Simulating Single Rigid Bodies

Richard Chaney 1999

1. Introduction

This document contains the following subjects :

- equations of motion for single rigid bodies
- efficient implementation using diagonalised coordinates and quaternions
- evolving the state with the 4th order Runge-Kutta method
- using the C++ source code for the `RigidBody` class

This document is intended for programmers who want to use the equations in their own code, or use the provided `RigidBody` class. It does not bother with deriving all the equations.

2. Data structure for a rigid body

2.1. State variables

The following data is needed to describe the current state of a rigid body. These are called the *state variables*. They are :

State variable	Format	Units	Description
$\underline{x}$	3D vector	metres	Position of centre of mass in world coordinates.
$\underline{\dot{x}}$	3D vector	metres / second	Velocity of centre of mass in world coordinates.
$\underline{\omega_b}$	3D vector	radians / second	Angular velocity in body coordinates. The length of this vector is the angular velocity in radians / second. The direction is the axis of rotation. Rotation is clockwise when looking along the axis.
Q	quaternion	n/a	Orientation quaternion (see section 4.1).
R	3 x 3 matrix	n/a	Orientation matrix. This is derived from the quaternion, so it is not really a state variable. This matrix transforms from body coordinates to world coordinates.

2.2. Constant properties

A rigid body has some constant properties. They are :

Property	Format	Units	Description
m	scalar	kg	Mass of body.
I_b	3 x 3 matrix	kg-m^2	Moment of inertia matrix relative to body axes. Can be diagonalised, so only 3 numbers are needed (see below).

The mass particles of the body should be translated so the centre of mass is at the origin of the body coordinates. This can be done by finding the centre of mass of the original mass particles, and subtracting it from the coordinates of each particle.

The centre of mass is found by calculating the following for each particle in the body. Each particle has a mass of Δm and coordinates (x, y, z) :

$$\begin{aligned} m &= \sum \Delta m \\ I_x &= \sum \Delta m x \\ I_y &= \sum \Delta m y \\ I_z &= \sum \Delta m z \end{aligned} \qquad (1)$$

$$\text{centre of mass} = \frac{1}{m}\begin{bmatrix} I_x \\ I_y \\ I_z \end{bmatrix} \qquad (2)$$

The moment of inertia matrix is defined as follows :

$$I_b = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} \qquad (3)$$

The elements of this matrix are calculated by summing across all the particles of mass in the body. The mass of each particle is multiplied by a function of its body coordinates. These are :

$$\begin{aligned} I_{xx} &= \sum \Delta m (y^2 + z^2) \\ I_{yy} &= \sum \Delta m (x^2 + z^2) \\ I_{zz} &= \sum \Delta m (x^2 + y^2) \\ I_{xy} &= \sum \Delta m x y \\ I_{xz} &= \sum \Delta m x z \\ I_{yz} &= \sum \Delta m y z \end{aligned} \qquad (4)$$

The moment of inertia matrix can be diagonalised by finding its eigen vectors. The eigen vectors are placed in a new matrix using one row for each eigen vector. This new matrix can be used to transform the mass particles of the body to a new coordinate system. In this new coordinate system the moment of inertia matrix is diagonal.

If you want to test this idea for yourself try the following :

1. Create a random collection of mass particles with (x, y, z) coordinates.
2. Find the centre of mass. Subtract this from each particle to shift the centre of mass to the origin.
3. Find the moment of inertia matrix.
4. Find the eigen vectors of the matrix. The `Vector3` class used by the `RigidBody` class has a method for this called `eigenvectors()`.
5. Make a matrix using the eigen vectors are rows. Transform each mass particle by this matrix (assuming you do the transformation as a matrix multiplied by a column vector).
6. Find the moment of inertia matrix for the transformed particles. It should be diagonal.

The `RigidBody` class has a method called `diagonalise()` which will shift the centre of mass to the origin, and diagonalise the moment of inertia matrix. This speeds up the calculations which evolve the state of the body. When this document refers to the body coordinates it always means the diagonalised body coordinates.

Note : When you diagonalise a rigid body you should also apply the same transform to any other descriptions of the body, such as those for graphics or collision detection. The `RigidBody` class sets up the variables `diagRotPos` and `diagLinPos` for doing this.

3. Linear position, velocity and acceleration

The linear movement of the rigid body is simple to calculate. Newtons law for force and acceleration gives the linear movement as follows :

$$\underline{F_w} = m\underline{\ddot{x}} \tag{5}$$

where :

$\underline{F_w}$	=	applied force on centre of mass (world coords)
m	=	mass of body
$\underline{\ddot{x}}$	=	acceleration of centre of mass (world coords)

This gives the following two differential equations to evolve the linear state of the rigid body :

$$\frac{d}{dt}\underline{x} = \underline{\dot{x}} \tag{6}$$

$$\frac{d}{dt}\underline{\dot{x}} = \frac{1}{m}\underline{F_w} \tag{7}$$

4. Angular position, velocity and acceleration

4.1. Defining the orientation of the body

The orientation of the body is defined by the matrix R. This matrix converts points in (diagonalised) body coords to points in world coords as follows :

$$\underline{r_w} = R\underline{r_b} + \underline{x} \tag{8}$$

where :

$\underline{r_w}$	=	point in world coords
$\underline{r_b}$	=	point in body coords
R	=	rotation matrix
$\underline{x}$	=	position of centre of mass in world coords

The matrix R should be orthogonal, which means each row should have unit length, and all rows are perpendicular. As this matrix gets evolved from one state to another it tends to lose this property. This would look like the matrix starting as a cube, and then becoming squashed out of shape.

To avoid this the quaternion Q is used instead. A quaternion uses only 4 numbers to define the orientation of the body, while the matrix uses 9. To define a valid orientation the quaternion must have unit length. It is much faster to normalise a quaternion to unit length than squash the R matrix back to being orthogonal.

A quaternion which defines a rotation θ radians about a unit length axis $\underline{a}$ is :

$$Q = \begin{bmatrix} Q_w \\ Q_x \\ Q_y \\ Q_z \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) \\ a_x \sin\left(\frac{\theta}{2}\right) \\ a_y \sin\left(\frac{\theta}{2}\right) \\ a_z \sin\left(\frac{\theta}{2}\right) \end{bmatrix} \tag{9}$$

To convert this into a matrix :

$$R = \begin{bmatrix} 1-2\left(Q_y{}^2 + Q_z{}^2\right) & 2\left(Q_x Q_y - Q_w Q_z\right) & 2\left(Q_x Q_z + Q_w Q_y\right) \\ 2\left(Q_x Q_y + Q_w Q_z\right) & 1-2\left(Q_z{}^2 + Q_x{}^2\right) & 2\left(Q_y Q_z - Q_w Q_x\right) \\ 2\left(Q_x Q_z - Q_w Q_y\right) & 2\left(Q_y Q_z + Q_w Q_x\right) & 1-2\left(Q_x{}^2 + Q_y{}^2\right) \end{bmatrix} \tag{10}$$

The rigid body state variable should store the orientation as a quaternion Q. The orientation is evolved from one frame to another using Q. The matrix R is calculated from Q whenever it changes. This is much faster than evolving the orientation using the matrix R.

4.2. Angular velocity

The angular velocity of the body is defined in body coords as :

$$\underline{\omega_b} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{11}$$

This vector defines the angular velocity in the following way :

- the length of the vector is the magnitude of the angular velocity in radians per second
- the direction of the vector is the axis of rotation. The body rotates clockwise when looking down the vector.

The derivative of the rotation matrix depends on the angular velocity as follows :

$$\dot{R} = R\underline{\tilde{\omega}_b} \tag{12}$$

where :

$$\underline{\tilde{\omega}_b} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{13}$$

The state variable for the orientation is actually a quaternion. The derivative of Q is as follows :

$$\frac{d}{dt}\begin{bmatrix} Q_w \\ Q_x \\ Q_y \\ Q_z \end{bmatrix} = \frac{1}{2}\begin{bmatrix} -Q_x & -Q_y & -Q_z \\ Q_w & -Q_z & Q_y \\ Q_z & Q_w & -Q_x \\ -Q_y & Q_x & Q_w \end{bmatrix} \cdot \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{14}$$

4.3. Angular acceleration

We can now define how the angular velocity changes when a torque is applied. Assume the torque applied in body coords is :

$$\underline{\tau_b} = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} \tag{15}$$

This torque is measured in Newton-metres (N-m). The torque applied by a force acting on the body is :

$$\underline{\tau_b} = \underline{r_b} \times \underline{F_b} \tag{16}$$

where :

$\underline{\tau_b}$	=	torque (body coords)
$\underline{r_b}$	=	vector from centre of mass to point of force application (body coords)
$\underline{F_b}$	=	force (body coords)

Assume the moment of inertia has been diagonalised using the process in section 2.2. The diagonalised moment of inertia in body coords is :

$$I_b = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{17}$$

The derivative of the angular velocity is as follows :

$$\frac{d}{dt}\omega_x = \frac{I_{yy} - I_{zz}}{I_{xx}}\omega_y\omega_z + \frac{\tau_x}{I_{xx}} \quad (18)$$

$$\frac{d}{dt}\omega_y = \frac{I_{zz} - I_{xx}}{I_{yy}}\omega_z\omega_x + \frac{\tau_y}{I_{yy}} \quad (19)$$

$$\frac{d}{dt}\omega_z = \frac{I_{xx} - I_{yy}}{I_{zz}}\omega_x\omega_y + \frac{\tau_z}{I_{zz}} \quad (20)$$

These are called Eulers equations. Notice how the angular velocity may change even when the torque is zero - this is called precession. Although this changes the angular velocity, the angular momentum in world coordinates remains the same.

These equations can be re-written as :

$$\underline{\tau_b} = I_b\underline{\dot{\omega}_b} + \underline{\omega_b} \times I_b\underline{\omega_b} \quad (21)$$

5. Applying force and torque to the body

5.1. Applying force at a point

You will often need to apply force at any point on the body. When a force is applied to any point on the body it is equivalent to a force on the centre of mass and a torque. The force and the point may be specified in world coordinates or body coordinates. The following combinations may be useful :

- force and point in world coords
- force in world coords, point in body coords
- force and point in body coords

For equation 7 we need the force in world coords. For equations 18 – 20 we need the torque in body coords. The following equations can be used in each case.

5.1.1. Force and point in world coords

The force is already in world coords. The torque in body coords is :

$$\begin{aligned} \underline{\tau_w} &= \left(\underline{r_w} - \underline{x}\right) \times \underline{F_w} \\ \underline{\tau_b} &= R^T \cdot \underline{\tau_w} \end{aligned} \quad (22)$$

where :

$\underline{F_w}$	=	force (world coords)
$\underline{r_w}$	=	point of application (world coords)
$\underline{\tau_w}$	=	torque (world coords)
$\underline{\tau_b}$	=	torque (body coords)

5.1.2. Force in world coords, point in body coords

The force is already in world coords. The torque in body coords is :

$$\begin{aligned} \underline{F_b} &= R^T \cdot \underline{F_w} \\ \underline{\tau_b} &= \underline{r_b} \times \underline{F_b} \end{aligned} \tag{23}$$

where :

$\underline{F}_w$	=	force (world coords)
$\underline{F}_b$	=	force (body coords)
$\underline{r}_b$	=	point of application (body coords)
$\underline{\tau}_b$	=	torque (body coords)

5.1.3. Force and point in body coords

The force in world coords is :

$$\underline{F_w} = R \cdot \underline{F_b} \tag{24}$$

where :

$\underline{F}_w$	=	force (world coords)
$\underline{F}_b$	=	force (body coords)

The torque in body coords is :

$$\underline{\tau_b} = \underline{r_b} \times \underline{F_b} \tag{25}$$

where :

$\underline{F}_b$	=	force (body coords)
$\underline{r}_b$	=	point of application (body coords)
$\underline{\tau}_b$	=	torque (body coords)

5.2. Applying torque

You may need to apply torque to a body without applying force. For equations 18 – 20 we need the torque in body coords. If the torque is in world coords the following equation will convert it into body coords :

$$\underline{\tau_b} = R^T \cdot \underline{\tau_w} \tag{26}$$

where :

$\underline{\tau}_w$	=	torque (world coords)
$\underline{\tau}_b$	=	torque (body coords

6. Integrating the state variables

6.1. Summary of state variables

The full set of state variables are :

$\underline{x}$	=	linear position
$\underline{\dot{x}}$	=	linear velocity
Q	=	angular orientation
$\underline{\omega_b}$	=	angular velocity

These can be concatenated into one long state vector $\underline{q}$.

The derivatives are :

$$\frac{d}{dt}\underline{x} = \underline{\dot{x}} \tag{6}$$

$$\frac{d}{dt}\underline{\dot{x}} = \frac{1}{m}\underline{F_w} \tag{7}$$

$$\frac{d}{dt}\begin{bmatrix} Q_w \\ Q_x \\ Q_y \\ Q_z \end{bmatrix} = \frac{1}{2}\begin{bmatrix} -Q_x & -Q_y & -Q_z \\ Q_w & -Q_z & Q_y \\ Q_z & Q_w & -Q_x \\ -Q_y & Q_x & Q_w \end{bmatrix} \cdot \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{14}$$

$$\frac{d}{dt}\omega_x = \frac{I_{yy} - I_{zz}}{I_{xx}}\omega_y\omega_z + \frac{\tau_x}{I_{xx}} \tag{18}$$

$$\frac{d}{dt}\omega_y = \frac{I_{zz} - I_{xx}}{I_{yy}}\omega_z\omega_x + \frac{\tau_y}{I_{yy}} \tag{19}$$

$$\frac{d}{dt}\omega_z = \frac{I_{xx} - I_{yy}}{I_{zz}}\omega_x\omega_y + \frac{\tau_z}{I_{zz}} \tag{20}$$

where :

$\underline{F}_w$	=	applied force on centre of mass (world coords)
$\underline{\tau}_b$	=	torque (body coords)

The derivatives are concatenated into one long vector :

$$\frac{d}{dt}\underline{q} = \underline{\dot{q}} \tag{27}$$

6.2. Integrating with Eulers method

Eulers method is a very simple way to integrate these equations. If you know the state of the body at time t and the applied forces and torques you can find the state at time ($t + h$) as follows :

$$\underline{q}(t+h) = \underline{q}(t) + h\underline{\dot{q}} \tag{28}$$

where :

$\underline{q}(t)$	=	state of body in current step
$\underline{q}(t+h)$	=	state of body in next step
h	=	size of step (seconds)
$\underline{\dot{q}}$	=	derivative at current time. Calculated from current forces and torques.

Although this is very simple to get up and running, it is not a good way to do it. You will need to set the step size very small (between 0.001 and 0.0001) to have good accuracy. This means you evaluate the equations lots of times to advance the simulation by a given time. The Runge-Kutta method in the next section can usually achieve the same accuracy with a much larger step size, hence is much faster to simulate.

6.3. Integrating with 4th order Runge-Kutta method

The 4th order Runge-Kutta method is usually written as :

$$\begin{aligned}
k_1 &= hf(q,t) \\
k_2 &= hf\left(q + \frac{k_1}{2}, t + \frac{h}{2}\right) \\
k_3 &= hf\left(q + \frac{k_2}{2}, t + \frac{h}{2}\right) \\
k_4 &= hf(q + k_3, t + h) \\
q(t+h) &= q + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}
\end{aligned} \tag{29}$$

where :

$f(q,t)$	=	derivative of q at time t

This can be re-written in terms of position, velocity and acceleration. The following variables will be needed to store temporary data :

```
pos, vel                    // current position and velocity
force                       // current force = f(pos, vel)
pos0                        // position at start of step
vel1, acc1                  // position and velocity for intermediate steps
vel2, acc2
vel3, acc3
```

The following steps will apply the Runge-Kutta method to a particle of mass :

```
RKInit :
       pos0 = pos;

RKStep1 :
       acc1 = force * (h / mass / 2.0);
       vel1 = vel;
       pos += vel1 * (h / 2.0);
       vel += acc1;

RKStep2 :
       acc2 = force * (h / mass / 2.0);
       vel2 = vel;
       pos = pos0 + vel2 * (h / 2.0);
       vel = vel1 + acc2;

RKStep3 :
       acc3 = force * (h / mass);
       vel3 = vel;
       pos = pos0 + vel3 * h;
       vel = vel1 + acc3;

RKStep4 :
       acc4 = force * (h / mass / 2.0);
       pos = pos0 + (vel1 + 2.0 * (vel2 + vel3) + vel) * (h / 6.0);
       vel = vel1 + (acc1 + 2.0 * acc2 + acc3 + acc4) / 3.0;
```

The numbers in brackets, such as `(h / mass / 2.0)` are constants and can be precalculated for extra speed.

Between each step the force is calculated from the current position and velocity. The whole process works as follows :

```
< state at time t : position = pos, velocity = vel >
RKInit();
force = f(pos, vel);
RKStep1();
force = f(pos, vel);
RKStep2();
force = f(pos, vel);
RKStep3();
force = f(pos, vel);
RKStep4();
< state at time t + h : position = pos, velocity = vel >
```

It looks more complicated than this for a rigid body because the rotational position and velocity are also required. However, the same idea can be used.

7. Some other useful equations for rigid bodies

7.1. Position of a point

The position of a point can be found in world coords as follows :

$$\underline{r_w} = R \cdot \underline{r_b} + \underline{x} \tag{30}$$

where :

$\underline{r_b}$	=	point on body (body coords)
$\underline{r_w}$	=	position of point (world coords)

7.2. Velocity of a point

The velocity of a point can be found in world coords as follows :

$$\underline{v_w} = R\left(\underline{\omega_b} \times \underline{r_b}\right) + \underline{\dot{x}} \tag{31}$$

where :

$\underline{r_b}$	=	point on body (body coords)
$\underline{v_w}$	=	velocity of point (world coords)

7.3. Angular momentum

The angular momentum in world coords can be found as follows :

$$\underline{L_w} = R \cdot I_b \cdot \underline{\omega_b} \tag{32}$$

This should remain constant assuming there is no drag or other forces. This can be useful to check when testing your code.

7.4. Energy

A rigid body has three types of energy :

- gravitational energy
- linear kinetic energy
- rotational kinetic energy

The total energy is as follows :

$$E = mgz + \frac{m\left(\underline{\dot{x}} \cdot \underline{\dot{x}}\right)}{2} + \frac{\left(I_b \underline{\omega_b}\right) \cdot \underline{\omega_b}}{2} \tag{33}$$

This should also remain constant assuming there is no drag or other forces.

8. The RigidBody class

8.1. Introduction

I have written a C++ class called RigidBody which does everything in this document and is available for free. I have tried to make the class fast and easy to use.

To compile it you need the following files :

- RigidBody.cpp
- RigidBody.h
- Matrix.cpp
- Matrix.h

8.2. Creating a rigid body

The following steps are needed :

- initialise the stepsize and gravity (same for all rigid bodies)
- declare a RigidBody instance
- set up its mass distribution
- set up the initial position and velocity

The following code gives an example of this :

```
float h = 0.01;                          // step size = 0.01 secs
float g = 9.81;                          // gravity = 9.81 m/s/s
float linKD = 0.01;                      // linear damping
float rotKD = 0.01;                      // rotational damping

RigidBody::setStatics(h, g);             // set up step size and g for all
                                         // rigid bodies

RigidBody mainBody, tempBody;            // declare two bodies. We will combine
                                         // the mass distributions into one
                                         // as an example.

mainBody.setBox(4.0, 2.0, 1.0, 1000.0); // set up a box :
                                         //     x radius = 4.0 metres
                                         //     y radius = 2.0 metres
                                         //     z radius = 1.0 metres
                                         //     mass = 1000.0 kg

tempBody.setCylinder(1.0, 2.0, 500.0);  // set up a cylinder (along x axis) :
                                         //     radius = 1.0 metres
                                         //     height = 2.0 metres
                                         //     mass = 500.0 kg

Vector3 tempLinPos(5.0, 0.0, 0.0);       // position and orientation of cylinder
Matrix3x3 tempRotPos;                    // relative to main body.
tempRotPos.identity();

mainBody.combine(tempBody, tempRotPos, tempLinPos);
                                         // merge the cylinder into the main body
                                         // to create one body

mainBody.diagonalise();                  // diagonalise its moment of inertia and
                                         // move the centre of mass to origin
```

(Note : Don't forget to transform the graphical representation by :)
(new coord = mainBody.diagRotPos * old coord + mainBody.diagLinPos)

```
Vector3 linPos(0.0, -10.0, 10.0);       // initial position
Vector3 linVel(0.0, 0.0, 0.0);          // initial velocity
Quaternion qRotPos;                     // initial rotation. Set up the
Vector3 rotAxis(1.0, 1.0, 0.0);         // rotation as 0.3 radians around the
qRotPos.rotation(rotAxis, 0.3);         // axis (1.0, 1.0, 0.0).
Vector3 rotVel(1.0, 2.0, -0.5);         // initial angular velocity

mainBody.initialise(linPos, linVel, qRotPos, rotVel, linKD, rotKD);
                                        // initialise state of body
```

This code creates two rigid bodies, a box and a cylinder, and merges them into one. Currently there are only three primitive mass distributions :

- box
- cylinder
- sphere

The `combine()` function merges them together with the second body at a given position and rotation from the first one. You can combine as many bodies as you like into one.

You can also set up your own mass distribution directly if you know the centre of mass and moment of inertia matrix.

The body is then diagonalised and the state is initialised. You will have to transform the graphical representation for the body to diagonalised coordinates. You can use `diagRotPos` and `diagLinPos` for this. The current state of the body holds the position and orientation for diagonalised coordinates.

The `initialise()` function uses a quaternion for the initial orientation. This example sets up the quaternion as a rotation around an axis. You could also set the quaternion from a matrix using :

```
qRotPos.fromMatrix(myMatrix);
```

8.3. Evolving the state of the body

To evolve the state of the body you must do the following :

- evolve it with the Runge-Kutta method.
- calculate forces and torques on the body at each of the four steps.
- draw the graphics at `linPos` and `rotPos`. You can take several Runge-Kutta steps between drawing the graphics if you want.

The following code gives an example of this :

```
float time = 0.0;
const int numSteps = 5;                 // take 5 steps between drawing graphics
while (running) {
       for (int i = numSteps; i--;) {
              mainBody.RKInit();        // start Runge-Kutta
              forces();                 // calculate forces
              mainBody.RKStep1();
              forces();
              mainBody.RKStep2();
              forces();
              mainBody.RKStep3();
              forces();
              mainBody.RKStep4();
              time += RigidBody::RKh;
       }

       drawBody(mainBody.linPos, mainBody.rotPos);
                                        // draw graphic at current position
                                        // with your own code.
}
```

The `forces()` function must calculate forces and torques on the body. Each time you call it you must add forces and torques dependent on the following variables :

- Linear position = `linPos`
- Linear velocity = `linVel`
- Orientation = `rotPos`
- Angular velocity = `rotVel`

The Runge-Kutta integrator functions will automatically set up these variables before each step so you can calculate the forces with the same function every time. You can add the forces directly to the `linForce` and `rotForce` variables or use the provided functions for adding forces and torques. There are also some functions provided for calculating position and velocity of a point on the body.

Note : You do not need to add gravity and damping forces. They are automatically included using the values for `g`, `linKD` and `rotKD`. These automatic damping forces are intended for increasing the stability of the system. You can add extra drag if you want.