Integration, rotation matrices, and quaternions

Dr. Giuseppe Maggiore

NHTV University of Applied Sciences
Breda, Netherlands

Table of contents

Integration and rotation

Integration and rotation

- We integrate L from τ
- We integrate w from L and J^-1
- We integrate R from w

Integration and rotation

Issues

- *Issue 1:* recomputing J from every particle of the body and at every tick is too slow
- *Issue 2:* integrating R slowly "breaks it", meaning that $[T \;\; N \;\; B]$ is not a rotation anymore

Caching J

Issue 1: body shape

- J represents the tendency of the body to resist rotation
- But the body just moves and rotates, it does not change shape
- We should be able to simply recompute J from its initial value and the rotation

Caching J

J from J_{body}

$$
\begin{aligned}
J &= \sum_i (|r_i(t)|^2 I - r_i(t) r_i^T(t)) && (1)\\
&= \sum_i (|R r_i(t_0)|^2 I - R r_i(t_0) r_i^T(t_0) R^T) && (2)\\
&= \sum_i (|r_i(t_0)|^2 I - R r_i(t_0) r_i^T(t_0) R^T) && (3)\\
&= \sum_i (|r_i(t_0)|^2 R R^T - R r_i(t_0) r_i^T(t_0) R^T) && (4)\\
&= \sum_i (R |r_i(t_0)|^2 R^T - R r_i(t_0) r_i^T(t_0) R^T) && (5)
\end{aligned}
$$

Caching J

J from J_{body}

$$
\begin{aligned}
J &= \sum_i (R|r_i(t_0)|^2 R^T - R r_i(t_0) r_i^T(t_0) R^T) && (6)\\
&= \sum_i R(|r_i(t_0)|^2 - r_i(t_0) r_i^T(t_0)) R^T && (7)\\
&= R\left(\sum_i (|r_i(t_0)|^2 - r_i(t_0) r_i^T(t_0))\right) R^T && (8)\\
&= R J_{body} R^T && (9)
\end{aligned}
$$

Caching J

J^{-1} from $J_{body}-1$

- We need J^{-1} more than J
- We can compute it without an inversion though
- We compute (just once) J_{body}^{-1}
- $J^{-1}(RJ_{body}R^T)^{-1} = RJ_{body}^{-1}R^T$

Orthonormalization of R

Issue 2: R is a rotation matrix

- $R = [T \;\; N \;\; B]$
- The vectors T, N, and B must be of unit length and orthonormal ($T \cdot N = 0, T \cdot B = 0, \dots$)
- As we integrate R, this stops being true
- R stops being just a rotation matrix, and also incorporates some scale and some skew
- This is very bad!

Boxes stop being boxes

Orthonormalization of R

Gram-Schmidt method

- $\hat{R} = [\hat{T}\ \hat{N}\ \hat{B}]$
- We normalize $T = \frac{\hat{T}}{|\hat{T}|}$
- We remove the projection of $\hat{N}$ over T: $\hat{N}' = \hat{N} - \hat{N}(\hat{N} \cdot T)$
- We normalize $N = \frac{\hat{N}'}{|\hat{N}'|}$
- We compute $B = T \times N$

Orthonormalization of R

Gram-Schimdt method

- Lot of useless computation
- Matrix representation very redundant
- $\hat{B}$ is completely unneeded and is ignored!

Using quaternions

About quaternions

- We just need a single rotation around an axis (which is an arbitrary rotation in 3D)
- Less degrees of freedom means less drift and less need for normalization

Using quaternions

Quaternion primer

- Let us consider a rotation matrix around the Z axis
- $R_0 = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Using quaternions

Quaternion primer

- Any vector v on the XY plane rotated by R_0 remains on XY, rotated around Z by an angle θ
- Any vector v on the Z axis rotated by R_0 remains (identical) on Z

Using quaternions

Quaternion primer

- Let us transform this rotation into an arbitrary rotation R_1 around an axis d and of angle θ
- We consider an *orthonormal basis* a, b, d
- We perform a *basis change* of R_0 onto a, b, d

Using quaternions

Quaternion primer

- We need to build R_1 such that
  - $R_1 a = ca + sb$
  - $R_1 b = -sa + cb$
  - $R_1 d = d$
- That is $R_1 \underbrace{[a \; b \; d]}_{P} = \underbrace{[a \; b \; d]}_{P} R_0$
- $P^{-1} = P^T$, so
  $R_1 = PR_0P^T = c(aa^T + bb^T) + s(ba^T - ab^T) + dd^T$

Using quaternions

Quaternion primer

- $R_1 = c(aa^T + bb^T) + s(ba^T - ab^T) + dd^T$
- This means that we can now construct a rotation matrix from an axis and an angle
- Unfortunately we also need two "dummy" vectors a and b
- Any pair of those suffices, as long as they are correctly related to d
- We now remove the explicit dependency on them

Using quaternions

Quaternion primer

- Consider a vector expressed in relationship to a, b, d:
  $v = \underbrace{\alpha}_{a \cdot v} a + \underbrace{\beta}_{b \cdot v} b + \underbrace{\delta}_{d \cdot v} d$
- We now consider two arbitrary but useful quantities:
- $d \times v = d \times (\alpha a + \beta b + \delta d) = \alpha d \times a + \beta d \times b + \delta d \times d = -\beta a + \alpha b = Dv = \begin{bmatrix} 0 & -d_z & d_y \\ d_z & 0 & -d_x \\ -d_y & d_x & 0 \end{bmatrix} v$
- $d \times (d \times v) = d \times (-\beta a + \alpha b) = -\alpha a - \beta b = D^2 v$

Using quaternions

Quaternion primer

- Now let us study the progression of Iv, Dv, D^v
- $Iv = v = \alpha a + \beta b + \delta d = aa^T v + bb^T v + dd^T v$, so $I = aa^T + bb^T + dd^T$
- $Dv = \alpha b - \beta a = ba^T v - ab^T v$, so $D = ba^T - ab^T$
- $D^2 v = -\alpha a - \beta b = \delta d - v = dd^T v - v$, so $D^2 = dd^T - I$

Using quaternions

Quaternion primer

- We combine those three into $R_1 = c(aa^T + bb^T) + s(ba^T - ab^T) + dd^T$
- $I = aa^T + bb^T + dd^T$, $D = ba^T - ab^T$, $D^2 = dd^T - I$

Using quaternions

Quaternion primer

- We combine those three into $R_1 = c(aa^T + bb^T) + s(ba^T - ab^T) + dd^T$
- $I = aa^T + bb^T + dd^T$, $D = ba^T - ab^T$, $D^2 = dd^T - I$
- $R_1 = c(I - dd^T) + sD + dd^T$
- $R_1 = I + sD + (1 - c)D^2$

Using quaternions

Quaternion primer

- We can now compute $R_1 = I + sD + (1-c)D^2$ from just d, θ
- Moreover, we can rotate a vector without even building R_1

$$\begin{aligned} R_1 v &= (I + sD + (1-c)D^2)v && (10) \\ &= Iv + sDv + (1-c)D^2 v && (11) \\ &= v + sd \times v + (1-c)d \times (d \times v) && (12) \end{aligned}$$

Using quaternions

Quaternion primer

- Any representation of d and θ would be fine
- A particularly convenient one is $[\cos\frac{\theta}{2}, \sin\frac{\theta}{2} d]$

Deriving quaternions

Derivative of rotation matrix

- We studied how to compute $\dot{R}$ from R and ω
- If we represent rotations with quaternions, we need to compute $\dot{q}$

Deriving quaternions

Derivative of quaternion

- Given the current orientation $q(t_0)$ and angular velocity $\omega(t_0)$
- Assuming very small $t - t_0$
- The derivative of the original quaternion is

$$\frac{d}{dt}\underbrace{\left[\cos\frac{|\omega(t_0)|(t-t_0)}{2}, \frac{\omega(t_0)}{|\omega(t_0)|}\sin\frac{|\omega(t_0)|(t-t_0)}{2}\right]}_{\omega \text{ converted to a quaternion}} q(t_0)$$

Deriving quaternions

Derivative of quaternion

- $\frac{d}{dt}[\cos\frac{|w(t_0)|(t-t_0)}{2}, \frac{w(t_0)}{|w(t_0)|}\sin\frac{|w(t_0)|(t-t_0)}{2}]q(t_0)$
- We now derive the w part ($q(t_0)$ is a constant) and replace t with t_0 because we want to know the value of the derivative at the current time t_0
- $[-\frac{|\omega(t_0)|}{2}\sin\frac{|\omega(t_0)|(t_0-t_0)}{2}, \frac{\omega(t_0)}{|\omega(t_0)|}\frac{|\omega(t_0)|}{2}\cos\frac{|\omega(t_0)|(t-t_0)}{2}]q(t_0)$
- $[-\frac{|\omega(t_0)|}{2}\sin 0, \frac{\omega(t_0)}{|\omega(t_0)|}\frac{|\omega(t_0)|}{2}\cos 0]q(t_0)$
- $[0, \frac{\omega(t_0)}{2}]q(t_0) = \frac{1}{2}[0, \omega(t_0)]q(t_0)$

Deriving quaternions

Derivative of quaternion

- At least remember this:
- $\dot{q} = \frac{1}{2}[0, \omega]q$

Kinematics source code

```
// initialization
RigidBody* body[n];
double t = <your choice of initial time>;
double dt = <your choice of time step>;
for (int i = 0; i < n; ++i) {
  // Set the initial state of the rigid bodies.
  body[i] = new RigidBody(...);

// Part of the physics tick.
for (i = 0; i < n; ++i) {
  body[i].Update(t, dt);
  t += dt;
}
```

Kinematics source code

```
struct RigidBody {
  RigidBody (double m, matrix inertia, Function force,
       Function torque);

  // force/torque function format
  typedef vector ( *Function ) (
    double, // time of application
    point, // position
    quaternion, // orientation
    ... // whole state of body, one var at a time
    )
  );

  // Runge-Kutta fourth order differential equation
      solver
  void Update (double t, double dt);
```

Kinematics source code

```
protected:
  // convert (Q,P,L) to (R,V,W)
  void Convert (quaternion Q, vector P, vector L,
     matrix& R, vector& V, vector& W) const;
  // constant quantities
  double m_mass, m_invMass;
  matrix m_inertia, m_invInertia;
  // state variables
  vector m_X; // position
  quaternion m_Q; // orientation
  vector m_P; // linear momentum
  vector m_L; // angular momentum
```

Kinematics source code

```
  // derived state variables
  matrix m_R; // orientation matrix
  vector m_V; // linear velocity vector
  m_W; // angular velocity
  // force and torque functions
  Function m_force; Function m_torque;
};
```

Kinematics source code

```
void RigidBody::Convert (quaternion Q, vector P,
    vector L, matrix& R, vector& V, vector& W) const {
  Q.ToRotationMatrix(R);
  V = m_invMass*P;
  W = R*m_invInertia*Transpose(R)*L;
}
```

Kinematics source code

```
void RigidBody::Update (double t, double dt) {
  double halfdt = 0.5 * dt, sixthdt = dt / 6.0;
  double tphalfdt = t + halfdt, tpdt = t + dt;
```

Kinematics source code

```
vector XN, PN, LN, VN, WN;
quaternion QN;
matrix RN;
// A1 = G(t,S0), B1 = S0 + (dt / 2) * A1
vector A1DXDT = m_V;
quaternion A1DQDT = 0.5 * m_W * m_Q;
vector A1DPDT = m_force(t,m_X,m_Q,m_P,m_L,m_R,m_V,
    m_W);
vector A1DLDT = m_torque(t,m_X,m_Q,m_P,m_L,m_R,m_V,
    m_W);
XN = m_X + halfdt * A1DXDT;
QN = m_Q + halfdt * A1DQDT;
PN = m_P + halfdt * A1DPDT;
LN = m_L + halfdt * A1DLDT;
Convert(QN,PN,LN,RN,VN,WN);
```

Kinematics source code

```
// A2 = G(t + dt / 2,B1), B2 = S0 + (dt / 2) * A2
vector A2DXDT = VN;
quaternion A2DQDT = 0.5 * WN * QN;
vector A2DPDT = m_force(tphalfdt,XN,QN,PN,LN,RN,VN,
    WN);
vector A2DLDT = m_torque(tphalfdt,XN,QN,PN,LN,RN,VN,
    WN);
XN = m_X + halfdt * A2DXDT;
QN = m_Q + halfdt * A2DQDT;
PN = m_P + halfdt * A2DPDT;
LN = m_L + halfdt * A2DLDT;
Convert(QN,PN,LN,RN,VN,WN);
```

Kinematics source code

```
// A3 = G(t + dt / 2,B2), B3 = S0 + dt * A3
vector A3DXDT = VN;
quaternion A3DQDT = 0.5 * WN * QN;

vector A3DPDT = m_force(tphalfdt,XN,QN,PN,LN,RN,VN,
    WN);
vector A3DLDT = m_torque(tphalfdt,XN,QN,PN,LN,RN,VN,
    WN);
XN = m_X + dt * A3DXDT;
QN = m_Q + dt * A3DQDT;
PN = m_P + dt * A3DPDT;
LN = m_L + dt * A3DLDT;
Convert(QN,PN,LN,RN,VN,WN);
```

Kinematics source code

```
    // A4 = G(t + dt,B3),
    // S1 = S0 + (dt / 6) * (A1 + 2 * A2 + 2 * A3 + A4)
    vector A4DXDT = VN;
    quaternion A4DQDT = 0.5 * WN * QN;
    vector A4DPDT = m_force(tpdt,XN,QN,PN,LN,RN,VN,WN);
    vector A4DLDT = m_torque(tpdt,XN,QN,PN,LN,RN,VN,WN);
    m_X = m_X + sixthdt*(A1DXDT + 2.0*(A2DXDT + A3DXDT)
        + A4DXDT);
    m_Q = m_Q + sixthdt*(A1DQDT + 2.0*(A2DQDT + A3DQDT)
        + A4DQDT);
    m_P = m_P + sixthdt*(A1DPDT + 2.0*(A2DPDT + A3DPDT)
        + A4DPDT);
    m_L = m_L + sixthdt*(A1DLDT + 2.0*(A2DLDT + A3DLDT)
        + A4DLDT);
    Convert(m_Q,m_P,m_L,m_R,m_V,m_W);
}
```

That's it

Thank you!