OpenFOAM Lagrangian Library and its Coupling with Rayleigh-Plesset Equation — project for the course CFD with OpenSource Software

Chuck (Boxiong) Chen

Applied Mechanics/Combustion, Chalmers University of Technology, Gothenburg, Sweden

2013-12-09
About the OpenFOAM framework

OpenFOAM (Open source Field Operation And Manipulation) is a C++ toolbox for the development of customized numerical solver for continuum mechanics.

The beauty of the package is the diversity of the included libraries that could be used to build up solvers for most of the common purposes.

Most of the libraries make good use of advanced features in C++, such as inheritance, template classes, to make the libraries as generalized as possible. This, on one hand, makes the direct using of these libraries extremely convenient as compared to other computational tools, on the other hand give rise to a rather painful experience if you have to modify or write up some rudimental structures. We will come back to this later.
About the lagrangian library

The lagrangian library is a compilation of a variety of lagrangian particle tracking (LPT) libraries. The freedom are given to the user to decide which library to use, depending on what are the variables of interest, e.g.:

- solidParticle library tracks velocity, position, and diameter (which is not updated though).
- intermediate library contains a well developed structure for LPT with different level of complexities.
  - Kinematic cloud and parcel
  - Kinematic with collision
  - Thermal
  - Reacting
  - Reacting and multiphase (which can be used to account for vaporisation)
- spray cloud and parcel contain models for atomization, breakup, and collision process, but it also make use of the intermediate library.
A table for the variables in libraries

So the question is, what to choose?

Ans: Having known the physical meaning of your problem, especially what should be tracked, we can then checkout the .H files for parcel, find out what is registered in the parcel. And checkout the .C files, the subroutines therein to figure out what are the variables being solved and updated (this is usually a tricky part).
A table for the variables in libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Major Variables</th>
<th>Variables being updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>solidParticle</td>
<td>vel., pos., d</td>
<td>vel., pos.</td>
</tr>
<tr>
<td>KinematicParcel</td>
<td>vel., pos., d</td>
<td>vel., pos.</td>
</tr>
<tr>
<td>CollidingParcel</td>
<td>vel., pos., d</td>
<td>vel., pos.</td>
</tr>
<tr>
<td>ThermalParcel</td>
<td>vel., pos., d, and T</td>
<td>Everything but d</td>
</tr>
<tr>
<td>ReactingParcel</td>
<td>vel., pos., d, T, and Y</td>
<td>Everything but d</td>
</tr>
<tr>
<td>ReactingMultiphase</td>
<td>vel., pos., d, T, Y, YLiquid, YSolid</td>
<td>Everything</td>
</tr>
</tbody>
</table>
Description

In fluid mechanics, the Rayleigh Plesset equation is an ordinary differential equation which governs the dynamics of a spherical bubble in an infinite body of liquid. Its general form is usually written as

\[
\frac{P_B(t) - P_\infty(t)}{\rho_L} = R \frac{d^2 R}{dt^2} + \frac{3}{2} \left( \frac{dR}{dt} \right)^2 + \frac{4\nu_L}{R} + \frac{2S}{\rho_L R}
\]

It can be derived from Navier-Stokes equation, with the spherical shape assumption for the dispersed phase, which means, the radius can be solved in a uncoupled manner with gas phase equations.
Solving

The analytical solutions of Rayleigh Plesset equation are only available for restricted specific cases, so numerical solution will be pursued. An ODE library is available in OpenFOAM. Three different methods, namely "RK4", "KRR4", and "SIBS" are implemented. Previous study reported that the compressible Rayleigh Plesset equation suffers from numerical stiffness, so compressibility will not be considered in the current work.
C++ and Object Oriented Programming

What is object oriented programming (OOP)?

In the traditional procedural programming language e.g. Fortran, C, procedures are generally more important/powerful than data. Data are less complex, and fewer logic are involved (more logics are expressed in the procedures that manipulate the code). Whereas in C++, data, especially user defined ones, like structures, classes, are more complicated. Inheritance allows us to distribute procedures into each level of the derived data.

- Procedural code passes data along the main stream that is made up by procedures.
- In object oriented program (OOP), data (classes) pass procedures to the base data (classes), the procedures are dissected and spread in data hierarchy.
Procedures

- Adding LPT library into the continuous phase solver (e.g. piso, simple, pimple, ...)
- Change the solving variables, in my case diameter into radius and radius growth rate
- Make a stand-alone solver for Rayleigh Plesset equation, starting from test ODE case, using as many constant parameters as possible to make it independent.
- Reform it into a class, include the class into the kinematicParcel/solidParticle
- Get the continuous phase interpolated local variable transferred into the ODE
- Setup the communication between the ODE and the top level KinematicCloud/SolidParticleCloud (TOUGH JOB!)
- Remove all the constants in ODE and get the values from KinematicCloud/SolidParticleCloud
**Lagrangian Particle Tracking, main frame**

In all the LPT libraries of OpenFOAM, the dispersed phase are described by two major structure: Cloud and Parcel. Cloud is a list of parcels plus some general properties for every parcel (note the Cloud here is not relevant to the template class Cloud, which will be covered later).

In the simplest case, solidParticleCloud is a list of solidParticle. Another example being KinematicCloud with KinematicParcel.

Two additional structure worth notice: particle and Cloud. They ’re both defined in the lagrangian/basic library and being applied in all the LPT libraries. Particle is a class that contains position (and it’s tracked in any library); Cloud is a list of parcel.
Lagrangian Particle Tracking, how it works(1)

The process starts from a call to the top level cloud constructor, which is, in my case, in the createField.H file in the solver package:

```cpp
basicKinematicRPCloud kinematicCloud
(
    kinematicCloudName,
    rho,
    U,
    mu,
    p,
    g
);
```
Lagrangian Particle Tracking, how it works(2)

The constructor of top level cloud is usually called in the continuous phase solver before the solving loop is started, I personally think it’s fair to compare the constructing procedure with the initialization process of procedural CFD code.

Next, we would start to appreciate the OOP style by checking out how this constructor passes the procedure down to its base classes. In myIntermediate/clouds/Templates/KinematicRPCloud:

```
CloudType(rho.mesh(), cloudName, false),
```

Through the application of template class, the constructor of KinematicRPCloud passes the procedure to its subsequent level class, the class being represented by template parameter CloudType, which is, in the current case, `Cloud<KinematicRPParcel.ParcelType>`.
Lagrangian Particle Tracking, how it works(3)

CloudType(rho.mesh(), cloudName, false) would, in turn, passes the constructing procedure to parcels through a loop. In myBasic/Cloud/CloudIO.C

template<class ParticleType>
Foam::Cloud<ParticleType>::Cloud
(
    const polyMesh& pMesh,
    const word& cloudName,
    const bool checkClass
)
:

To be continued in the next slide ...
Lagrangian Particle Tracking, how it works(4)

```cpp
cloud(pMesh, cloudName),
polyMesh_(pMesh),
labels_(),
nTrackingRescues_(),
cellWallFacesPtr_()
{
    initCloud(checkClass);
}
```

And here comes a question, where is the ”stream” going to?
Lagrangian Particle Tracking, how it works(5)

Could it be \texttt{cloud(pMesh, cloudName)}, or any of the constructors? No!
Check where this constructor starts from:

\begin{verbatim}
template<class ParticleType>
Foam::Cloud<ParticleType>::Cloud
\end{verbatim}

Whatever the next step is, it gotta operate on the template parameter. But there is no constructor that makes use of \texttt{ParticleType}. So ...?
Lagrangian Particle Tracking, how it works

```cpp
{
    initCloud(checkClass);
}
```

is the answer. initCloud is a function of `Cloud<ParticleType>` class, so it has access to `ParticleType` without the need of `ParticleType` reference or pointer. Through the application of a loop, the stream then goes to the contractor of each `ParticleType` reference. The procedure can be traced down and down through `KinematicRPPParcel<ParcelType>` in a similar manner, we would not go through all of that.
A Brief for ODERP

A good place to start with is really the Test-ODE.C, in which the main function declare and construct an object of ODE (which literally does nothing!) Then it starts solving by loading the initial value, then repeatively calls the derivative function, and solve for each time step.

We change the derivative function and jacobian to integrate the Rayleigh Plesset equation. It is self-evident that the number of equations has to be 2, in that the RP equation is a second order ODE.

Once the result is validated, the name of the ODE should be changed, in my case, into ODERP. Then an object is included in the ”parcel” classes.

Implementation details of ODE will be included in a dedicated manual, which will hopefully be finished some time next year.
Warming Up

We have briefly walked through the classes and their hierarchy. In `myIntermediate/clouds/derived/basicKinematicRPCloud/basicKinematicRPCloud.H`, `KinematicRPCloud<Cloud<basicKinematicRPParcel> >` is renamed as `basicKinematicRPCloud`, using `typedef`.

`basicKinematicRPParcel` is in turn a renamed class of `KinematicRPParcel<particle>`, and this is done in `myIntermediate/parcels/derived/basicKinematicRPParcel/basicKinematicRPParcel.H`. Note that the `ParcelType` parameter in `KinematicRPParcel` is filled up with class `particle`. 
Templated ODERP Class (1)

We have had a few words on ODERP as well, the current implementation of ODERP in KinematicCloud borrowed many of the ideas from a continuous phase uncoupled ODERP implementation on solidParticleCloud. But ODERP is one part of the work that has to take care of the huge differences between KinematicCloud and solidParticleCloud.

Check this out:
(In the next slide)
Templated ODERP Class(2)

class solidParticle :
    public particle
{
    ...

template<class ParcelType>
class KinematicRPPParcel :
    public ParcelType
{
    ...

See the problem here?
**Templated ODERP Class(3)**

It’s the template again!

Since in some constant properties are needed in ODERP, which is one of the base classes of `KinematicRPCloud<...>`, a reference or pointer of the top level cloud would be needed.

**DOWNCASTING** is the solution.

What it means?

Downcasting: converting a base-class pointer (reference) to a derived-class pointer (reference) is called downcasting.

How is it done? (Don’t creep out...)
Templated ODERP Class(4)

```cpp
static_cast<
    const KinematicRPCloud
    <
        Cloud
        <
            KinematicRPPParcel
            <
                ParcelType
                >
            >
        >
    >
> (cloud)
```
Templated ODERP Class(5)

That’s it for the brief on ODERP and its communication of basicKinematicRPCloud.

For more details, you’re welcomed to checkout the tutorial file, or the even more detailed manual which I hope could be finished soon, or reach me through email at: chenb@chalmers.se
That's it

Questions?
Thank you!