Dynamic Physics for Simulation and Game Programming

Mike Bailey
mjb@cs.oregonstate.edu

Discrete Dynamics

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

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

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

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

These can be thought of as summing areas under curves
Integrating the Physics Equations if Acceleration is Constant

\[ \Delta v = a \Delta t = \text{area} \]

What if Acceleration is not Constant?

\[ \Delta v = a \Delta t = \text{area} \]
What is $v(t+\Delta t)$?

$$v(t+\Delta t) = v(t) + \Delta v = v(t) + a \Delta t$$

This is close, but clearly not exactly right!

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.

```c
struct state state;
struct derivatives derivs;
void GetDerivs( state, &derivs )
{
    . . .
}
```

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.

```c
struct state state;  // state
struct derivatives derivs;  // derivs
void GetDerivs( state, &derivs )
{
    . . .
}
```

The application, then, consists of:

```c
void AdvanceOneTimeStep( )
{
    GetDerivs( state, &derivs );  // get derivatives

    state.x = state.x + derivs->vx * Δt;  // use derivatives
    state.vx = state.vx + derivs->ax * Δt;  // use derivatives
    state.t = state.t + Δt;
}
```

```c
void AdvanceOneTimeStep( )
{
    GetDerivs( state, &derivs );  // get derivatives

    state.x = state.x + derivs->vx * Δt;  // use derivatives
    state.vx = state.vx + derivs->ax * Δt;  // use derivatives
    state.t = state.t + Δt;
}
```

The application, then, consists of:

```c
Initialize( );
AdvanceOneTimeStep( );
Finish( );
```
What does a GetDerivs function look like in a Program?

```c
void GetDerivs( State state, Derivatives *derivs )
{
    float sumfy = -Weight; // amount the spring is stretched:
    float ym = Y0 - state.y;
    float stretch = ym - LENGTH0;
    sumfy += K * stretch;
    sumfy -= Cd * state.vy; // the damping
    derivs->vy = state.vy;
    derivs->ay = sumfy / Mass;
}
```

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

A Second Order solution is obtained by doing the First Order solution, determining all quantities at time $t + \Delta t$ then averaging them with the quantities at time $t$ and then treating them as constant throughout the interval.

1. $a(t) = \frac{F(t)}{m}$
2. $a(t + \Delta t) = \frac{F(t + \Delta t)}{m}$
3. $\Delta v = a \Delta t = a_{avg} \Delta t = \frac{a(t) + a(t + \Delta t)}{2} \Delta t$
4. $v(t + \Delta t) = v(t) + \Delta v$
What does a Second Order Solution look like in a Program?

```c
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 application, then, consists of:
- Initialize();
- AdvanceOneTimeStep();
- Finish();

The Runge-Kutta Fourth Order Solution

\[
\begin{align*}
\{v_1\} &= GetDerivs(t, x, v) \\
\{a_1\} &= GetDerivs(t + \frac{Δt}{2}, x + v_1 \frac{Δt}{2}, v + a_1 \frac{Δt}{2}) \\
\{v_2\} &= GetDerivs(t + \frac{Δt}{2}, x + v_2 \frac{Δt}{2}, v + a_2 \frac{Δt}{2}) \\
\{a_2\} &= GetDerivs(t + Δt, x + v_2 Δt, v + a_2 Δt) \\
\{v_3\} &= GetDerivs(t + Δt, x + v_3 Δt, v + a_3 Δt) \\
\{a_3\} &= GetDerivs(t + Δt, x + v_3 Δt, v + a_3 Δt) \\
\{v_4\} &= GetDerivs(t + Δt, x + v_4 Δt, v + a_4 Δt)
\end{align*}
\]

\[
\begin{align*}
\{x(t + Δt)\} &= \left\{x\right\} + \frac{Δt}{6} \left( \left\{v_1\right\} + 2\left\{v_2\right\} + 2\left\{v_3\right\} + \left\{v_4\right\} \right)
\end{align*}
\]

Adapted from: http://en.wikipedia.org/wiki/Runge-Kutta
### The Runge-Kutta Fourth Order Solution

```c
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

This is known as Hooke’s law

\[
F_{spring} = -ky
\]

\[
Δv = \frac{\sum F}{m} Δt = \frac{-W - ky}{m} Δt
\]

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

For clarity, I’ve left out the “- LENGTH0”, but it should be there
Air Resistance Force

\[ F_{\text{drag}} = \frac{1}{2} \rho v^2 A c_d \]

- \( F_{\text{drag}} \): Air Resistance Force
- \( \rho \): Fluid density
- \( v \): Y Velocity
- \( A \): Cross-sectional area
- \( c_d \): Drag Coefficient

Air Resistance always acts in a direction opposite to the velocity of the object.

Some Drag Coefficients

<table>
<thead>
<tr>
<th>( c_d )</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</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>rough sphere</td>
</tr>
<tr>
<td>0.1</td>
<td>smooth sphere</td>
</tr>
<tr>
<td>0.001</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>1.8-2.0</td>
<td>Eiffel Tower</td>
</tr>
</tbody>
</table>

http://en.wikipedia.org/wiki/Drag_coefficient
Solving Motion where there is Air Resistance

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

\[ \rho_{air} = 1.293 \, \text{kg/m}^3 \]

The `Sign()` function returns +1. or -1., depending on the sign of argument

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

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**.

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

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

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

\[ v_y = \sqrt{\frac{2W}{\rho A C_d}} \]
Human Terminal Velocity

Assume:

Weight = 200 pounds = 890 Newtons
\(C_d = 1.28\)
\(A = 6 \text{ ft}^2 = 0.558 \text{ m}^2\)
\(\rho_{\text{air}} = 1.293 \frac{\text{kg}}{\text{m}^3}\)

\[v_t = \sqrt{\frac{2W}{\rho AC_d}} = 43.90 \frac{m}{\text{sec}} \approx 98 \text{mph}\]

How about a Cliff Jumper on a Bungee Cord?

\[F_{\text{spring}} = -ky\]
\[F_{\text{drag}} = -\text{Sign}(v_y) \frac{1}{2} \rho v_y^2 C_d A\]
\[\Delta v = \sum_{m} \frac{F}{\Delta t} = \frac{-W - ky - \text{Sign}(v_y) \frac{1}{2} \rho v_y^2 AC_d}{m} \Delta t\]

```c
void GetDerivs( State state, Derivatives *derivs )
{
    derivs->vy = state.vy;
    derivs->ay = ( -W - K*state.y .5*Sign(state.vy)*DENSITY*state.vy*state.vy*AREA*DRAG ) / MASS;
}
```
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:

\[ F_{damping} = -cv \]

Lift – Another Good Force to Know About

\[ F_{lift} = \frac{1}{2} \rho v^2 AC_L \]

Co-efficient of Lift, for a given angle of attack

http://en.wikipedia.org/wiki/Lift_%28force%29
**Coefficient of Lift vs. Angle of Attack**

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

**Lift and Drag Dramatically Working Together – Flight of a Frisbee**
Friction Force – Another Good Force to Know About

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

- Normal force (i.e., amount of force that is perpendicular to the surface)
- Coefficient of Friction

\[ N = W \cos \theta \]

\[ \text{Sliding Force} = W \sin \theta \]

Some Coefficients of Friction

<table>
<thead>
<tr>
<th>Materials</th>
<th>Dry &amp; clean</th>
<th>Lubricated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aluminum Steel</td>
<td>0.61</td>
<td></td>
</tr>
<tr>
<td>Copper Steel</td>
<td>0.53</td>
<td></td>
</tr>
<tr>
<td>Brass Steel</td>
<td>0.51</td>
<td></td>
</tr>
<tr>
<td>Cast iron Copper</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>Cast iron Zinc</td>
<td>0.85</td>
<td></td>
</tr>
<tr>
<td>Concrete (wet) Rubber</td>
<td>0.30</td>
<td></td>
</tr>
<tr>
<td>Concrete (dry) Rubber</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>Concrete Wood</td>
<td>0.62</td>
<td></td>
</tr>
<tr>
<td>Copper Glass</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>Glass Glass</td>
<td>0.94</td>
<td></td>
</tr>
<tr>
<td>Metal Wood</td>
<td>0.2–0.6</td>
<td>0.2 (wet)</td>
</tr>
<tr>
<td>Polythene Steel</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>Steel Steel</td>
<td>0.80</td>
<td>0.16</td>
</tr>
<tr>
<td>Steel Teflon</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Teflon Teflon</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>Wood Wood</td>
<td>0.25–0.5</td>
<td>0.2 (wet)</td>
</tr>
</tbody>
</table>

http://en.wikipedia.org/wiki/Friction
Determining the Coefficient of Friction Experimentally

At what angle, Θ, does the block just begin to slide? It begins to slide when the sliding force equals the friction force:

\[ W \sin \theta = \mu W \cos \theta \]

Thus, if you raise the board just enough that the block starts to slide, the coefficient of friction is the tangent of that angle.

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.

So, for a helium balloon, it has its weight pulling it down, the weight of the helium pulling it down, and a buoyancy force pushing it up.
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:

$$F = \rho_{\text{helium}} V - \rho_{\text{air}} V = \rho_{\text{helium}} V (\rho_{\text{helium}} - \rho_{\text{air}})$$

$$V = \frac{4}{3} \pi r^3 = 904.78 \text{ in}^3$$

$$F_{\text{buoyancy total}} = 904.78 \text{ in}^3 (4.01 \times 10^{-2} \text{ pounds/in}^3) = 0.036 \text{ pounds}$$

Note that we would still need to take the balloon’s skin weight into account.

You have a waterproof box that you lay on the surface of the water. You put a 100-pound dog in the box. How deep will the box sink into the water before it starts to float?

(Let’s assume the weight of the box and the weight of the air in the box are negligible.)

$$\rho_{\text{water}} = 0.03635 \text{ pounds/in}^3$$

The water displaced is Length*Width*D, so the upwards force is 0.03635*Length*Width*D which needs to counterbalance 100 pounds. So, we find that the box sinks to a depth of:

$$D = \frac{2751}{\text{Length} \times \text{Width}}$$

If D ends up exceeding the height of the box, well, I sure hope the dog can swim.
Spinning Motion: Dynamics

\[ T = I \alpha \]

Moment of Inertia = an angular "mass"
(newton-meters-sec\(^2\)/kg-meters\(^2\))

Torque = an angular "force"
(newton-meters)

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

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

What does this look like in a Program?

```c
void GetDerivs( State state, Derivatives *derivs )
{
    derivs->vx = state.vx;
    derivs->x   = SomeOfAllForces / MASS;
    derivs->omega = state.omega;
    derivs->alpha    = SomeOfAllTorques / INERTIA
}
```

The state and derivative vectors now include angular components.