- PA 1 grade is out
  - No one seems to get collision response completely right….
  - I will select 3~4 games and post them online
  - Since no one did PA1 bonus points, I will reopen it and have the deadline on 11/10 midnight
- PA2 due 11/6 might night
  - Start early!
- PA3 is coming soon

- Since we never have quizzes, I will redistribute the grades:
  - PA1. 30%
  - PA2. 30%
  - Final project (40%)

```
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) */
};
```

```
/* Copy the state information into an array */
void StateToArray(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];
}
```

```
/* Copy information from an array into the state variables */
void ArrayToState(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;
}
```

```
#define STATE_SIZE      18

void ArrayToBodies(double x[])
{
    for(int i = 0; i < NBODIES; i++)
        ArrayToState(&Bodies[i], &x[i * STATE_SIZE]);
}


void BodiesToArray(double x[])
{
    for(int i = 0; i < NBODIES; i++)
        StateToArray(&Bodies[i], &x[i * STATE_SIZE]);
}
```

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

```
void Dxdt(double t, double x[], double xdot[])
{
    /* put data in x[] into Bodies[] */
    ArrayToBodies(x);

    for(int i = 0; i < NBODIES; i++)
    {
        ComputeForceAndTorque(t, &Bodies[i]);
        DdtStateToArray(&Bodies[i],
                        &xdot[i * STATE_SIZE]);
    }
}
```

```
void DdtStateToArray(RigidBody *rb, double *xdot)
{
    /* copy d/dt x(t) = v(t) into xdot */
    *xdot++ = rb->v[0];
    *xdot++ = rb->v[1];
    *xdot++ = 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++)
            *xdot++ = Rdot[i,j];

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

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

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

and returns the matrix

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

```
void RunSimulation()
{
    double  x0[STATE_SIZE * NBODIES],
            xFinal[STATE_SIZE * NBODIES];

    InitStates();
    BodiesToArray(xFinal);


    for(double t = 0; t < 10.0; t += 1./24.)
    {
        /* copy xFinal back to x0 */
        for(int i = 0; i < STATE_SIZE * NBODIES; i++)
        {
             x0[i] = xFinal[i];

        ode(x0, xFinal, STATE_SIZE * NBODIES,
            t, t+1./24., Dxdt);

        /* copy d/dt X(t + 1/24) into state variables */

        ArrayToBodies(xFinal);
        DisplayBodies();
    }
}
```

```
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;             /* ω(t) */

    /* Computed quantities */



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

Next, in the routine `StateToArray`, 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;
```

Normalize R

- R is derived from integration so may deform over time

- How do you Normalize R?

Add Quaternion

```
/* 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 = QuaterionToMatrix(normalize(rb->q));
```

```
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++)
        *xdot++ = Rdot[i,j];
```

we'll use

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

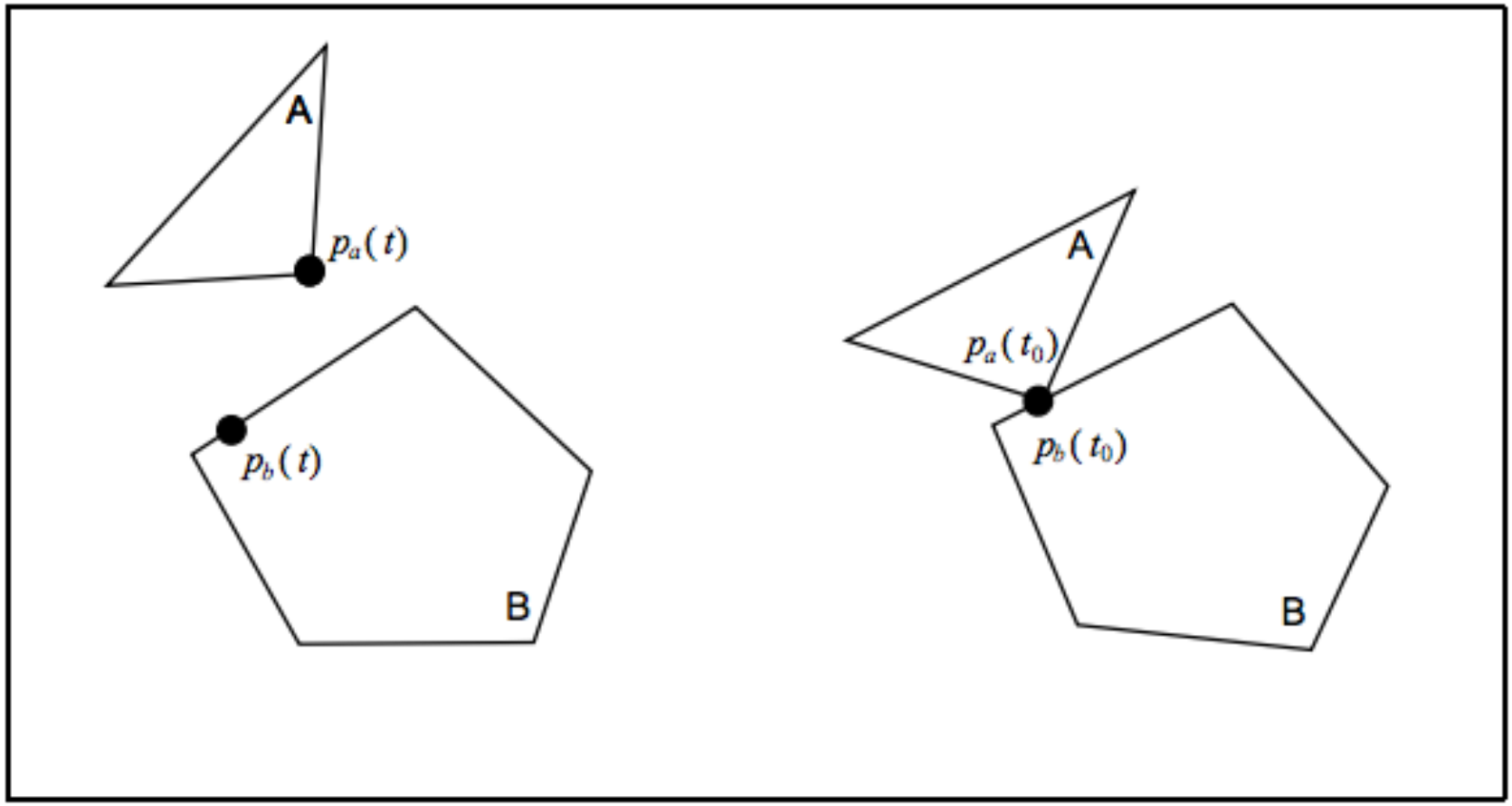
[0, rb->omega]q.

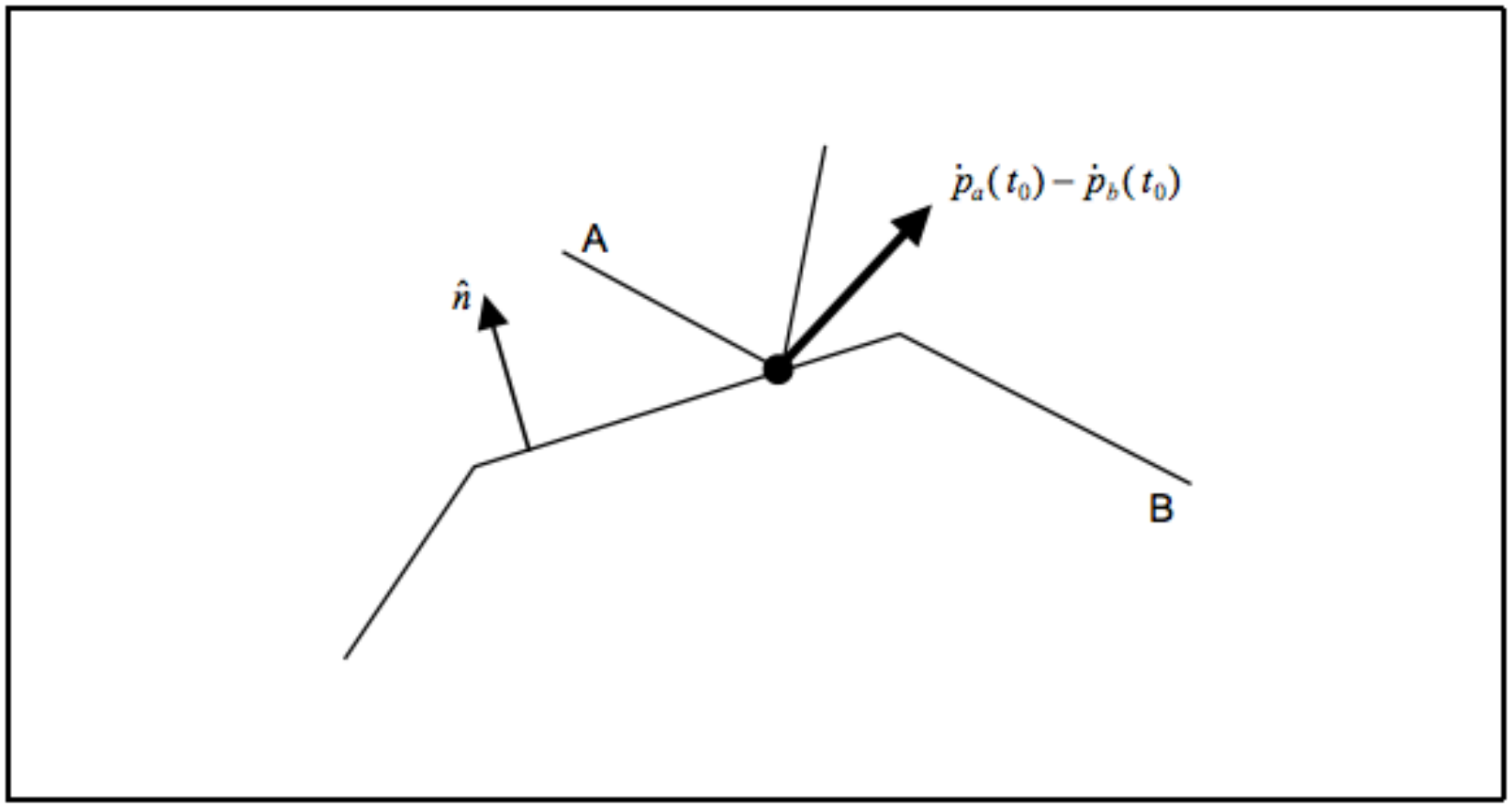
```
struct Contact {
    RigidBody       *a,     /* body containing vertex */
                    *b;     /* body containing face */
    triple          p,      /* world-space vertex location */
                    n,      /* outwards pointing normal of face */
                    ea,     /* edge direction for A */
                    eb;     /* edge direction for B */
    bool            vf;     /* true if vertex/face contact */
};

int     Ncontacts;
Contact *Contacts;
```

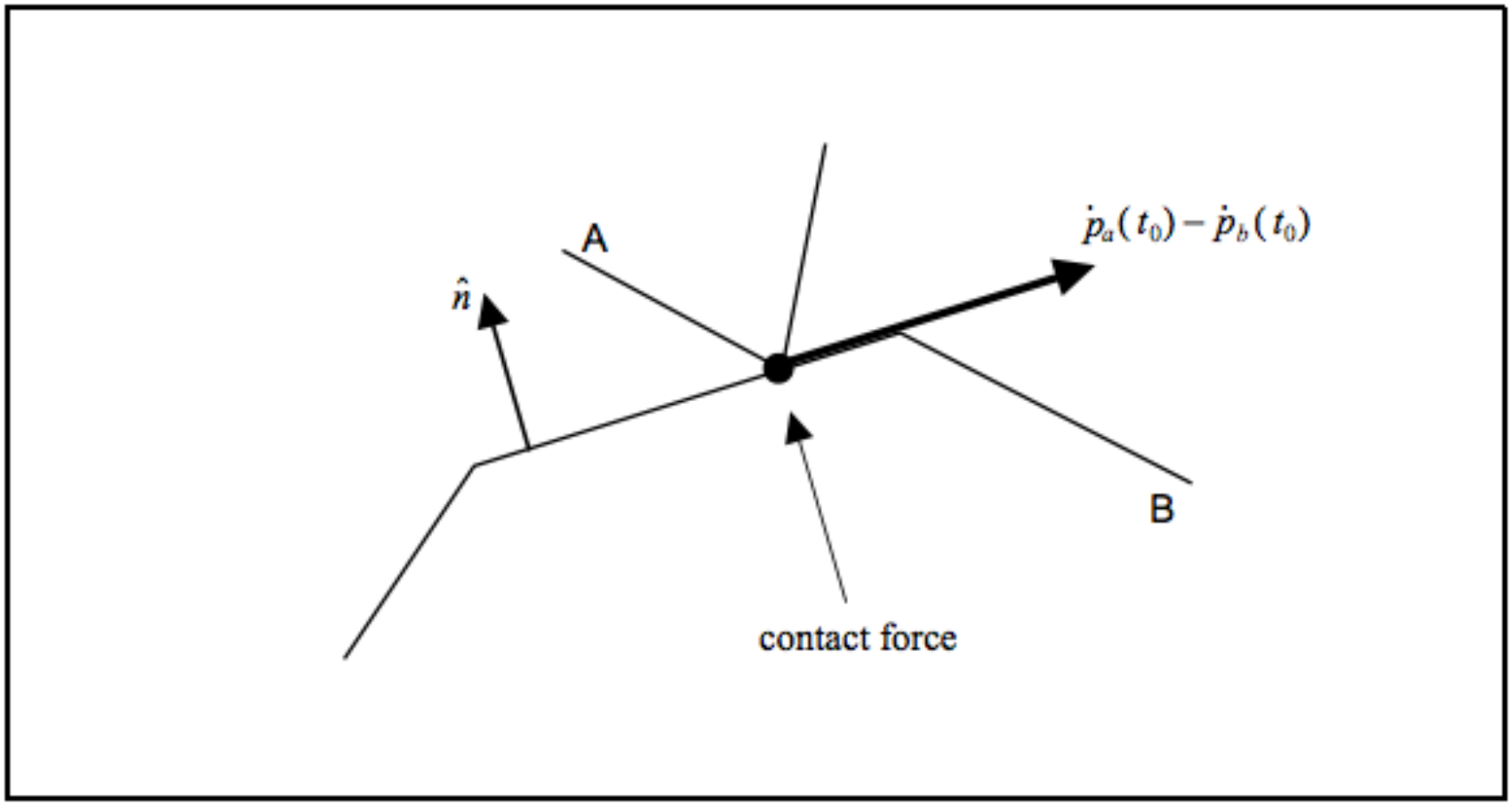
PA 2

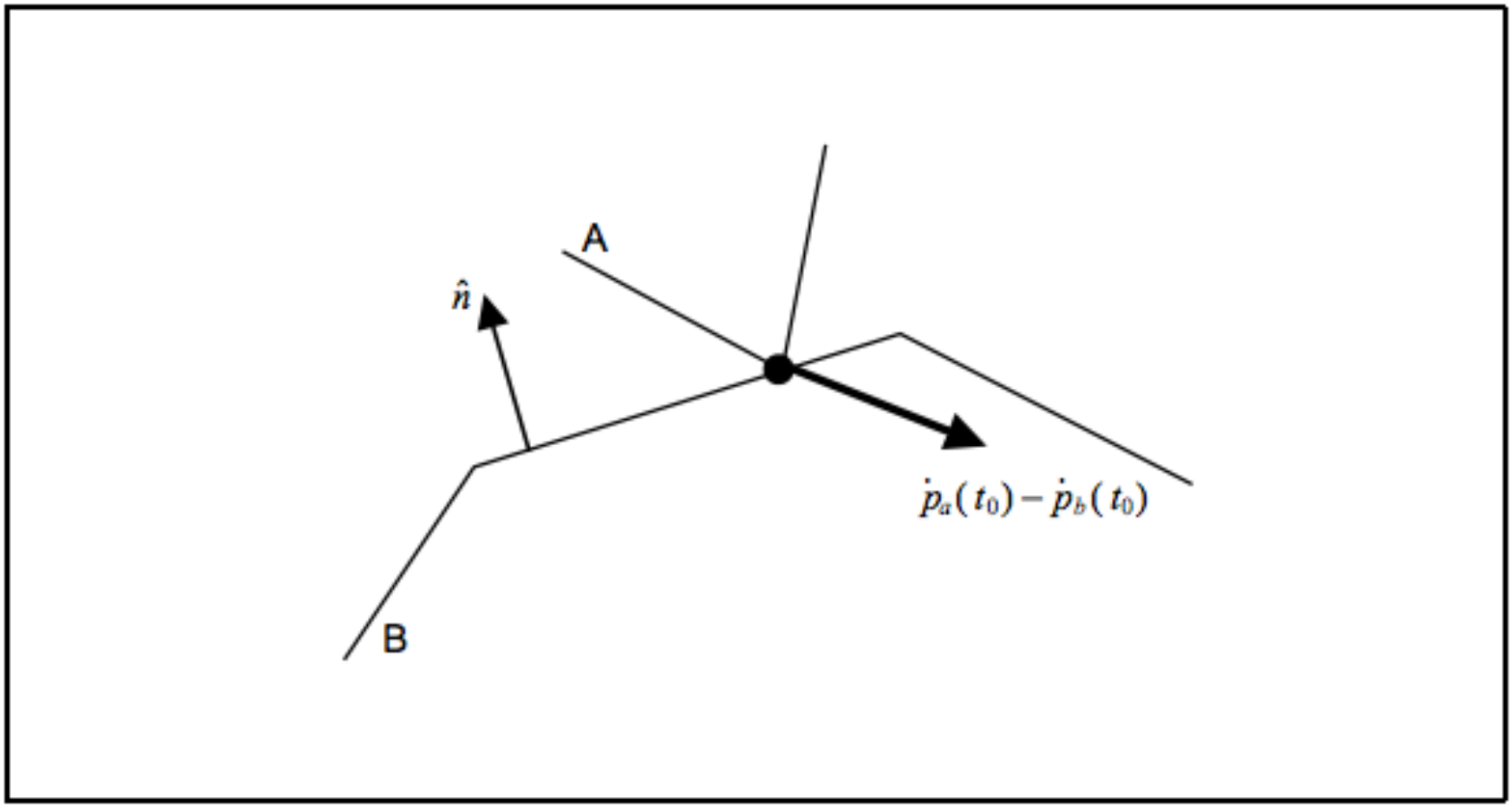
- 10 points bonus if you replace matrix with quaternion in your PA2


A
$p_a(t)$
$p_b(t)$
B
A
$p_a(t_0)$
$p_b(t_0)$
B



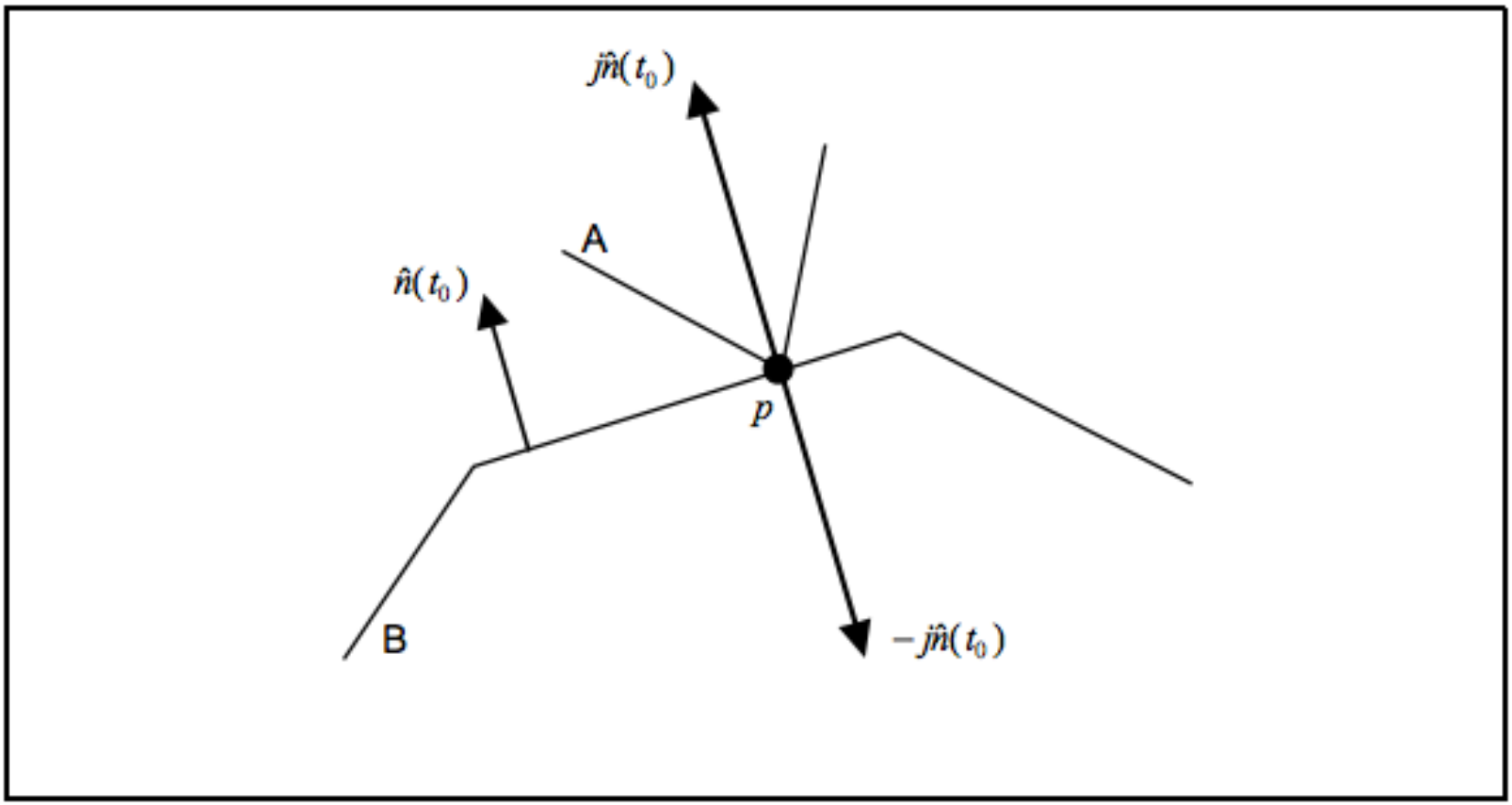
$$v_{rel} = \hat{n}(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0)),$$

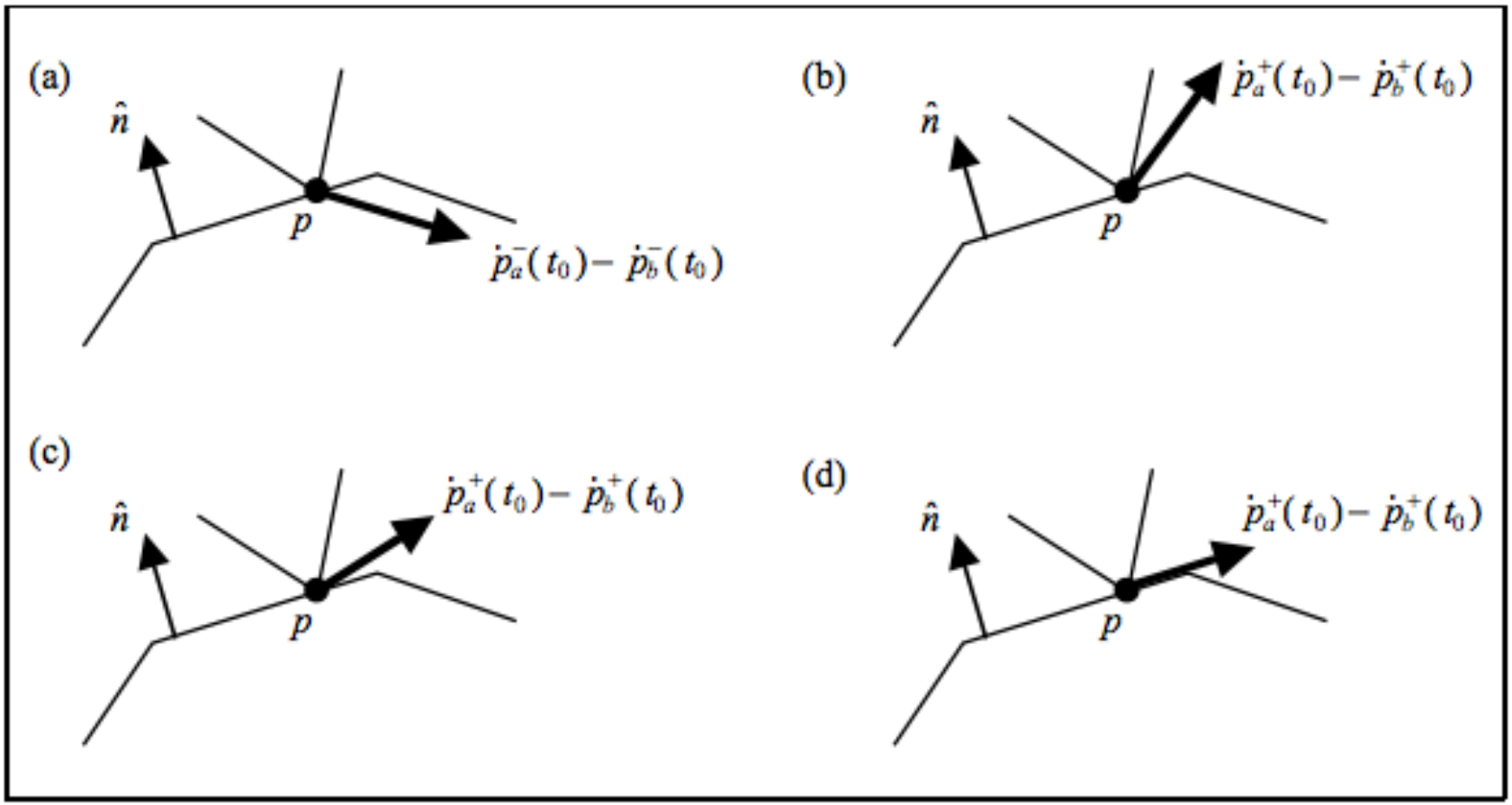

A
$\hat{n}$
$\dot{p}_a(t_0) - \dot{p}_b(t_0)$
B
contact force


A
$\hat{n}$
$\dot{p}_a(t_0) - \dot{p}_b(t_0)$
B

Impulse

- Instead of apply force, we apply impulse at the location of collision
  - Force takes time to take effect due to momentum
- Impulse is similar to force but
  - Has unit of momentum
  - Has immediate effect on the velocity


$j\hat{n}(t_0)$
A
$\hat{n}(t_0)$
p
B
$-j\hat{n}(t_0)$


(a)
$\hat{n}$
p
$\dot{p}_a^-(t_0) - \dot{p}_b^-(t_0)$
(b)
$\hat{n}$
p
$\dot{p}_a^+(t_0) - \dot{p}_b^+(t_0)$
(c)
$\hat{n}$
p
$\dot{p}_a^+(t_0) - \dot{p}_b^+(t_0)$
(d)
$\hat{n}$
p
$\dot{p}_a^+(t_0) - \dot{p}_b^+(t_0)$

```
/*
 * Operators: if 'x' and 'y' are triples,
 * assume that 'x ^ y' is their cross product,
 * and 'x * y' is their dot product.
 */

/* Return the velocity of a point on a rigid body */
triple pt_velocity(Body *body, triple p)
{
    return body->v + (body->omega ^ (p - body->x));
}

/*
 * Return true if bodies are in colliding contact.  The
 * parameter 'THRESHOLD' is a small numerical tolerance
 * used for deciding if bodies are colliding.
 */
bool colliding(Contact *c)
{
    triple  padot = pt_velocity(c->a, p),   /* p_a^-(t_0) */
            pbdot = pt_velocity(c->b, p);   /* p_b^-(t_0) */
    double  vrel = c->n * (padot - pbdot);  /* v_rel^- */

    if(vrel > THRESHOLD)    /* moving away */
        return false;
    if(vrel > -THRESHOLD)   /* resting contact */
        return false;
    else                    /* vrel < -THRESHOLD */
        return true;
}
```

```
void collision(Contact *c, double epsilon)
{
    triple  padot = pt_velocity(c->a, c->p),  /* ṗ_a^-(t_0) */
            pbdot = pt_velocity(c->b, c->p),  /* ṗ_b^-(t_0) */
            n = c->n,                         /* n̂(t_0) */

            ra = p - c->a->x,                 /* r_a */
            rb = p - c->b->x;                 /* r_b */
     double vrel = n * (padot - pbdot),       /* v_rel^- */
            numerator = -(1 + epsilon) * vrel;

     /* We'll calculate the denominator in four parts */
     double term1 = 1 / c->a->mass,
            term2 = 1 / c->b->mass,
            term3 = n * ((c->a->Iinv * (ra ^ n)) ^ ra),
            term4 = n * ((c->b->Iinv * (rb ^ n)) ^ rb);

     /* Compute the impulse magnitude */

     double j = numerator / (term1 + term2 + term3 + term4);
     triple force = j * n;

     /* Apply the impulse to the bodies */
     c->a->P += force;
     c->b->P -= force;
     c->a->L += ra ^ force;
     c->b->L -= rb ^ force;

     /* recompute auxiliary variables */
     c->a->v = c->a->P / c->a->mass;
     c->b->v = c->b->P / c->b->mass;

     c->a->omega = c->a->Iinv * c->a->L;
     c->b->omega = c->b->Iinv * c->b->L;
}
```

```
void FindAllCollisions(Contact contacts[], int ncontacts)
{
    bool    had_collision;
    double  epsilon = .5;

    do {
        had_collision = false;

        for(int i = 0; i < ncontacts; i++)
            if(colliding(&contacts[i]))
            {
                collision(&contacts[i], epsilon);
                had_collision = true;

                /* Tell the solver we had a collision */
                ode_discontinuous();

             }

     } while(had_collision == true);
 }
```