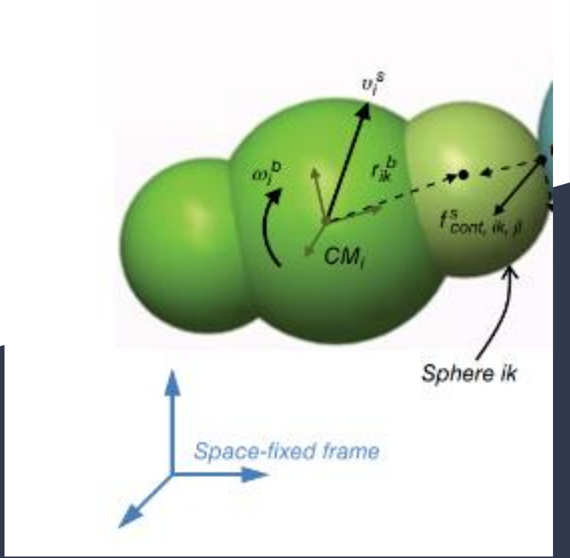
Single Multisphere Particle Simulation

Simulating a Single Multisphere Particle enclosed in a box of the given edges (0,0,0) and (a,a,a).

The code for MUSEN uses Quaternions for the Simulation of Motion of Non-Spherical Particles.

Both the MUSEN and LIGGGHTS codes make use of the Object Oriented Programming in C++.



Defining the Quaternion:

To make use of the Quaternion Mathematical Framework we first need to define the Attributes and Properties of the Quaternions in C++.

Hence we define the class **quaternion.**

A quaternion is defined using a scalar value **s** and a 3D vector **v** as follows:

$$q = (s, \vec{v}) \quad \text{or} \quad q = s\tilde{e}_0 + v_1\tilde{e}_1 + v_2\tilde{e}_2 + v_3\tilde{e}_3$$

```
class quaternion

{

        float  s, vx, vy, vz;

        quaternion product(quaternion Q1, quaternion
Q2);

        // Calculates the product of two quaternions

    quaternion sum(quaternion Q1, quaternion Q2);

    // Calculates the sum of two quaternions

        quaternion diff(quaternion Q1, quaternion Q2);

        // Calculates the difference of two quaternions

        quaternion T(quaternion Q1);

        void display(quaternion Q1);

        // Displays the quaternion

}
```

Quaternion Product Function:

The product of two quaternions is defined as follows:

$$q_1.q_2 = (s_1, \vec{v}_1).(s_2, \vec{v}_2) = (s_1 s_2 - \vec{v}_1.\vec{v}_2,\ s_1\vec{v}_2 + s_2\vec{v}_1 + \vec{v}_1 \times \vec{v}_2)$$

```
quaternion product(quaternion q1, quaternion q2)

{

quaternion p;

p.s =  (q1.s)*(q2.s)-(q1.vx)*(q2.vx)-(q1.vy)*(q2.vy)-(q1.vz)*(q2.vz);

p.vx = (q1.s)*(q2.vx)+(q2.s)*(q1.vx)+((q1.vy)*(q2.vz) - (q1.vz)*(q2.vy));

p.vy = (q1.s)*(q2.vy)+(q2.s)*(q1.vy)+((q1.vz)*(q2.vx) - (q1.vx)*(q2.vz));

p.vz = (q1.s)*(q2.vz)+(q2.s)*(q1.vz)+((q1.vx)*(q2.vy) - (q1.vy)*(q2.vx));

return p;

}
```

Defining the Sphere:

LIGGGHTS uses the Multisphere model.

Since all the Non Spherical Particles which we are going to represent will be made out of spheres hence we need to define a **sphere** as a class.

Each sphere will be defined by the position of its center in the Local Body Frame and in the Global Frame and its radius.

We also define the Force acting on a sphere at any given instant of time.

```
class sphere
{
        quaternion rs, rb; // Position Vectors
        quaternion F; // Force
        float R;      //Radius of the Sphere
}
```

Defining the Particle:

Now since we have defined our tools for the Multisphere Model for Non Spherical Particles we can define the Particle.

```
class particle
{
        quaternion rc; // Global coordinates of the CoM
        quaternion q; // Orientation wrt Global Coordinates
        quaternion wb; // Angular Velocity of the body
        quaternion v; // Translational Velocity of the body
        quaternion dq; // Derivative of q at that moment
        quaternion T; // Net Torque acting on the body
        quaternion F; // Net Force acting on the body
        int N_spheres; // Total Number of Spheres
    sphere S[N_spheres]; // Array of spheres which make the particle
    void particle(); // Constructor
    //constructor which defines the array of spheres on object creation
    //constructor also initializes the Physical Properties of the object
}
```

Defining the Simulation Domain:

```
class simDom

{

        float x1, y1, z1;

        float x2, y2, z2;

        void simDom();// Constructor to define
simDom()

}
```

Main Algorithm for Collision with Walls of Simulation Domain:

```
particle A;                                        // Define the Particle
simDom box;            //Define simulation domain
for i = 1 to N_iterations
{
            for j = 1 to A.N_spheres
            {
            curr = A.S[i];
            curr.rs = quaternion.product(particle.q, curr.rb);
            curr.rs = quaternion.product(curr.rs, quaternion.T(particle.q));
            oxy1 = (curr.rs.vz - box.z1);  oxy2 = (-curr.rs.vz + box.z2);
            oyz1 = (curr.rs.vx - box.x1);  oyz2 = (-curr.rs.vx + box.x2);
            ozx1 = (curr.rs.vy - box.y1);  ozx2 = (-curr.rs.vy + box.y2);
            bool contact = [0,0,0,0,0,0];
            // Update the array with contact status for the current sphere
            float contF = [0,0,0,0,0,0];
            A.S[i].F = // Define the Force acting on the sphere;
            }
A.computeNetF();
A.computeNetT();
}
```