Cite as: Kumar, Navdeep: A rigidBodyDynamics tutorial with demostrations. In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H.,
`http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016`

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY
TAUGHT BY HÅKAN NILSSON

Project work:

A rigidBodyDynamics tutorial with demostrations

Developed for OpenFOAM-4.x

Author:
NavdeepKUMAR
Chalmers University of Technology
nddpsdw@gmail.com

Peer reviewed by:
MOHAMMAD HOSSEIN ARABNEJAD
HÅKAN NILSSON

February 14, 2017

Contents

Learning outcomes

How to use it:

- How to setup cases involving rigid body motion with constraints and restraints.

The theory of it:

- The theory of rigid body dynamics and the algorithm used in this library.

How it is implemented:

- How the library solves rigid body motion for each time step.

How to modify it:

- How to add new bodies and restraints.

Chapter 1

Theory

1.1 Introduction

This project explores the use of the rigidBodyDynamics library which has been introduced in OpenFOAM 4.x. The code has been developed for solving rigid body motion problems where the bodies are connected to each other having some motion. Bodies may be restrained (forces acting on the body) and constrained (degree of freedom is restricted). A simple case is shown in figure 1.1 as an illustration: It is based on field of robotics, although this library has been developed to solve wave

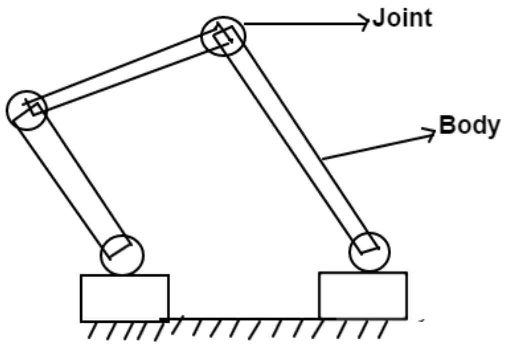

Figure 1.1: Illustrative figure of multi body system

energy problems. Multiphase (air and water) motions are involved in wave energy problems, which is why the interDyMFoam solver has been used in the demonstration problems. With this library such problems can be solved to calculate the accelerations, velocities etc at different points in the system. The algorithm used to develop the code has been taken from the book by Ry Featherstone[1]. The present tutorial starts with some theory behind the algorithm. After that some cases will be discussed to explore the options in this library which can be used to set up the problems.

1.1.1 Mathematical Background

Some Definitions:
These terms will be used in this section.
Articulated-Body: Consider the case of a serial linkage of n-links. Sever the linkage at joint i. The remaining linkage is called a sub-chain. This sub-chain is called an articulated body of the original linkage. The inertia of this body is called articulated-body inertia.
Link are the bodies as seen in the figure below.
Inboard means that we are moving towards the base from the body.
Outboard means that we are moving in opposite direction to inboard.
Joint means the connection between two consecutive links.

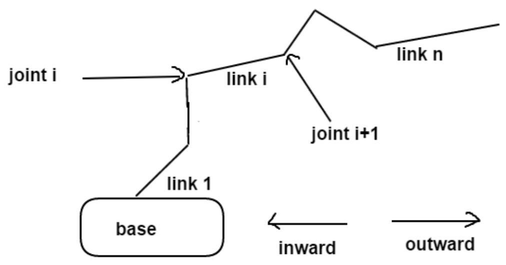

Figure 1.2: n link system with base

The Rigid body dynamics library uses articulated body algorithm to solve rigid body problems. This section will elaborate upon this algorithm. This algorithm can be used to solve highly complex rigid body model problems. We will restrict ourselves to a simple case to understand the algorithm after which it can be extended to a general case.
The equation of motion of a rigid body system can be written as $H(q)\ddot{q} + C(q, \dot{q}) = \tau$.

Here q, $\dot{q}$, $\ddot{q}$ are the position, velocity and acceleration of the the joints. H is the inertia matrix. C is the force which will produce zero acceleration. C accounts for forces like gravity, Coriolis and centrifugal forces. τ is the total force acting on the bodies. C can also be said to be the value of τ that will produce zero acceleration. In this equation multiple values can be calculated depending on what has been provided. It can be divided into forward dynamics and inverse dynamics.
For forward dynamics we have:

$$\ddot{q} = FD(model, q, \dot{q}, \tau) \tag{1.1}$$

It means that the initial position, velocities and forces have been provided which will be used to calculate the acceleration.
For inverse dynamics we have:

$$\tau = ID(model, q, \dot{q}, \ddot{q}) \tag{1.2}$$

It is the inverse of forward dynamics, which means that acceleration has been provided and it will be used to calculate the required force to have that acceleration. We will work with forward dynamics as the forces will be known to us which will be used to calculate the acceleration. This acceleration will be integrated for the first time step to get the new position and velocity which will be used as an input for the next time step. And this will continue during the whole period of the simulation.

Now, we will get into some details on how it is done. First of all the process must have less computational complexity. Articulated body algorithm has a complexity of the order of(n) which is the least possible order of complexity. It is a propagation method of solving forward dynamics problem. It will be seen shortly why it is called propagation method. As can be seen in figure 1.2 that there are many joints and links. The acceleration of every link needs to be known to get the position and velocities. It is not possible to solve for the unknowns locally (link frame), but rigid body dynamics equations can be formulated for all the bodies. The Propagation method works by calculating the coefficients of the rigid body dynamics equations for one link and then propagating it to neighbouring bodies until the end of the chain is reached. It can be understood as:

$$f = I^A * a + C \tag{1.3}$$

Here I^A is the articulated body inertia which is starting from the inboard joint of this link to the end of the chain, a is the link acceleration and C is the bias force.f is the force applied by the inboard joint of this link. All the values are unknown over here. These values are calculated starting from the tip and ending at the base of the system. It is done inductively. It means that none of the equations can be solved individually. All the equations have to be formed which will then be solved simultaneously because there is interdependence between them. To make it more clear an equation for acceleration of link i has been shown from a thesis on rigid body dynamics [2] as:

$$\ddot{q_i} = \frac{Q_i - \hat{s_i}'\hat{I}_i^A \hat{X}_{i-1}\hat{a}_{i-1} - \hat{s_i}'(\hat{Z}_i^A + \hat{I}_i^A \hat{c}_i)}{\hat{s_i}'\hat{I}_i^A \hat{s}_i} \tag{1.4}$$

Here subscript i denotes the link number varying between 1 to n. $\ddot{q}$ is link acceleration. All the values seen in the above figure needs to be calculated for every value of i. Similar equation exists for articulated body inertia, bias forces. This means that all the equations have to be formed from 1 to n which will then be solved together to give the acceleration of all the links. This is why it is called propagation method. To conclude there are three basic steps in this algorithm:

- Computing the absolute velocities of all links starting from the tip to the base.
- Computing the articulated inertia and zero acceleration force (bias force) for each link starting from tip to the base.
- Computing the acceleration of each joint starting from the tip to the base.

1.2 Rigid body dynamics library

```
rigidBodyDynamics
├─ bodies
├─ joints
├─ restraints
├─ rigidBodyInertia
├─ rigidBodyModel
├─ rigidBodyModelState
├─ rigidBodyMotion
└─ rigidBodySolvers
```

This is the original file structure of the library. For the ease of understanding of the readers, it is divided into two parts, Main files and auxiliary files. The division is shown below:

```
Main Files
├─ bodies
├─ joints
└─ restraints
```

```
Auxiliary Files
├─ rigidBodyInertia
├─ rigidBodyModel
├─ rigidBodyModelState
├─ rigidBodyMotion
└─ rigidBodySolvers
```

1.2.1 Main Files

In this section RBD library will be explored. The library has been divided into many files but this project will focus on the three main divisions which will be used while setting up the case in dynamicMeshdict. The three main divisions are:

- bodies
- joints
- restraints

Bodies

This section will explore the types of bodies available in this library. The functionality of `mergeWith` will also be discussed. The files contained in this folder are:

- compositeBody

- cuboid
- jointBody
- masslessBody
- rigidBody
- sphere
- subBody

These files are used to create the bodies such as cuboid, sphere, massless body etc.
Cuboid is a simple geometry and everyone is familiar with it. To create a cuboid, the length of the three edges is required. It can be entered in the dynamicMeshdict as will be shown later with illustrative examples.
Sphere needs radius of the body which will be shown later.
Massless body can be used as a connector between 2 bodies and it has zero mass.
rigidBody can be used to create different geometries which are not of the predefined shapes. In the tutorial of DTC hull the use of rigidBody can be seen which has imported a geometry, patches has been created after which it has been declared a rigidBody. The path of the tutorial is mentioned below:

```
cd $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/DTCHull
```

The use of rigidBody can be demonstrated better with an example. In the floatingObject tutorial in `dynamicMeshDict` the cuboid can be changed with a rigidBody as shown below:

```
    floatingObject
        {
            type            rigidBody;
            parent          root;

            // Cuboid properties
            mass           15;
            centreOfMass    (0 0 0.25);
            inertia         (0.3625 0 0 0.425 0 0.1625);
            transform       (1 0 0 0 1 0 0 0 1) (0.5 0.45 0.1);//0.5 0.45 0.1

            joint
            {
                type            composite;
                joints
                (
                    {
                        type Py;
                    }
                    {
                        type Ry;
                    }
                );
            }

            patches         (floatingObject);
            innerDistance   0.05;
            outerDistance   0.35;
        }
```

Here inertia is written in the following format: `(Ixx,Ixy,Ixz,Iyy,Iyz,Izz)`. Ixx, Iyy and Izz is the inertia about x,y and z axis while Ixy, Ixz and Iyz are the product of inertia. The changes made above in `dynamicMeshDict` will work exactly like the previous entry which has been modified in floatingObject tutorial. However it is not preferable to use rigidBody for simple geometries like cuboid because it is already available in the library. rigidBody is crucial for irregular geometries. The option of entering the product of inertia is not available in sixDofRigidBodyMotion library.
jointBody has not been explored and the syntax for using it needs to be studied further. Valid types of the bodies which can be used in the dynamicMeshDict are cuboid, jointBody, masslessBody, rigidBody and sphere. The rest two bodies cannot be entered as the entry to 'type' in the dictionary.

mergeWith can be used to merge two bodies together. An example of a pendulum is shown to see how it can be used:

```
bodies
{
    hinge
    {
        type            masslessBody;
        parent          root;
        transform       (1 0 0 0 1 0 0 0 1) (0 0 0);
        joint
        {
            type            Rz;
        }
    }
    weight
    {
        type            sphere;
        mass            1;
        centreOfMass    (0 -1 0);
        radius          0.05;
        transform       (1 0 0 0 1 0 0 0 1) (0 -1 0);
        mergeWith       hinge;
    }
```

Here a pendulum has been created by merging the bob of the pendulum with a hinge point. The hinge has been taken as a massless body. mergeWith has been used here. Notice that weight does not need a parent body since it is getting merged with the hinge. The hinge has a parent body 'root'. Root is a massless body at origin. Every independent body (body which is not attached to any other body) has to be merged with this root body to make a rigid body model.

Joints

In a multibody system as shown in figure 1.2 in the previous section, the bodies are connected to other bodies through some joints. Mainly the joints can be classified into 3 primitive types:

- Prismatic
- Revolute
- Spherical

A prismatic joint defines an axis of translation as shown in the figure 1.4. Only sliding motion can be performed in a purely prismatic joint.

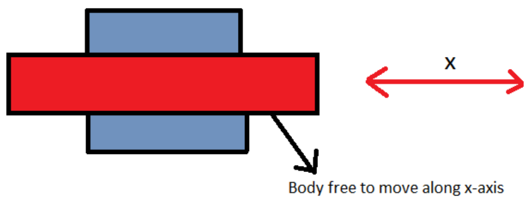

Figure 1.3: Prismatic joint-Body free to translate along an xis

A revolute joint defines an axis of revolution as shown in figure 1.5. Only rotational motion can be given around some specified axis in a purely revolute joint.

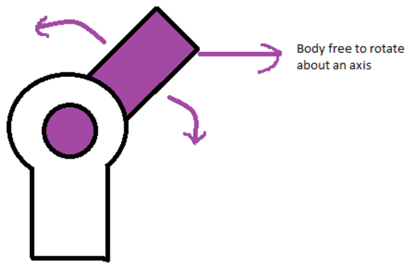

Figure 1.4: Revolute joint-body free to rotate about an axis

A spherical joint defines a pivot point where 3-dimensional spherical rotation can take place.It has been illustrated in figure 1.5.

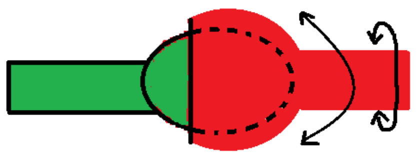

Figure 1.5: Spherical joint-Body has six degrees of freedom

There are some other joints as well, such as composite joint, floating joint, null joint etc. Composite joint is used when a body is constrained by a combination of joints such as a mixture of prismatic and revolute joint. Floating joint provides 6 degrees of freedom. Null joint is used in the construction of joints in C++ files and all the bodies are initiated from this joint. Prismatic joints with the corresponding degrees of freedom have been listed below:

- Px-translation along the x-axis
- Py-translation along the y-axis
- Pz-translation along the z-axis
- Pxyz-translation along the x,y and z-axis
- Pa-translation along arbitrary axis and this axis can be entered by user

The axis can be entered by the user for arbitrary axis using keyword `axis` under joints option in dynamicMeshDict. Similarly for revolute joints the following options are available:

- Rx-rotation about x-axis
- Ry-rotation about y-axis
- Rz-rotation about z-axis
- Rxyz-spherical joint for rotation about x,y and z-axis in order of x,y and z using euler angles
- Ryxz-spherical joint for rotation about x,y and z-axis in order of y,x and z using euler angles
- Rzyx-spherical joint for rotation about x,y and z-axis in order of z,y and x using euler angles
- Ra-rotation about arbitrary axis and this axis can be entered by user

All these things will be more clear as we proceed with different examples demonstrating the use of these joints.

Restraints

Restraints contains the different restraints which can be applied to the body. The following restraints can be used:

- linearAxialAngularSpring
- linearDamper
- linear Spring
- sphericalAngularDamper

linearAxialAngularSpring
linearAxialAngularSpring is torsional spring which can be attached to a body. It needs some specific inputs as shown in the example below:

```
restraints
        {
           torsionalspring
           {
            body        floatingObject;
            type       linearAxialAngularSpring;
            axis                        (1.0 2.0 3.0);
            referenceOrientation        (1 0 0 1 0 0 1 0 0);
            stiffness            100;
```

```
                damping              0;
                }
             }
```

Here axis specifies the axis of rotation of the body which has to be entered as a vector. The reference Orientation has to be entered as a tensor where there is zero moment on the body. Stiffness is the torsional spring stiffness while damping is the torsional spring damping coefficient. This restraint will compute the torque proportional to the rotation angle of the body and will always act in the opposite direction of rotation.

sphericalAngularDamper
sphericalAngularDamper restraint gives a reactionary force directly proportional to the angular velocity of the body. In the dynamicMeshDict, it requires only the coefficient of the damper which is *torque exerted/angular velocity*. As an example the following is shown:

```
restraints
            {
               torsionalspring
               {
                body        floatingObject;
                type       sphericalAngularDamper;
                coeff           10;
                }
             }
```

The Linear damper restraint is similar to the one described above. The force exerted is directly proportional to the velocity of the body. It needs only the coeff of the damper. Linear spring restraint has been used in the demonstration. This restraint gives a force directly proportional to the extension of the spring.

1.2.2 Auxiliary Files

Here all the remaining files will be discussed. They are used to perform all the background computations that are necessary to make the main files work the way they should.

rigidBodyModel

The rigidbodyModel class is used to create a system of rigid bodies which are connected to each other using 1-6 DoF joints. For this purpose a convention is followed as discussed by Roy Featherstone [1](Chapter 4). `rigidBodyModel.C` has the code for creation of the rigid body system. It nicely follows the algorithm presented by Roy Featherstone[1]. The code starts with the root body. which is a massless body placed at the origin after which bodies mentioned in the dynamicMeshDict are appended to this body. Similarly, joints are created and attached to the mentioned bodies as instructed in the dynamicMeshDict. This library is using forward dynamics. It can be seen in the `forwarddynamics.C` file present in the `rigidBodyModel`. To solve the forward dynamics problems there are certain algorithms. Two of these have been discussed by Roy Featherstone[1]. This library uses *articulated body algorithm*. This algorithm has been discussed in previous section. In the file `forwardDynamics.C` it has been implemented where the acceleration of the joint is calculated from the joint state data, velocity of joint and force data taken from model state. This file is present in the `rigidBodyModel` folder. It is discussed in some detail below.

forwardDynamics

The Articulated body algorithm takes place in three sweeps. The first sweep is an outward sweep as is seen below:

```
    for (label i=1; i<nBodies(); i++)
       {
           const joint& jnt = joints()[i];
           jnt.jcalc(J, q, qDot);

           S_[i] = J.S;
          S1_[i] = J.S1;

           Xlambda_[i] = J.X & XT_[i];

           const label lambdai = lambda_[i];

           if (lambdai != 0)
           {
               X0_[i] = Xlambda_[i] & X0_[lambdai];
           }
           else
           {
               X0_[i] = Xlambda_[i];
           }

           v_[i] = (Xlambda_[i] & v_[lambdai]) + J.v;
           c_[i] = J.c + (v_[i] ^ J.v);
           IA_[i] = I(i);
           pA_[i] = v_[i] ^* (I(i) & v_[i]);
            if (fx.size())
           {
               pA_[i] -= *X0_[i] & fx[i];
           }
  }
```

The calculation is done starting from the base to the tip of the model. The velocities of the bodies are calculated. Using this velocity rigid body bias force and the velocity-product acceleration is computed. The inertia of all the individual bodies are taken and stored in IA as can be seen in the code above. After this the second sweep is carried out from the tip of the model to base of the model. In this the articulated inertia and bias forces are calculated as shown below in the for loop:

```
    for (label i=nBodies()-1; i>0; i--)
```

The whole code is not shown above. It can be seen by opening the file. After this the joint acceleration is computed which is denoted by qDdot. It can be seen in the code. This sweep is carried out from the base to the tip of the model. The main purpose of this exercise is to calculate all the joint accelerations. This updated joint acceleration is stored in rigidBodyModelstate.

rigidBodyModelState

The rigidBodyModelState class holds the motion state of the rigid body model. The motion state of the body means the joint position, velocity, acceleration etc. The initial values of the joint position, orientation, velocity and acceleration can be entered in the dynamicMeshDict using keywords q, qdot etc as can be seen from the following lines which is present in `rigidBodyModelState.C`:

```
Foam::RBD::rigidBodyModelState::rigidBodyModelState
(
    const rigidBodyModel& model,
    const dictionary& dict
)
:
    q_(dict.lookupOrDefault("q", scalarField(model.nDoF(), Zero))),
    qDot_(dict.lookupOrDefault("qDot", scalarField(model.nDoF(), Zero))),
    qDdot_(dict.lookupOrDefault("qDdot", scalarField(model.nDoF(), Zero))),
    deltaT_(dict.lookupOrDefault<scalar>("deltaT", 0))
{}
```

The output of the member functions in this class are called upon by rigidBodyMotion and rigidBodySolver wherever these classes require the motion state of the body.

rigidBodyInertia

The rigidBodyInertia directory contains the following files:

- rigidBodyInertia.H
- rigidBodyInertiaI.H

The first file creates the `rigidBodyInertia` class and contains the declarations of the variables, member functions, constructors which will be used later. In the second file the variables required for inertia are initialized to zero value. The dictionary is read and the corresponding values for inertia about center of mass, mass and center of mass are read. These values are then operated upon to get inertia tensor about origin. As an example the function is shown below:

```
inline Foam::symmTensor Foam::RBD::rigidBodyInertia::Io() const
   {
       return Ic_ + Ioc();
   }
```

This function returns the inertia tensor about the origin. Ic is the inertia about the center of mass and Ioc is the difference in inertia about the center of mass and origin. Since the body will be moving during the simulation it will result in the moving center of mass, which will change the difference in inertia between the center and origin. This difference is given by `Icc` as shown below:

```
inline Foam::symmTensor Foam::RBD::rigidBodyInertia::Icc(const vector& c) const
    {
       return Ioc(m_, c - c_);
    }
```

Here the arguments m is mass, c is moving center of mass and c_ is initial center of mass. The function shown below works on inertia of the combined bodies.

```
inline rigidBodyInertia operator+
   (
       const rigidBodyInertia& rbi1,
       const rigidBodyInertia& rbi2
   )
   {
       const scalar m12 = rbi1.m() + rbi2.m();
       const vector c12 = (rbi1.m()*rbi1.c() + rbi2.m()*rbi2.c())/m12;

       return rigidBodyInertia
```

```
        (
            m12,
            c12,
            rbi1.Ic() + rbi1.Icc(c12) + rbi2.Ic() + rbi2.Icc(c12)
        );
    }
```

To conclude, this class works on the inertia of the body doing the necessary computations while the body is moving and returning its values to `rigidBody.H` file to be used by other bodies present in the bodies folder.

rigidBodySolver

The rigidBodySolver class contains the different methods that can be used to solve the equations. The solvers contained in it are:

- Crank Nicholson
- Newmark
- Symplectic

One of these solvers can be chosen according to requirements of the problem to be solved. This has to be entered in the `dynamicMeshDict` dictionary under `rigidBodyMotionCoeffs`. There is one more folder, `rigidBodySolver`, which contains the required source and header files for the `rigidBodySolver` class. This class will return the required joint position, velocity, acceleration, time step etc to the solvers (crank nicholson, newmark, symplectic).

rigidBodyMotion

The rigidBodyMotion class calls rigidBodySolver, rigidBodyModel, rigidBodyModelState to compute and print the status of the rigid body in each time step. `rigidBodyMotion.C` implements forward dynamics (articulated body algorithm) as can be seen in the following lines:

```
void Foam::RBD::rigidBodyMotion::forwardDynamics
  (
       rigidBodyModelState& state,
       const scalarField& tau,
       const Field<spatialVector>& fx
   ) const
   {
       scalarField qDdotPrev = state.qDdot();
       rigidBodyModel::forwardDynamics(state, tau, fx);
       state.qDdot() = aDamp_*(aRelax_*state.qDdot() + (1 - aRelax_)*qDdotPrev);
   }
```

After this implementation, the acceleration of the joints is known. These accelerations are relaxed or damped as entered in the dynamicMeshDict. Because suddden increase in accelerations lead to divergence of the solution, it becomes important to use acceleration relaxation. The new joint acceleration is computed after doing the necessary relaxation. This new acceleration is utilized to compute new joint velocities and positions. It is done by any of the rigidbody solvers which are crank Nicholson, symplectic and newmark as selected in the dictionary. This is performed by the solve function in the following lines:

```
if (Pstream::master())
     {
           solver_->solve(tau, fx);
       }
```

After this forwarddynamics correction is called upon which updates the velocities and accelerations of the bodies and these updated values update the rigidbodymodel. It can be understood better when the knowledge of the algorithm is profound. Finally the status of the rigid body is printed as can be seen with the following lines in `rigidBodyMotion.C`:

```
void Foam::RBD::rigidBodyMotion::status(const label bodyID) const
   {
       const spatialTransform CofR(X0(bodyID));
       const spatialVector vCofR(v(bodyID, Zero));

       Info<< "Rigid-body motion of the " << name(bodyID) << nl
           << "    Centre of rotation: " << CofR.r() << nl
           << "    Orientation: " << CofR.E() << nl
           << "    Linear velocity: " << vCofR.l() << nl
           << "    Angular velocity: " << vCofR.w()
           << endl;
   }
```

These info lines are printed in the terminal window when the simulation is being run.

Chapter 2

Demonstration

2.1 Introduction

The Floating object tutorial will be used as a starting point. The Floating object tutorial can be viewed by navigating to the following location:

```
OF4x
cd $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/floatingObject
```

It can be seen in the `constant` folder that there are two files named `dynamicMeshDict` and `dynamicMeshDict.sixDoF`. The latter contains the settings that are used when using the sixDoFRigidBodyMotion library, while the former file contains the settings which can be used with rigidBodyMeshMotion, which is an addition in OpenFOAM 4x. All the settings will be carried out in `dynamicMeshDict` as this project is dealing with the use of these new additions in OpenFOAM. It can also be observed by looking at the dictionary that the settings are very simple as compared to the sixDoF settings hence making it simpler to use.

2.2 Floating sphere

2.2.1 Case setup introduction

The Floating object tutorial is simulating a cuboid floating on water, constrained by a composite joint (mixture of joints). A volume of water is kept at some height and it is released when the simluation starts, hence creating waves which force the body to move which can be visualized using paraFoam. This tutorial will be modified in which the body will be changed to a sphere (two spheres will be used) and its movement will be restrained using linear springs whose fixed end will be attached to the stationary walls.

2.2.2 Procedure

The first step is changing the `blockMeshDict` to increase the mesh resolution. So the number of cells have been increased in all three directions and also the box has been made larger to accommodate the sphere of radius 1.5. First copy the floatingobject tutorial

```
OF4x
cp -r  $FOAM_TUTORIALS/multiphase/interDyMFoam/ras/floatingObject $FOAM_RUN
cd $FOAM_RUN
mv floatingObject floatingObjectSphere
cd floatingObjectSphere
```

Open blockMeshDict:

```
vi system/blockMeshDict
```

Edit the following in `blockMeshDict`. We have refined the mesh to get better results (divergence occuring at later stage):

```
vertices
(
    (0 0 0)
    (3 0 0)
    (3 3 0)
    (0 3 0)
    (0 0 3)
    (3 0 3)
    (3 3 3)
    (0 3 3)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (50 50 50) simpleGrading (1 1 1)
);
```

The second step involves changing the body from a cuboid to sphere. For this in `topoSetDict` the following changes needs to be made: Open `topoSetDict`:

```
vi system/topoSetDict
```

Edit the following

```
       actions
(
    {
        name    c0;
        type    cellSet;
        action  new;
        source  sphereToCell;//changed from boxtocell
        sourceInfo
        {
       centre   (1.5 1.5 1.5);
       radius   0.3;
        }
    }

    {
        name    c0;
        type    cellSet;
        action  invert;
    }
);
```

It can be noticed that sphereToCell has been used here to make cell set corresponding to a sphere. Since the dimensions of the box has been changed, the `setFieldsDict` needs to be edited to get proper field of water that will generate wave to move the sphere. Open setFieldsDict:

```
vi system/setFieldsDict
```

Make the changes:

```
       regions
(
boxToCell
    {
        box (0 0 0) (3 3 1.5);
        fieldValues ( volScalarFieldValue alpha.water 1 );
    }

    boxToCell
    {
        box (0 0 1.5) (0.5 0.5 2);
        fieldValues ( volScalarFieldValue alpha.water 1 );
    }

);
```

Further `dynamicMeshDict` needs to be edited to make the necessary changes so that it identifies this sphere. Open this dictionary:

```
    vi constant/dynamicMeshDict
```

Edit the following:

```
dynamicFvMesh       dynamicMotionSolverFvMesh;

motionSolverLibs   ("librigidBodyMeshMotion.so");

solver             rigidBodyMotion;

rigidBodyMotionCoeffs
{
    report          on;

    solver
    {
        type Newmark;
    }

    accelerationRelaxation 0.7;

    bodies
    {
        floatingObject
        {
            type            sphere;//changed
            parent          root;

            // sphere dimensions
            radius           0.3;//changed

            // Density of the sphere
            rho             500;

            // sphere mass
            mass            #calc "$rho*4/3*3.14*$radius*$radius*$radius";//changed
            centreOfMass    (1.5 1.5 1.5);//changed
            transform       (1 0 0 0 1 0 0 0 1) (0.5 0.45 0.1);

            joint
            {
                type            composite;
                joints
                (
                    {
                        type Py;
                    }
                    {
                        type Ry;
                    }
                );
            }

            patches         (floatingObject);
            innerDistance   0.05;
            outerDistance   0.35;
        }
    }
}
```

The allrun file need not be touched. The file from floating object will work fine for this case. Finally run the simulation using Allrun command as shown below:

```
    ./Allrun
```

The results can be visualized using paraFoam. Few snippets from the result is shown in figure 2.1 and 2.2.

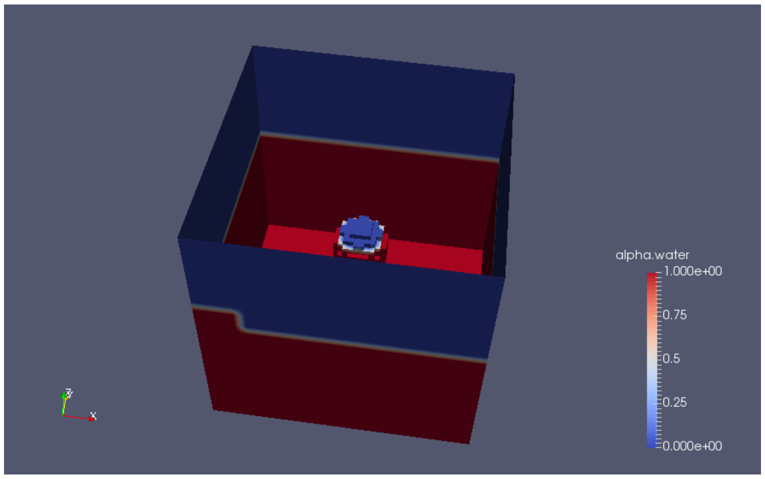

Figure 2.1: Result view- at the start of the simulation

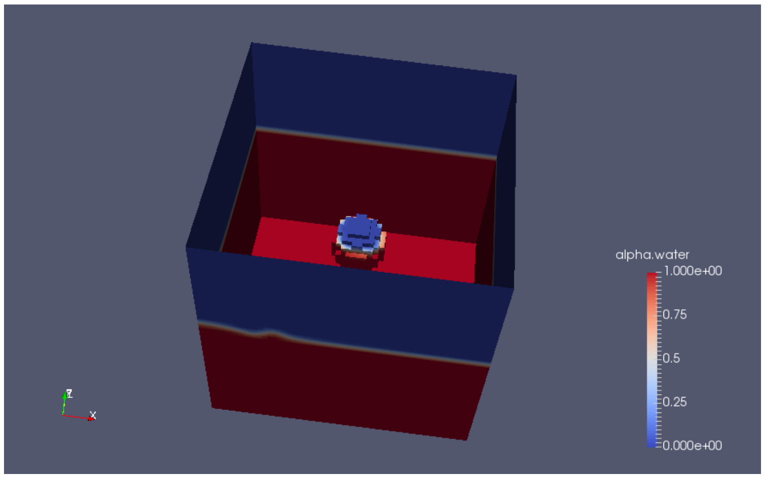

Figure 2.2: Result view- at an intermediate time step during the simulation

2.3 Two floating objects restrained

2.3.1 Case setup introduction

This case will be an extension to the previous case. In this case two bodies are floating on water which will be restrained by a spring to the bottom of the box which contains water and the bodies. It will be run for some time to visualize the movement of the bodies.

2.3.2 Procedure

The first step is changing the `blockMeshDict` to increase the box size to accommodate the two bodies. Since there are two bodies, it needs to be declared in this dictionary First copy the floatingobjectSphere tutorial

```
OF4x
run
cp -r floatingObjectSphere floatingObjectwobody
cd floatingObjectwobody
./Allclean
vi system/blockMeshDict
```

Make the following changes in blockMeshDict:

```
convertToMeters 1;

vertices
(
    (0 0 0)
    (3 0 0)
    (3 3 0)
    (0 3 0)
    (0 0 3)
    (3 0 3)
    (3 3 3)
    (0 3 3)
);

blocks  //changed
(
    hex (0 1 2 3 4 5 6 7) (20 20 30) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    stationaryWalls
    {
        type wall;
        faces
        (
            (0 3 2 1)
            (2 6 5 1)
            (1 5 4 0)
            (3 7 6 2)
            (0 4 7 3)
        );
    }
    atmosphere
    {
        type patch;
        faces
        (
            (4 5 6 7)
        );
    }
    body1//added
    {
        type wall;
        faces ();
    }
    body2//added
    {
        type wall;
        faces ();
    }
);

mergePatchPairs
(
);
```

The number of cells has been decreased because high resolution is not required for this case. Now since this case setup has 2 bodies `topoSetDict` will be used to create the cellSets which will be overwritten with the bodies using subSetMesh in the Allrun file. First topoSetDict needs to be copied:

```
cp topoSetDict topoSetDict2
mv topoSetDict topoSetDict1
vi topoSetDict1
```

To edit the `topoSetDict1` please copy and paste the following lines:

```
actions
(
    {
        name    c1;
        type    cellSet;
        action  new;
        source  boxToCell;//changed
        sourceInfo
        {
            box (0.85 1.4 1.2) (1.15 1.6 1.7);//changed
        }
    }

    {
        name  c1;
        type cellSet;
        action  invert;
    }
);
```

Now further edit `topoSetDict2` with the following lines:

```
actions
(
    {
        name    c2;
        type    cellSet;
        action  new;
        source  boxToCell;//changed
        sourceInfo
        {
            box (1.85 1.4 1.2) (2.15 1.6 1.7);//changed
        }
    }

    {
        name  c2;
        type  cellSet;
        action  invert;
    }
);
```

Since the name of the bodies have been changed to body1 and body 2 the same changes needs to be made in the boundary conditions in `0.orig` folder so that these bodies are recognised and the boundary conditions are implemented while simulation is running. sed command will be used to make the changes as it will make the work very easy rather than opening each and every file to make the changes which is very tedious.

```
cd $FOAM_RUN/floatingObjectwobody/0.orig
sed -i s/floatingObject/body1/g *
sed -i 37r<(sed  '34,37!d'  alpha.water) alpha.water
sed -i 45r<(sed  '38,45!d'  epsilon) epsilon
sed -i 39r<(sed  '35,39!d'  k) k
sed -i 44r<(sed  '37,44!d'  nut) nut
sed -i 37r<(sed  '33,37!d'  pointDisplacement) pointDisplacement
sed -i 35r<(sed  '32,35!d'  p_rgh) p_rgh
sed -i 37r<(sed  '33,37!d'  U) U
sed -i  '0,/body1/! {0,/body1/ s/body1/body2/}'  *
```

Finally, after making these changes we can move on to dynamicMeshDict where the case has to be setup. It will be seen in this dictionary that we add the second body, add the restraints on the two bodies and change the joint to constrain the movement of the bodies only in the z-direction. The center of rotation of the bodies which is given by the transform keyword in the dictionary is irrelevant in this case as the bodies have been constrained to move in the z-direction only, which means there is a prismatic joint in the z-direction and there will be a sliding motion. Open dynamicMeshDict:

```
cd ..
cd constant
vi dynamicMeshDict
```

Copy the below dictionary to dynamicMeshDict.

```
dynamicFvMesh       dynamicMotionSolverFvMesh;
```

```
motionSolverLibs   ("librigidBodyMeshMotion.so");

solver             rigidBodyMotion;

rigidBodyMotionCoeffs
{
    report          on;

    solver
    {
        type Newmark;
    }

    accelerationRelaxation 0.8;//changed

    bodies
    {
        body1 //added
        {
            type            cuboid;
            parent          root;

            // Cuboid dimensions
            Lx              0.3;
            Ly              0.2;
            Lz              0.5;

            // Density of the cuboid
            rho             500;

            // Cuboid mass
            mass            #calc "$rho*$Lx*$Ly*$Lz";
            L               ($Lx $Ly $Lz);
            centreOfMass    (1.5 1.5 1.5);
            transform       (1 0 0 0 1 0 0 0 1) (0.5 0.45 0.1);

            joint
            {
                type            Pz;//changed

            }

            patches         (floatingObject1);
            innerDistance   0.05;
            outerDistance   0.35;

          }
        body2//added
        {
            type            cuboid;
            parent          root;

            // Cuboid dimensions
            Lx              0.3;
            Ly              0.2;
            Lz              0.5;

            // Density of the cuboid
            rho             500;

            // Cuboid mass
            mass            #calc "$rho*$Lx*$Ly*$Lz";
            L               ($Lx $Ly $Lz);
            centreOfMass    (2 1.5 1.5);
            transform       (1 0 0 0 1 0 0 0 1) (0.5 0.45 0.1);

            joint
            {
                type            Pz;//changed

            }

            patches         (floatingObject2);
            innerDistance   0.05;
            outerDistance   0.35;
        }
    }
            restraints //added
            {
               verticalspring1
               {
                body                body1;
                type                linearSpring;
                anchor              (0.6 1.05 0);//immovable point
                refAttachmentPt     (0.6 1.05 1.2);//attachment point on body
                stiffness            100;
                damping              0;
                restLength           0.1;
                }
             verticalspring2
               {
                body                body2;
                type                linearSpring;
                anchor              (1.95 1.05 0);//immovable point
                refAttachmentPt     (1.95 1.05 1.2); //attachment point on body
                stiffness            100;
                damping              0;
                restLength           0.1;
                }
             }
}
```

Finally the Allrun file needs to be edited to include subsetMesh and topoSet for the two bodies. Open Allrun file:

```
cd ..
vi Allrun
```

Make the following changes:

```
runApplication blockMesh
topoSet -dict system/topoSetDict1 //added
subsetMesh -overwrite c1 -patch body1 //added
topoSet -dict system/topoSetDict2 //added
subsetMesh -overwrite c2 -patch body2 //added
cp -r 0.orig 0 > /dev/null 2>&1
runApplication setFields
runApplication $application
```

Now it can be visulaized using paraFoam. Few snippets from the result is shown in figure 2.3 and 2.4.

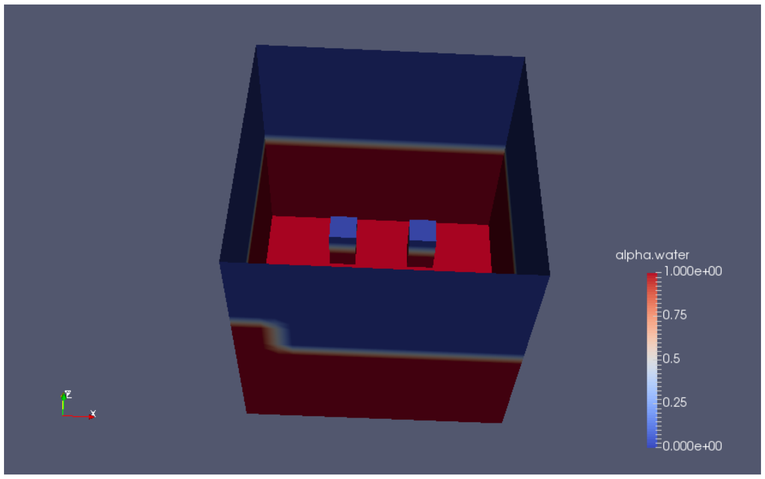

Figure 2.3: Result view-at the start of simulation

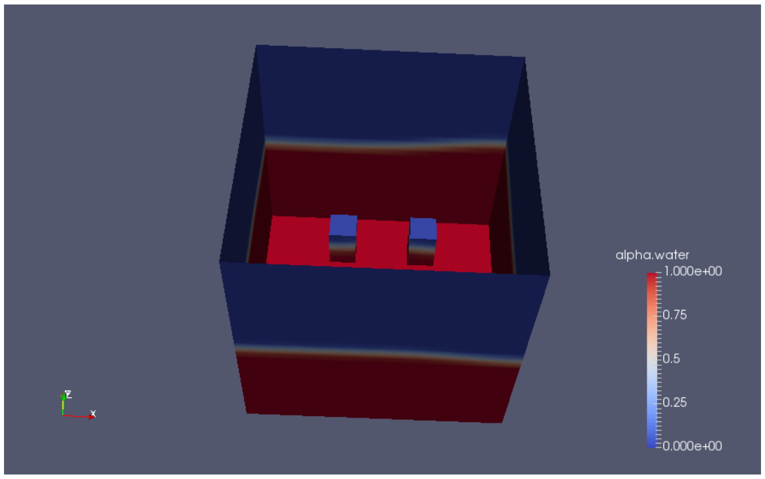

Figure 2.4: Result view-at an intermediate time step during the simulation

Chapter 3

Conclusion and Future Work

This library can be compared with the sixDofRigidBodyMotion library. In the sixDof library, the constraints and restraints were setup in the point displacement file. But in this library everything is done in dynamicMeshDict, which makes it easier to setup the cases. It is more user friendly as everything is at one place. The rigidBodyDynamics library also has some predefined geometries like cuboid and sphere which can be used very easily. For the joints in the sixDof library the constraint axis had to be mentioned which is a bit difficult to visualize. The rigidBody library makes it easy through its list of revolute, prismatic and spherical joints. The algorithm used to solve the problem (articulated body algorithm) is of the least possible order making it quite fast.
The result from the first case (sphere floating body) diverges after some time. This is due to the fact that the sphere is created using topoSetDict(sphere to cell) which makes the sphere as a collection of cubes as can be seen in figure 3.1:

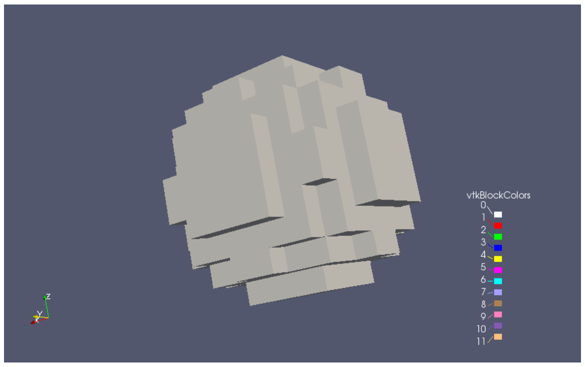

Figure 3.1: Sharp edges of sphere

Due to this the result diverges after some time. When the mesh is refined, it is found that the simulation runs for a longer time before diverging. By refining the mesh, the sphere becomes smoother. It is also observed that for the second case in the demonstration, the result do not diverge. This shows that the tutorial does not work perfectly with spherical body. Some more cases are needed to demonstrate the full potential of this library, especially the cases where bodies are connected to each other through some restraints.

Chapter 4

Study Questions

- **Will the articulated body inertia change in each time step?**
- **Can a cylindrical body be simulated using this library?**
- **Where is meant by the model state at a particular instant of time?**
- **How can the solver used for the simulation be changed?**
- **How can two bodies at some distance be merged together using a massless connector?**

Bibliography

[1] Roy Featherstone. *Rigid Body Dynamics Algorithms.*

[2] Thesis:Brian Vincent Mirtich, *Impulse-based dynamic simulation of rigid body systems*, `http://www.kuffner.org/james/software/dynamics/mirtich/mirtichThesis.pdf`