Dynamic Physics for Simulation and Game Programming

Mike Bailey
mjb@cs.oregonstate.edu

Discrete Dynamics

2. Force equals mass times acceleration (F=ma)

\[ a = \frac{F}{m} \]

\[ \frac{\Delta v}{\Delta t} = a \rightarrow \Delta v = a \Delta t \rightarrow v = \sum \Delta v = \sum a \Delta t \]

\[ \frac{\Delta x}{\Delta t} = v \rightarrow \Delta x = v \Delta t \rightarrow x = \sum \Delta x = \sum v \Delta t \]

These can be thought of as summing areas under curves.

Integrating the Physics Equations if Acceleration is Constant

What if Acceleration is not Constant?

What is \( v(t+\Delta t) \) ?

The problem is that we are treating all of the quantities as if they always have the value that they had at the start of the Time Step, even though they don’t.

This is known as a First Order solution.
What does a First Order Solution look like in a Program?

You need a way to hold the entire state of the system. This will be the input to our numerical integrator. You also need a way to return the derivatives once you determine them.

```
struct state State;
```

The inputs are the state, which consists of all variables necessary to completely describe the state of the physical system. The outputs are the derivatives of each state variable.

```
struct derivatives Derivatives;
```

```
void GetDerivs(State, Derivatives)
{
  . . .
}
```

The outputs are the derivatives of the state variables.

```
struct state {
  float time;
  float x;
  float vx;
};
```

```
struct derivatives {
  float vx;
  float ax;
};
```

The application, then, consists of:

```
Initialize();
AdvanceOneTimeStep();
Finish();
```

What does a Second Order Solution look like in a Program?

```
void AdvanceOneTimeStep()
{
  GetDerivs(State, Derivatives1);
  State2.t  = State.t + Δt;
  State2.x  = State.x + Derivatives1.vx * Δt;
  State2.vx = State.vx + Derivatives1.ax * Δt;
  GetDerivs(State2, Derivatives2);
  float aavg = (Derivatives1.ax + Derivatives2.ax)/2.;
  float vavg = (Derivatives1.vx + Derivatives2.vx)/2.;
  State.x = State.x + vavg * Δt;
  State.vx = State.vx + aavg * Δt;
  State.t = State.t + Δt;
}
```

The Runge-Kutta Fourth Order Solution

```
void AdvanceOneTimeStep()
{
  GetDerivs(State, Derivatives1);
  State2.t  = State.t + Δt/2;
  State2.x  = State.x +  Derivatives1.vx * (Δt/2.);
  State2.vx = State.vx + Derivatives1.ax * (Δt/2.);
  GetDerivs(State2, Derivatives2);
  State3.t  = State.t + Δt/2;
  State3.x  = State.x +  Derivatives2.vx * (Δt/2);
  State3.vx = State.vx + Derivatives2.ax * (Δt/2);
  GetDerivs(State3, Derivatives3);
  State4.t  = State.t + Δt;
  State4.x  = State.x +  Derivatives3.vx * Δt;
  State4.vx = State.vx + Derivatives3.ax * Δt;
  GetDerivs(State4, Derivatives4);
  State.x = State.x +  (Δt/6.) * ( Derivatives1.vx + 2.*Derivatives2.vx + 2.*Derivatives3.vx + Derivatives4.vx );
  State.vx = State.vx +  (Δt/6.) * (Derivatives1.ax + 2.*Derivatives2.ax + 2.*Derivatives3.ax + Derivatives4.ax );
}```
Solving Motion where there is a Spring

\[ F_{spring} = -ky \]  
This is known as Hooke's law

\[ \Delta v = \sum \frac{F}{m} \Delta t = \frac{-W - ky}{m} \Delta t \]

void GetDerivs( State, Derivatives )
{
  Derivatives.vy = State.vy;
  Derivatives.ay = (-W - K*State.y) / MASS;
}

Some Drag Coefficients

<table>
<thead>
<tr>
<th>C_d</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>a smooth brick</td>
</tr>
<tr>
<td>0.9</td>
<td>a typical bicycle plus cyclist</td>
</tr>
<tr>
<td>0.4</td>
<td>smooth sphere</td>
</tr>
<tr>
<td>0.1</td>
<td>laminar flat plate</td>
</tr>
<tr>
<td>0.005</td>
<td>turbulent flat plate</td>
</tr>
<tr>
<td>0.295</td>
<td>bullet</td>
</tr>
<tr>
<td>1.0-1.3</td>
<td>person (upright position)</td>
</tr>
<tr>
<td>1.28</td>
<td>flat plate perpendicular to flow</td>
</tr>
<tr>
<td>1.0-1.1</td>
<td>skier</td>
</tr>
<tr>
<td>1.0-1.3</td>
<td>wires and cables</td>
</tr>
<tr>
<td>1.3-1.5</td>
<td>Empire State Building</td>
</tr>
<tr>
<td>1.9-2.0</td>
<td>ColFront Tower</td>
</tr>
</tbody>
</table>

http://en.wikipedia.org/wiki/Drag_coefficient

Solving Motion where there is Air Resistance

\[ \Delta v = \sum \frac{F}{m} \Delta t = \frac{-W - \text{Sign}(v) \frac{1}{2} \rho v^2 A C_d}{m} \Delta t \]

void GetDerivs( State, Derivatives )
{
  Derivatives.vy = State.vy;
  Derivatives.ay = (-(W - .5*Sign(State.vy)*DENSITY* State.vy * State.vy * AREA*DRAG) / MASS);
}

The \text{Sign}( ) function returns +1. or -1., depending on the sign of argument.

Terminal Velocity

When a body is in free fall, it is being accelerated by the force of gravity. However, as it accelerates, it is encountering more and more air resistance force. At some velocity, these two forces balance each other out and the velocity becomes constant, that is, \( \Delta v = 0 \).

This is known as the terminal velocity.

\[ \Delta v = \frac{-W + \frac{1}{2} \rho v^2 A C_d}{m} \]

The velocity becomes constant when \( \Delta v = 0 \):

\[ W - \frac{1}{2} \rho v^2 A C_d = 0 \]

\[ v_t = \sqrt{\frac{2W}{\rho A C_d}} \]

Human Terminal Velocity

Assume:

Weight = 200 pounds = 890 Newtons
\( C_d = 1.28 \)
\( A = 6 \) ft\(^2\) = 0.558 m\(^2\)
\( \rho = 1.293 \) kg/m\(^3\)

\[ v_t = \sqrt{\frac{2W}{\rho A C_d}} = 43.30 \text{ m/sec} = 98 \text{ mph} \]
How about a Cliff Jumper on a Bungee Cord?

\[ F_{\text{spring}} = -k y \]

\[ F_{\text{drag}} = -\text{Sign}(v) \frac{1}{2} \rho v^2 C_D A \]

\[ \Delta v = \frac{\sum F}{m} \Delta t = \frac{-W - F_{\text{drag}} - F_{\text{res}}} {m} \]

Coulomb Damping

This is very much like drag force, but it is the resistance of a fluid being squeezed through a small opening. The resisting force is proportional to the velocity.

\[ F_{\text{damping}} = -c v \]

Lift – Another Good Force to Know About

\[ F_{\text{lift}} = \frac{1}{2} \rho v^2 A C_L \]

Coefficient of Lift vs. Angle of Attack

Friction Force – Another Good Force to Know About

\[ F_{\text{friction}} = \mu N \]

Note: "\( \mu \)" is the coefficient of friction, \( N \) is the normal force (i.e., amount of force that is perpendicular to the surface).
### Some Coefficients of Friction

<table>
<thead>
<tr>
<th>Materials</th>
<th>μ Dry &amp; clean</th>
<th>μ Lubricated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aluminum</td>
<td>0.61</td>
<td></td>
</tr>
<tr>
<td>Copper</td>
<td>0.51</td>
<td></td>
</tr>
<tr>
<td>Brass</td>
<td>0.51</td>
<td></td>
</tr>
<tr>
<td>Cast iron</td>
<td>0.65</td>
<td></td>
</tr>
<tr>
<td>Cast iron</td>
<td>0.65</td>
<td></td>
</tr>
<tr>
<td>Concrete</td>
<td>0.38</td>
<td></td>
</tr>
<tr>
<td>Concrete (wet)</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>Concrete (dry)</td>
<td>0.62</td>
<td></td>
</tr>
<tr>
<td>Copper</td>
<td>0.64</td>
<td></td>
</tr>
<tr>
<td>Glass</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>Metal</td>
<td>0.2 - 0.6</td>
<td>0.2 (wet)</td>
</tr>
<tr>
<td>Polyethylene</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>Steel</td>
<td>0.68</td>
<td>0.16</td>
</tr>
<tr>
<td>Steel</td>
<td>0.68</td>
<td>0.16</td>
</tr>
<tr>
<td>Teflon</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>Teflon</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>Wood</td>
<td>0.35 - 0.5</td>
<td>0.2 (wet)</td>
</tr>
</tbody>
</table>

### Buoyancy – Another Good Force to Know About

Archimedes’ Principle says that the buoyancy force on an object in a fluid is the weight of the fluid that is being displaced by the object.

\[
F_{\text{buoyancy}} = \rho_{\text{fluid}} V_{\text{liquid}} \Delta \rho_{\text{fluid}} = \rho_{\text{fluid}} \left( \rho_{\text{fluid}} - \rho_{\text{object}} \right)
\]

So, for a helium balloon that is one foot in diameter (i.e., radius=6 inches), it has its weight pulling it down and a buoyancy force pushing it up. The net force pushing it up because of the gas inside the balloon is:

\[
F_{\text{buoyancy}} = \frac{4}{3} \pi r^3 \rho_{\text{helium}} = 904.78 \text{ lb}
\]

\[
F_{\text{buoyancy}} = 904.78 \text{ lb} \left( 4.01 \times 10^{-4} \text{ lb/in}^3 \right) = 0.036 \text{ lb}
\]

Note that this must still counterbalance the weight of the balloon material, or the balloon will not fly.

### Spinning Motion: Dynamics

\[
T = I \alpha
\]

\[
\alpha = \frac{T}{I}
\]

\[
\Delta \omega = \frac{T}{I} \Delta t
\]

\[
\text{Torque} = \text{an angular "force" (newton-meters)}
\]

\[
\text{Moment of Inertia} = \text{an angular "mass" (newton-meters-sec}^2\text{=kg-meters}^3)\]

### What does this look like in a Program?

```c
struct state
{
    float t;
    float x;
    float vx;
    float theta;
    float omega;
};

struct derivatives
{
    float vx;
    float ax;
    float omega;
    float alpha;
};
```

The state and derivative vectors now include angular components.