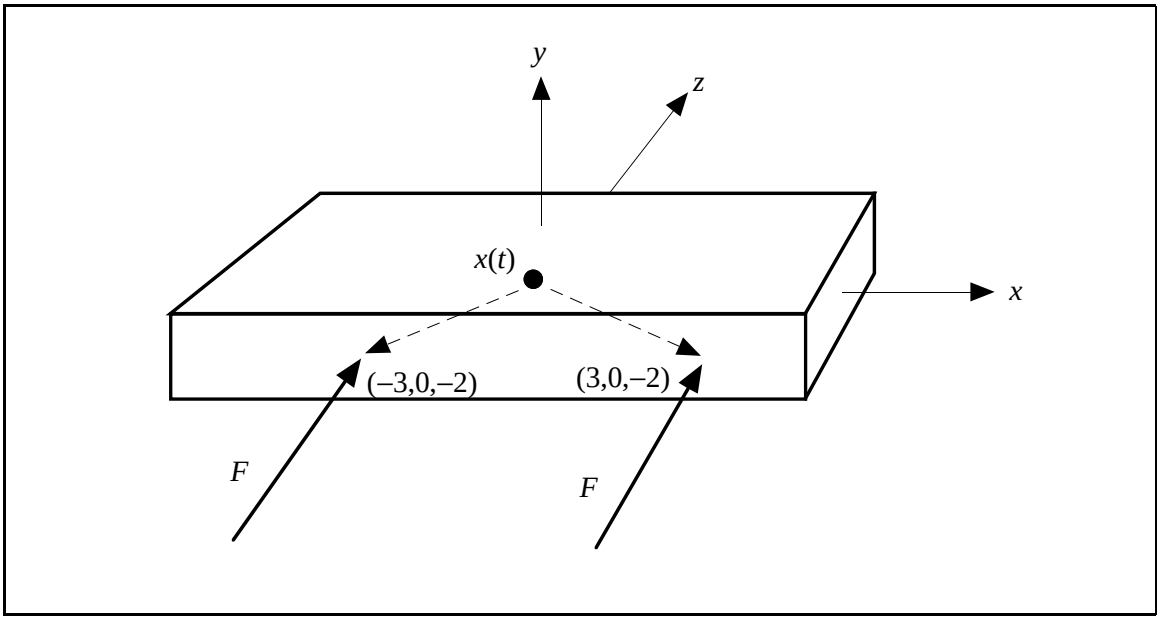

Figure 9: A block acted on by two opposite forces F_1 and $F_2 = -F_1$, at two different points.

$$((x(t) + \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix}) - x(t)) \times F_1 +$$
$$((x(t) + \begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix}) - x(t)) \times F_2 = \begin{pmatrix} -3 \\ 0 \\ -2 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ f \end{pmatrix} + \begin{pmatrix} 3 \\ 0 \\ -2 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ -f \end{pmatrix} \quad (5\text{–}6)$$
$$= \begin{pmatrix} 0 \\ 3f \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 3f \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 6f \\ 0 \end{pmatrix}.$$

Thus, the net torque is $(0, 6f, 0)$, which is parallel to the y axis. The final result is that the forces acting on the block cause it to angularly accelerate about the y axis.

5.5 Force vs. Torque Puzzle

In considering the effect of a force acting at a point on a body, it sometimes seems that the force is being considered twice. That is, if a force F acts on a body at a point $r + x(t)$ in space, then we first consider F as accelerating the center of mass, and then consider F as imparting a spin to the body.

This gives rise to what at first seems a paradox: Consider the long horizontal block of figure 10 which is initially at rest. Suppose that a force F acts on the block at the center of mass for some period of time, say, ten seconds. Since the force acts at the center of mass, no torque is exerted on the body. After ten seconds, the body will have acquired some linear velocity v. The body will not have acquired any angular velocity; thus the kinetic energy of the block will be $\frac{1}{2}M|v|^2$.

Now suppose that the same force F is applied off-center to the body as shown in figure 11. Since the force acting on the body is the same, the acceleration of the center of mass is the same. Thus,

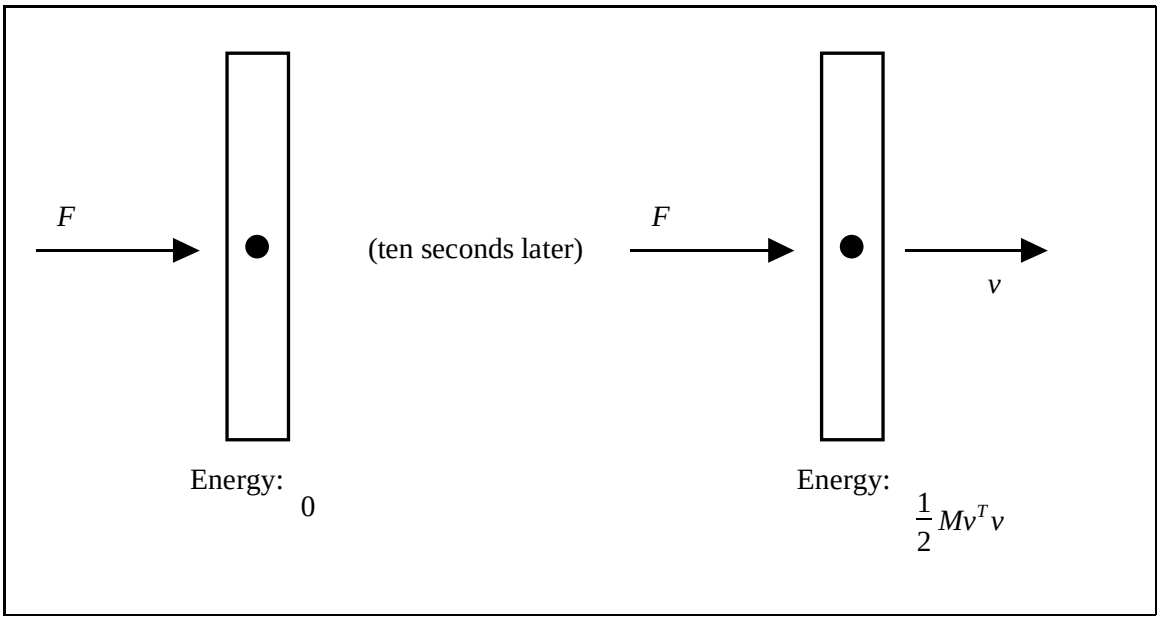

Figure 10: A rectangular block acted on by a force through its center of mass.

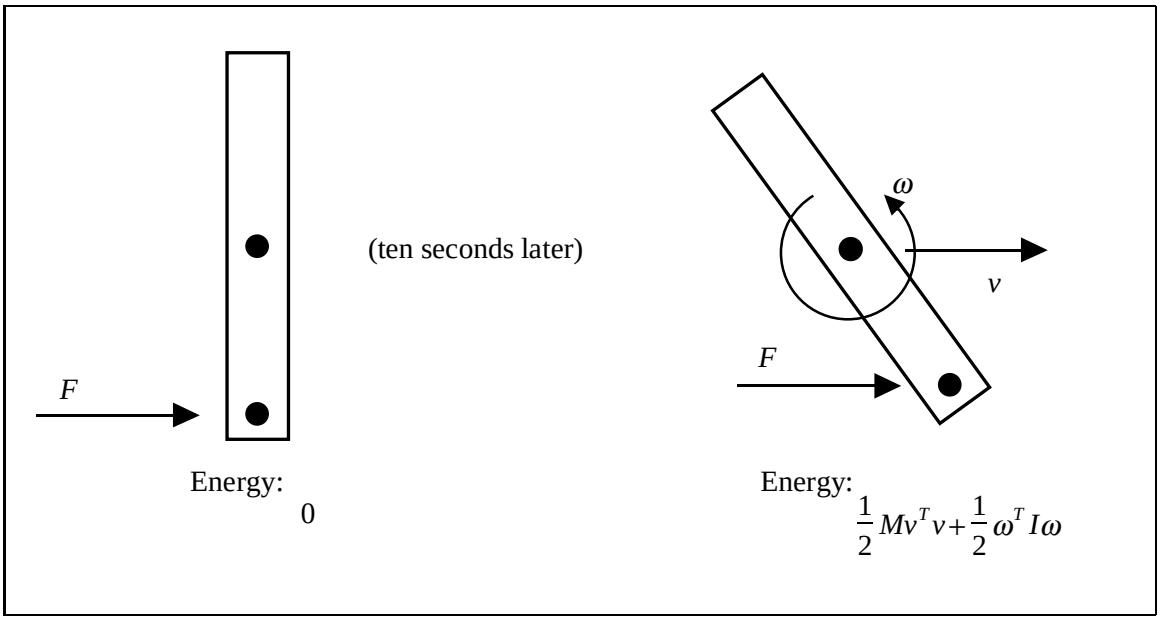

Figure 11: A block acted on by a force, off-center of the center of mass.

after ten seconds, the body will again have linear velocity v. However, after ten seconds, the body will have picked up some angular velocity ω, since the force F, acting off center, now exerts a torque on the body. Since the kinetic energy is (see appendix C)

$$\frac{1}{2}M|v|^2 + \frac{1}{2}\omega^T I\omega$$

the kinetic energy of the block is higher than when the force acted through the center of mass. But if identical forces pushed the block in both cases, how can the energy of the block be different? Hint: Energy, or work, is the integral of force over distance.

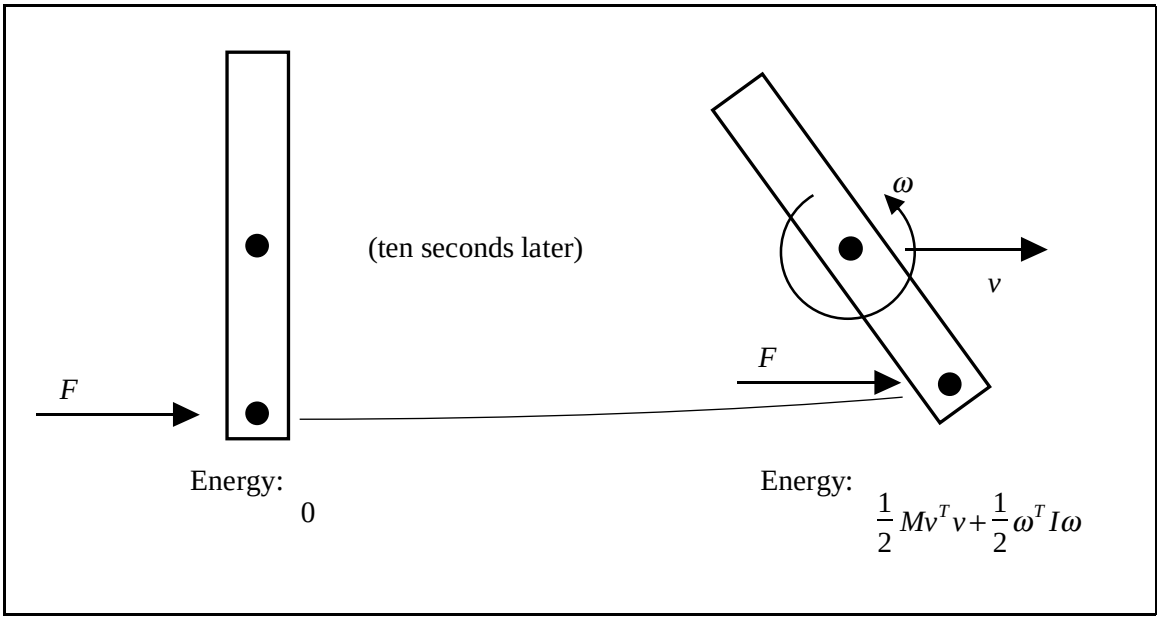

Figure 12: The path the force acts over is longer than in figure 10. As a result, the force does more work, imparting a larger kinetic energy to the block.

Figure 12 shows why the force acting off center results in a higher kinetic energy. The kinetic energy of the block is equivalent to the work done by the force. The work done by the force is the integral of the force over the path traveled in applying that force. In figure 11, where the force acts off the center of mass, consider the path traced out by the point where the force is applied. This path is clearly longer than the path taken by the center of mass in figure 10. Thus, when the force is applied off center, more work is done because the point p at which the force is applied traces out a longer path then when the force is applied at the center of mass.