Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.
Chapter 1

Tutorial

1.1 Introduction

This tutorial describes the OpenFOAM course project on the coupling of a lagrangian particle tracking library/intermediate with the piso flow solver, and the setup and coupling of an ordinary differential equations (ODE) solver for Rayleigh Plesset equation to the particle for simulations on cavitation phenomena. In this tutorial, an elaborated description on the logic behind the implementation will be given. Hopefully it provide an informative answer to the question "How to program step by step towards the final implementation in OpenFOAM framework".

In this tutorial we will try to address the following five major problems:

- How to change an ODE test case solver into a ODE Rayleigh Plesset solver?
- How should we couple a lagrangian/intermediate solver to the continuous phase solver?
- How shall we modify it further to facilitate its coupling with ODE solver later?
- How shall we change the ODE stand alone solver into a class?
- How to couple it with intermediate library and setup its communication with other classes?

1.2 ODE Solver

This section covers the necessary implementation details to adapt the test case ODE solver into a stand-alone ODE solver for ODE Rayleigh Plesset (which will be denoted as ODERP in the following text) equation solver.

1.2.1 Rayleigh-Plesset equation

The fundamentals about Rayleigh Plesset equation can be found in many previous literatures. It is an ordinary differential equation which governs the dynamics of a spherical bubble in an infinite body of liquid. The Rayleigh-Plesset equation is derived from the Navier-Stokes equations under the assumption of spherical symmetry. Neglecting surface tension and viscosity, the equation was first derived by John Strutt, 3rd Baron Rayleigh in 1917. The equation was first applied to traveling cavitation bubbles by Milton S. Plesset in 1949. A general form is given as follow:

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

No general analytical solution is available yet for the Rayleigh-Plesset equation. However, numerical
solutions to any accuracy can be easily obtained. In the current implementation we will make use of the ODE numerical solving algorithm KRR4.

1.2.2 Test Solver

We will now contribute our effort to the implementation side of the story.

An ODE library is available in OpenFOAM. Three different methods, namely "RK4", "KRR4", and "SIBS" are implemented, but none of them seems to be capable of dealing with stiff system. Previous study reported that the compressible Rayleigh Plesset equation suffers from numerical stiffness, so compressibility will not be considered in the current work.

We start dealing with the stand-alone ODE part by checking the ODE test case. The ODE test case is located in verbatim /OpenFOAM/OpenFOAM-2.0.1/applications/test/ODE

Copy the ODE test case to the run directory.

```
cp -r $FOAM_APP/test/ODE $WM_PROJECT_USER_DIR/application
cd $WM_PROJECT_USER_DIR/application/ODE
```

Since the Rayleigh Plesset equation is a second order ODE, which can be transformed into a first order ODE equation set with 2 equations. So in `Test-ODE.C`, we change the `nEqns` function's return into 2.

The derivatives and the jacobian function have to be adapted into the Rayleigh Plesset equation. To make it more specific, from now on the ODE class name will be switched to ODERP.

```c
void Foam::ODERP::derivatives
{
    const scalar x,
    const scalarField& y,
    scalarField& dydx
)
) const
{
    //Water properties
    scalar rhoL=1000;
    scalar muL=0.001;
    scalar sigma=0.07;
    scalar p0=101e3;
    scalar pL=p0;
    //Bubble properties
    scalar R0=10e-06;
    scalar kappa=1.4;
    if (y[0] >= R0) {kappa=1.;}
    scalar pv=2330;
    scalar pB=(p0+2*sigma/R0-pv)*Foam::pow(R0/y[0],3*kappa)+pv;

    dydx[0] = y[1] ;
    dydx[1] = (-3/2*sqr(y[1])+pB-pL)/rhoL/y[0]
              -(4*muL*y[1]+2*sigma)/(rhoL*sqr(y[0]));
}
```
The properties, pressure, and surface tension above are right now simply "hard coded" into constant values. Later on the properties will be obtained from the dictionary in constant directory, and the pressure should be taken from the gas phase value.

```cpp
void Foam::ODERP::jacobian
(
    const scalar x,
    const scalarField& y,
    scalarField& dfdx,
    scalarSquareMatrix& dfdy
) const
{
    // Water properties
    scalar rhoL=1000;
    scalar muL=0.001;
    scalar sigma=0.07;
    scalar p0=101e3;
    scalar pL=p0;

    // Bubble properties
    scalar R0=10e-06;
    scalar kappa=1.4;
    if (y[0] >= R0) {kappa=1. ;}
    scalar pv=2330;
    scalar pB=(p0+2*sigma/R0-pv)*Foam::pow(R0/y[0],3*kappa)+pv;

dfdx[0] = 0.0;
    dfdx[1] = 0.0;
    dfdy[0][0] = 0.0;
    dfdy[0][1] = 1.0;
    dfdy[1][0] = (-3/2*sqr(y[1])-(pB-pL)/rhoL)/sqr(y[0])
        - (8*muL*y[1]+4*sigma)/(rhoL*pow(y[0],3));
    dfdy[1][1] = -3*y[1] / y[0] - 4*muL / (rhoL*sqr(y[0]));
}
```

Make the changes in the calling function accordingly, then compile and test the solver. At the current stage, since it is merely an intermediate implementation, we would not include any more discussion here. The final implementation of calling function is available in the last section of the tutorial.

### 1.2.3 Break it up

The test-ODE.C solver contains all the functionality, but in order to make ODE into a ODERP which fits into later implementation, the main function is yanked out and put into a separated solveRP.H, which will later be included in the parcel solver.

At this stage, it is not strictly necessary to consider how to further dissect the ODERP class file, but for the sake of convenience for the coupling, it’s better to group all the data of ODERP class into a dedicated ODERP.H file. Since we don’t get many member functions for ODERP type, and considering the fact that ODERP will later be upgraded into a template class, which wouldn’t allow for an forward declaration when compile, all the functions will be included in a ODERPI.H file as inline functions. We will come back to this later.
1.3 Lagrangian Libraries

1.3.1 A brief about lagrangian tracking libraries in OpenFOAM

Many literatures could be found for a brief introduction to the lagrangian tracking, so we would not cover it here. The lagrangian library in OpenFOAM 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, and no injector).
- 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 summarization of the included scalars and solving scalars is given as follow.

<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>

Table 1.1: Summary for the OpenFOAM lagrangian library

1.3.2 Intermediate Library

As can be seen from the table above, a variety of parcels and corresponding clouds files are available to account for different tracking scalars. Since the purpose of our current work is restricted to merely account for velocity, position, and radius changes, solidParticle or KinematicParcel should be sufficient for it.

A previous implementation based on solidParticle has been finished by Aurelia. The convenience in implementation of the solidParticle is quite attractive to us. However, making use of the existing
models, specifically, the injection part would become important to our project later, so it is decided that a KinematicParcel and Cloud implementation is pursued.

There is still another complexity yet to be considered. None of the particle libraries here includes radius changing rate. In Aurelia’s implementation, changing rate was included in the ODERP in addition to radius and the diameter in the original parcel was then replaced by an object of ODERP. In the current implementation, the diameter was changed into radius, and the radius changing rate is included as well, which corresponds to the first order derivative of \( y \) in the ODE solver.

Before delving into the implementation details, it is worthy of mentioning that in both parcel and cloud directories, `Templates` directories are the ones contains the most details on solving. `derived` directory, on the other hand, is merely a renaming of the parcel or cloud in order to avoid writing the long name involved in the nested templates. In `KinematicParcel.H`, protected data declaration part, we replace the member diameter with:

```cpp
//-- Radius [m]
scalar radius_;

//-- Growth rate of radius [m/s]
scalar radiusGrowthRate_;
```

Accordingly, the functions that returns the protected data has to be replaced by,

```cpp
//-- Return const access to radius
inline scalar radius() const;

//-- Return const access to radius growth rate
inline scalar radiusGrowthRate() const;
```

It also need changes in the function that returns the access as well,

```cpp
//-- Return access to radius
inline scalar& radius();

//-- Return access to radius growth rate
inline scalar& radiusGrowthRate();
```

And the functions that use diameters would need replacement as well,

```cpp
//-- Reynolds number
inline scalar Re
(
    const vector& U,    // particle velocity
    const scalar radius, // particle radius
    const scalar rhoc,   // carrier density
    const scalar muc     // carrier dynamic viscosity
) const;

//-- Weber number
inline scalar We
(
    const vector& U,    // particle velocity
    const scalar radius, // particle radius
    const scalar rhoc,   // carrier density
    const scalar sigma   // particle surface tension
) const;
```
The change that we have made on the inline functions call for changes in the corresponding `KinematicParcelI.H` file. Many other functions in `KinematicParcel.C` needs to be changed as well. Besides the functions, there're five constructors in `KinematicParcel.C`, `KinematicParcelI.H`, and `KinematicParcelIO.C` that also call for modifications. We include all of them in the appendix.

### 1.3.3 Weld Them Together

Here, we will briefly check how ODERP should be coupled with KinematicParcel.

**Issues Regarding ODERP**

In `KinematicRPParcel.H`, we firstly include the ODERP definition:

```cpp
#include "ODERP.H"
```

And include an object of template ODERP class in the protected data:

```cpp
protected:
    // Protected data
    ODERP<ParcelType> oderp_;  // Parcel properties
    // Active flag - tracking inactive when active = false
    bool active_;  
```

Apparently, a significant amount of editing work has to be done for `oderp_` data in the constructors of `KinematicRPParcel`.

As the coupling between phases requires ODERP class to obtain data from the KinematicRP-Cloud (which is initialized by the kinematicCloudProperties dictionary), we need to reconstruct the reference of the cloud in the ODERP here. Below we included two constructors of ODERP.

```cpp
template<class ParcelType>
inline Foam::ODERP<ParcelType>::ODERP
(
    const Cloud<KinematicRPParcel<ParcelType> >& cloud
)
:
    cloud_  
    {  
        static_cast
        <  
            const KinematicRPCloud  
            <  
                Cloud  
                <  
                    KinematicRPParcel  
                    <  
                        ParcelType
```
1.3. LAGRANGIAN LIBRARIES  

```cpp
// Member functions
template<class ParcelType>
inline void Foam::ODERP<ParcelType>::setEnvironment
(
    const scalar& pL,
    const scalar& rhoL,
    const scalar& muL
)
{
    pL_ = pL;
    rhoL_ = rhoL;
    muL_ = muL;
}
```

The function `setEnvironment` passes the interpolated continuous phase properties into the dispersed phase Rayleigh Plesset integration. More details relevant to the coupling can be found in the next subsection.

```cpp
// Member functions
template<class ParcelType>
inline void Foam::ODERP<ParcelType>::derivatives (......)
......
scalar kappa=cloud_.gamma();
    if (y[0] >= R0) {kappa=1.;}
    scalar pb= (cloud_.p0()+2*cloud_.sigma()/R0-cloud_.pv())
        *Foam::pow(R0/y[0],3*kappa)+cloud_.pv();
    dydx[0] = y[1];
    dydx[1] = (-3/2*sqr(y[1])+(pb-pL_)/rhoL_)/y[0]
    -(4*muL_*y[1]+2*cloud_.sigma())/(rhoL_*sqr(y[0]));
```

Some similar modification has been done in the jacobian function as well, which would not be elaborated here.

Now that the reference has been casted into reference of KinematicRPCloud, the surface tension and vapor pressure could now be introduced into the ODE integration of Rayleigh Plesset.
Solving

One thing that definitely worth mentioning is how we insert the "Rayleigh Plesset equation". All the
nested class templates that use KinematicCloud would finally pass the process to the move function
in KinematicParcel.C for the position tracking. Therefore, in the cases when additional tracking
scalar is desired (and they are not dealt with in any other class templates), it’s highly convenient to
add tracking functions in the move function.

In the move function (where all the amazing happens) in KinematicRPPParcel.C,

template<class ParcelType>
template<class TrackData>
bool Foam::KinematicRPPParcel<ParcelType>::move
(
    TrackData& td,
    const scalar trackTime
)
......
if (dt > ROOTVSMALL)
{
    // Update cell based properties
    p.setCellValues(td, dt, cellI);

    if (td.cloud().solution().cellValueSourceCorrection())
    {
        p.cellValueSourceCorrection(td, dt, cellI);
    }

    p.calc(td, dt, cellI);
}
#include "solveRP.H"
if (p.onBoundary() && td.keepParticle)
{
    if (isA<processorPolyPatch>(pbMesh[p.patch(p.face())]))
    {
        td.switchProcessor = true;
    }
}

Next, I think it’s necessary to have a few words about the solveRP.H, part of the story has been
covered in the stand-alone ODE solver section, but solveRP.H is the part that takes care of the
coupling between phases, and the ODE solving procedure.

    label cellB = mesh.findCell(p.position());
    vector posB = p.position();
    const scalar& rhoL = td.rhoInterp().interpolate(posB, cellB);
    const scalar& muL = td.muInterp().interpolate(posB, cellB);
    const scalar& pL = td.pInterp().interpolate(posB, cellB);
    oderp_.setEnvironment(pL, rhoL, muL);

As can be seen, the last fews lines coupled the continuous phase with the dispersed phase. The
location and the cell that contains the bubble are first located, then they are used to get a local
value from the surrounding points’ values through interpolation. These interpolated values will then
be used in the Rayleigh Plesset ODE sets (namely the jacobian and derivative functions of ODERP class template). For the sake of readability, brief and scattered comments instead of a chunk of gathered discussion will be inserted into the rest of the solverRP.H file.

    // Initiate the new solver.
    autoPtr<ODESolver> solveRP = ODESolver::New("KRR4", oderp_);

    // Start time.
    scalar xStart = mesh.time().value();

    // Declare the initial value.
    scalarField yStart(oderp_.nEqns());

    // Copy from the ODERP.
    yStart[0] = this->radius();
    yStart[1] = this->radiusGrowthRate();

    scalarField dyStart(oderp_.nEqns());
    int numIter = dt/1e-5;

    for (int i=0; i<numIter; i++)
    {
        oderp_.derivatives(xStart, yStart, dyStart);

        scalar x = xStart;
        scalar xEnd = x + 1.0e-05;
        scalarField y(yStart);
        scalar hEst = 1.0e-10;

        solveRP -> solve(oderp_, x, xEnd, y, 1e-4, hEst);

        if((!(i%100000))
        {
            Info<< xEnd << " " << y[0] <<" " << y[1] << endl;
        }

        yStart[0] = y[0];
        yStart[1] = y[1];
        xStart = xEnd;
    }

    // Feed the integrated value back to the ODERP object.
    this->setRadius(yStart[0]);
    this->setRadiusGrowthRate(yStart[1]);

    And that completes the brief discussion on the solver.
Chapter 2

Test Case

2.1 Configuration

The configuration was set up starting from a configuration similar to the 2D square case in icoFoam, it can then easily be adapted into a 3D by assigning a mesh in the 3rd direction and changing patches for boundary conditions.

To generate a strong pressure gradient, a configuration that consisting of a cube (0.1*0.1*0.1) and a cuboid (0.02*0.1*0.1) is given here. The decrease of cross sectional area in the middle would greatly increase the velocity (by approximately 5 times in the current case), meanwhile creating a strong pressure drop, which is crucial for the bubble radius growth.

2.2 Boundary Conditions

The original moving wall patch will be removed, inlet and outlet will be added as patch type in the blockMeshDict file.

```
boundary
{
    inlet
    {
        type patch;
        faces
        {
            (3 7 6 2)
            (2 6 11 9)
            (9 11 15 13)
        };
    }
    outlet
    {
        type patch;
        faces
        {
            (16 17 19 18)
        };
    }
    fixedWalls
    {
```
type wall;
faces
{
    (0 4 7 3)
    (1 5 4 0)
    (5 1 16 18)
    (15 14 12 13)
    (14 10 8 12)
    (10 19 17 8)
    (0 3 2 1)
    (1 2 9 8)
    (8 9 13 12)
    (16 1 8 17)
    (4 5 6 7)
    (5 10 11 6)
    (10 14 15 11)
    (5 18 19 10)
};
}

The velocity and pressure can be assigned accordingly to the patches in the boundary condition files in 0 directory. Since we’re interested in the k-epsilon modeling of turbulence. With a presumed an estimation for turbulence injection and dissipation can be given following the procedure similar to P-53 of OpenFOAM programmer’s guide.
Chapter 3

Appendix

This section includes the major changes inside the KinematicParcel directory. For the sake of convenience, we rename KinematicParcel and KinematicCloud as KinematicRPParcel and KinematicRPCloud, respectively to specify the changes we made concerning Rayleigh Plesset integration. Correspondingly, the name changes should be applied to everything in conjunction with them.

In KinematicParcel.C,

```cpp
template<class ParcelType>
template<class TrackData>
void Foam::KinematicRPParcel<ParcelType>::calc
(
    TrackData& td,
    const scalar dt,
    const label cellI
)
.....
    // Reynolds number
    const scalar Re = this->Re(U_, radius_, rhoc_, muc_);
```

```cpp
template<class ParcelType>
Foam::KinematicRPParcel<ParcelType>::KinematicRPParcel
(
    const KinematicRPParcel<ParcelType>& p
)
.....
    radius_(p.radius_),
    radiusGrowthRate_(p.radiusGrowthRate_),
```

```cpp
template<class ParcelType>
Foam::KinematicRPParcel<ParcelType>::KinematicRPParcel
(
    const KinematicRPParcel<ParcelType>& p,
    const polyMesh& mesh
)
.....
    radius_(p.radius_),
    radiusGrowthRate_(p.radiusGrowthRate_),
```

```cpp
template<class ParcelType>
```
Foam::scalar Foam::KinematicRPParcel<ParcelType>::wallImpactDistance
(const vector&
) const
......
return radius_;
{  
    return radius_;  
}

template<class ParcelType>
inline Foam::scalar& Foam::KinematicRPParcel<ParcelType>::radiusGrowthRate()
{
    return radiusGrowthRate_;  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::momentOfInertia() const
{
    return 0.1*mass()*sqr(2.0*radius_);  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::volume() const
{
    return volume(radius_);  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::volume(const scalar radius)
{
    return pi/6.0*pow3(2.0*radius);  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::areaP() const
{
    return areaP(radius_);  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::areaP(const scalar radius)
{
    return 0.25*areaS(radius);  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::areaS() const
{
    return areaS(radius_);  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::areaS(const scalar radius)
{
    return pi*2.0*radius*2.0*radius;  
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::Re

( const vector& U,
    const scalar radius,
    const scalar rhoc,
    const scalar muc
) const
{
    return rhoc*mag(U - Uc_)*2.0*radius/(muc + ROOTVSMALL);
}

template<class ParcelType>
inline Foam::scalar Foam::KinematicRPParcel<ParcelType>::We
( const vector& U,
    const scalar radius,
    const scalar rhoc,
    const scalar sigma
) const
{
    return rhoc*magSqr(U - Uc_)*2.0*radius/(sigma + ROOTVSMALL);
}

In KinematicParcelIO.C.

template<class ParcelType>
Foam::string Foam::KinematicRPParcel<ParcelType>::propHeader =
    ParcelType::propHeader
+ " active"
+ " typeId"
+ " nParticle"
+ " radius"
+ " radiusGrowthRate"
......
    radius_(0.0),
    radiusGrowthRate_(0.0),

template<class ParcelType>
Foam::KinematicRPParcel<ParcelType>::KinematicRPParcel
( const polyMesh& mesh,
    Istream& is,
    bool readFields
)
......
{
    if (readFields)
    {
        if (is.format() == IOstream::ASCII)
        {
            active_ = readBool(is);
            typeId_ = readLabel(is);
            nParticle_ = readScalar(is);
            radius_ = readScalar(is);
            radiusGrowthRate_ = readScalar(is);
......
else
{
    is.read
    (  
        reinterpret_cast<char*>(active_),
        sizeof(active_)
        + sizeof(typeId_)
        + sizeof(nParticle_)
        + sizeof(radius_)
        + sizeof(radiusGrowthRate_)
    ......

template<class ParcelType>
template<class CloudType>
void Foam::KinematicRPParcel<ParcelType>::readFields(CloudType& c)
......
    IOField<scalar> radius(c.fieldIOobject("radius", IOobject::MUST_READ));
    c.checkFieldIOobject(c, radius);

    IOField<scalar> radiusGrowthRate(c.fieldIOobject("radiusGrowthRate", IOobject::MUST_READ));
    c.checkFieldIOobject(c, radiusGrowthRate);

......
    forAllIter(typename CloudType, c, iter)
    {
        KinematicRPParcel<ParcelType>& p = iter();

        p.active_ = active[i];
        p.typeId_ = typeId[i];
        p.nParticle_ = nParticle[i];
        p.radius_ = radius[i];
        p.radiusGrowthRate_ = radiusGrowthRate[i];

......

template<class ParcelType>
template<class CloudType>
void Foam::KinematicRPParcel<ParcelType>::writeFields(const CloudType& c)
......
    IOField<scalar> radius(c.fieldIOobject("radius", IOobject::NO_READ), np);
    IOField<scalar> radiusGrowthRate(c.fieldIOobject("radiusGrowthRate", IOobject::NO_READ), np);

......
    forAllConstIter(typename CloudType, c, iter)
    {
        const KinematicRPParcel<ParcelType>& p = iter();

        active[i] = p.active();
        typeId[i] = p.typeId();
        nParticle[i] = p.nParticle();
        radius[i] = p.radius();
        radiusGrowthRate[i] = p.radiusGrowthRate();

......
        active.write();
        typeId.write();
        nParticle.write();
radius.write();
radiusGrowthRate.write();

......

template<class ParcelType>
Foam::Ostream& Foam::operator<<(Ostream& os,
const KinematicRPParcel<ParcelType>& p)

......
if (os.format() == IOstream::ASCII)
{
    os << static_cast<const ParcelType&>(p)
    << token::SPACE << p.active()
    << token::SPACE << p.typeId()
    << token::SPACE << p.nParticle()
    << token::SPACE << p.radius()
    << token::SPACE << p.radiusGrowthRate()

......
else
{
    os << static_cast<const ParcelType&>(p);
    os.write(
        reinterpret_cast<const char*>(
p.active_)
        ,
        sizeof(p.active())
        + sizeof(p.typeId())
        + sizeof(p.nParticle())
        + sizeof(p.radius())
        + sizeof(p.radiusGrowthRate())
        ......