Disclaimer: 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 reviewer has no responsibility for the contents.

This document is just for the explaination of the sprayFoam. and it is based on OpenFOAM-8.

since there are some changes in the afterwards version

First, this is the original sprayFoam solver

```
#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "momentumTransportModel.H"
#include "fluidThermophysicalTransportModel.H"
#include "psiReactionThermophysicalTransportModel.H"
#include "basicSprayCloud.H"
#include "psiReactionThermo.H"
#include "CombustionModel.H"
#include "radiationModel.H"
#include "SLGThermo.H"
#include "pimpleControl.H"
#include "CorrectPhi.H"
#include "fvOptions.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "createDyMControls.H"
    #include "createFields.H"
    #include "createFieldRefs.H"
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"
    #include "initContinuityErrs.H"
    #include "createRhoUfIfPresent.H"

    turbulence->validate();


    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
Info<< "\nStarting time loop\n" << endl;

//- check the PISO or PIMPLE mode
while (pimple.run(runTime))
{
    #include "readDyMControls.H"

    // Store divrhoU from the previous mesh so that it can be mapped
    // and used in correctPhi to ensure the corrected phi has the
    // same divergence
    autoPtr<volScalarField> divrhoU;
    if (correctPhi)
    {
        divrhoU = new volScalarField
        (
            "divrhoU",
            fvc::div(fvc::absolute(phi, rho, U))
        );
    }

    #include "compressibleCourantNo.H"
    #include "setDeltaT.H"

    runTime++;

    Info<< "Time = " << runTime.timeName() << nl << endl;

    // Store momentum to set rhoUf for introduced faces.
    autoPtr<volVectorField> rhoU;
    if (rhoUf.valid())
    {
        rhoU = new volVectorField("rhoU", rho*U);
    }

    // Store the particle positions
    parcels.storeGlobalPositions();

    // Do any mesh changes
    mesh.update();

    if (mesh.changing())
    {
        MRF.update();

        if (correctPhi)
        {
            // Calculate absolute flux from the mapped surface velocity
            phi = mesh.Sf() & rhoUf();
```

```
            #include "correctPhi.H"

            // Make the fluxes relative to the mesh-motion
            fvc::makeRelative(phi, rho, U);
        }

        if (checkMeshCourantNo)
        {
            #include "meshCourantNo.H"
        }
    }

    // parcels.evolve is the starting point of inserting sprayParcel to the domain
    parcels.evolve();

    #include "rhoEqn.H"

    // --- Pressure-velocity PIMPLE corrector loop
    while (pimple.loop())
    {

        #include "UEqn.H"
        #include "YEqn.H"
        #include "EEqn.H"

        // --- Pressure corrector loop
        while (pimple.correct())
        {
            #include "pEqn.H"
        }

        if (pimple.turbCorr())
        {
            turbulence->correct();
            thermophysicalTransport->correct();
        }
    }

    rho = thermo.rho();

    runTime.write();

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << "  ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}

Info<< "End\n" << endl;
```

```
    return 0;
}
```

```
// UEqn.H
// Solve the Momentum equation

MRF.correctBoundaryVelocity(U);

tmp<fvVectorMatrix> tUEqn
(
    fvm::ddt(rho, U) + fvm::div(phi, U)
  + MRF.DDt(rho, U)
  + turbulence->divDevTau(U)
 ==
    rho()*g
  + parcels.SU(U)                       // This is the most important part for the
lagrangian                                      //source term exchange in U eqn
                                        // it is the same for the Energy Equation etc.
  + fvOptions(rho, U)
);
fvVectorMatrix& UEqn = tUEqn.ref();

UEqn.relax();

fvOptions.constrain(UEqn);

if (pimple.momentumPredictor())
{
    solve(UEqn == -fvc::grad(p));

    fvOptions.correct(U);
    K = 0.5*magSqr(U);
}
```

```
//- createFields.H
//- Include the gravity effect
#include "readGravitationalAcceleration.H"

//- reading thermo/species properties
Info<< "Reading thermophysical properties\n" << endl;
autoPtr<psiReactionThermo> pThermo(psiReactionThermo::New(mesh));
psiReactionThermo& thermo = pThermo();
thermo.validate(args.executable(), "h", "e");

SLGThermo slgThermo(mesh, thermo);
```

```
basicSpecieMixture& composition = thermo.composition();
PtrList<volScalarField>& Y = composition.Y();

const word inertSpecie(thermo.lookup("inertSpecie"));
if (!composition.species().found(inertSpecie))
{
    FatalIOErrorIn(args.executable().c_str(), thermo)
        << "Inert specie " << inertSpecie << " not found in available species "
        << composition.species()
        << exit(FatalIOError);
}

//- pressure field
volScalarField& p = thermo.p();

//- creating the fluid gas density field
volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT,
        IOobject::AUTO_WRITE
    ),
    thermo.rho()
);

//- creating the fluid Velocity field
Info<< "\nReading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

#include "compressibleCreatePhi.H"

mesh.setFluxRequired(p.name());

dimensionedScalar rhoMax
```

```
(
    dimensionedScalar::lookupOrDefault
    (
        "rhoMax",
        pimple.dict(),
        dimDensity,
        great
    )
);

dimensionedScalar rhoMin
(
    dimensionedScalar::lookupOrDefault
    (
        "rhoMin",
        pimple.dict(),
        dimDensity,
        0
    )
);

//- Turbulence
Info<< "Creating turbulence model\n" << endl;
autoPtr<compressible::momentumTransportModel> turbulence
(
    compressible::momentumTransportModel::New
    (
        rho,
        U,
        phi,
        thermo
    )
);


Info<< "Creating thermophysical transport model\n" << endl;
autoPtr<psiReactionThermophysicalTransportModel> thermophysicalTransport
(
    psiReactionThermophysicalTransportModel::New(turbulence(), thermo)
);

Info<< "Creating combustion model\n" << endl;
autoPtr<CombustionModel<psiReactionThermo>> combustion
(
    CombustionModel<psiReactionThermo>::New(thermo, turbulence())
);

Info<< "Creating field dpdt\n" << endl;
volScalarField dpdt
```

```
(
    IOobject
    (
        "dpdt",
        runTime.timeName(),
        mesh
    ),
    mesh,
    dimensionedScalar(p.dimensions()/dimTime, 0)
);




//- creating the turbulence kinetic energy
Info<< "Creating field kinetic energy K\n" << endl;
volScalarField K("K", 0.5*magSqr(U));
multivariateSurfaceInterpolationScheme<scalar>::fieldTable fields;

forAll(Y, i)
{
    fields.add(Y[i]);
}
fields.add(thermo.he());

#include "createMRF.H"

//- createClouds.H is the part of lagrangian calcualtion
#include "createClouds.H"



#include "createRadiationModel.H"
#include "createFvOptions.H"
```

```
//- CreateClouds.H
//- this means what properties are going to the parcels calculation

Info<< "\nConstructing reacting cloud" << endl;
basicSprayCloud parcels
(
    "sprayCloud",
    rho,
    U,
    g,
    slgThermo
);
```

After we create the spray clouds and the transfer the require parameters to the cloud, the calcualtion of lagrangian particles is started.

```
//- OpenFOAM-8/src/lagrangian/spray/clouds/derived/basicSprayCloud/basicSprayCloud.H
//- the defination of spraycloud.
//- in OpenFOAM, it is just layers by layers from clouds to parcels.
//- Cloud is just a collection of parcels

namespace Foam
{
    typedef SprayCloud
    <
        ReactingCloud
        <
            ThermoCloud
            <
                KinematicCloud
                <
                    Cloud
                    <
                        basicSprayParcel
                    >
                >
            >
        >
    > basicSprayCloud;
}
```

```
namespace Foam
{
    typedef SprayParcel
    <
        ReactingParcel
        <
            ThermoParcel
            <
                KinematicParcel
                <
                    particle
                >
            >
        >
    > basicSprayParcel;

    template<>
    inline bool contiguous<basicSprayParcel>()
```

```
    {
        return false;
    }
}
```

```
//- since the layers setup of parcels in the OpenFOAM, the changes of the other parcels would have effect on the other parcel layers as well

//- OpenFOAM-8/src/lagrangian/spray/parcels/Templates/SprayParcel/SprayParcel.C
//- This function called calc is the most important part in the sprayParcel and the other parcels as well
//- In the sprayParcel, it contains the updating of the thermo properties, Atomization model, and primary breakup model etc.



template<class ParcelType>
template<class TrackCloudType>
void Foam::SprayParcel<ParcelType>::calc
(
    TrackCloudType& cloud,
    trackingData& td,
    const scalar dt
)
{
    typedef typename TrackCloudType::reactingCloudType reactingCloudType;
    const CompositionModel<reactingCloudType>& composition =
        cloud.composition();

    // Check if parcel belongs to liquid core
    if (liquidCore() > 0.5)
    {
        // Liquid core parcels should not experience coupled forces
        cloud.forces().setCalcCoupled(false);
    }

    // Get old mixture composition
    scalarField X0(composition.liquids().X(this->Y()));

    // Check if we have critical or boiling conditions
    scalar TMax = composition.liquids().Tc(X0);
    const scalar T0 = this->T();
    const scalar pc0 = td.pc();
    if (composition.liquids().pv(pc0, T0, X0) >= pc0*0.999)
    {
        // Set TMax to boiling temperature
```

```
        TMax = composition.liquids().pvInvert(pc0, X0);
    }

    // Set the maximum temperature limit
    cloud.constProps().setTMax(TMax);

    // Store the parcel properties
    this->Cp() = composition.liquids().Cp(pc0, T0, X0);
    sigma_ = composition.liquids().sigma(pc0, T0, X0);
    const scalar rho0 = composition.liquids().rho(pc0, T0, X0);
    this->rho() = rho0;
    const scalar mass0 = this->mass();
    mu_ = composition.liquids().mu(pc0, T0, X0);

    ParcelType::calc(cloud,td, dt);

    if (td.keepParticle)
    {
        // Reduce the stripped parcel mass due to evaporation
        // assuming the number of particles remains unchanged
        this->ms() -= this->ms()*(mass0 - this->mass())/mass0;

        // Update Cp, sigma, density and diameter due to change in temperature
        // and/or composition
        scalar T1 = this->T();
        scalarField X1(composition.liquids().X(this->Y()));
        this->Cp() = composition.liquids().Cp(td.pc(), T1, X1);
        sigma_ = composition.liquids().sigma(td.pc(), T1, X1);
        scalar rho1 = composition.liquids().rho(td.pc(), T1, X1);
        this->rho() = rho1;
        mu_ = composition.liquids().mu(td.pc(), T1, X1);
        scalar d1 = this->d()*cbrt(rho0/rho1);
        this->d() = d1;
        if (liquidCore() > 0.5)
        {
            //- calculation the atomization model in spray
            calcAtomization(cloud, td, dt);
            // Preserve the total mass/volume by increasing the number of
            // particles in parcels due to breakup
            scalar d2 = this->d();
            this->nParticle() *= pow3(d1/d2);
        }
        else
        {
            //- calculation the primay breakup model
            calcBreakup(cloud, td, dt);
        }
    }
```

```
    // Restore coupled forces
    cloud.forces().setCalcCoupled(true);
}
```

```
//- OpenFOAM
8/src/lagrangian/intermediate/parcels/Templates/KinematicParcel/KinematicParcel.C
//- The Kinematic Parcel is the root of all the parcels which is related with the
parcel movement, its name is changes after OpenFOAM8 to MomentumParcel.
//- And After OpenFOAM4, the parcel location is changed from global coordinates into
barycentric coordinates. This change gives a lot of performance improvement, but it is
not good for postprocessing.


template<class ParcelType>
template<class TrackCloudType>
void Foam::KinematicParcel<ParcelType>::calc
(
    TrackCloudType& cloud,
    trackingData& td,
    const scalar dt
)
{

    // Define local properties at beginning of time step
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    const scalar np0 = nParticle_;
    const scalar mass0 = mass();

    // Reynolds number
    //- the calculation of Re is defined in the KinematicParcelI.H
    const scalar Re = this->Re(td);
    // Sources
    //~~~~~~~~
    //- Su is a vetor and Spu is a scalar in the forceSuSp.H
    //- F = Sp(U - Up) + Su
    //-Explicit contribution, Su specified as a force
    //-Implicit coefficient, Sp specified as force/velocity

    // Explicit momentum source for particle
    vector Su = Zero;
    // Linearised momentum source coefficient
    scalar Spu = 0.0;

    // Momentum transfer from the particle to the carrier phase
    vector dUTrans = Zero;
```

```
    // Motion
    // ~~~~~~

    // Calculate new particle velocity
    //- this will goes to calculate the parcel velocity
    this->U_ =
        calcVelocity(cloud, td, dt, Re, td.muc(), mass0, Su, dUTrans, Spu);




    // Accumulate carrier phase source terms
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if (cloud.solution().coupled())
    {
        // Update momentum transfer
        cloud.UTrans()[this->cell()] += np0*dUTrans;

        // Update momentum transfer coefficient
        cloud.UCoeff()[this->cell()] += np0*Spu;
    }
}



// * * * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //

template<class ParcelType>
template<class TrackCloudType>
bool Foam::KinematicParcel<ParcelType>::move
(
    TrackCloudType& cloud,
    trackingData& td,
    const scalar trackTime
)
{
    typename TrackCloudType::parcelType& p =
        static_cast<typename TrackCloudType::parcelType&>(*this);
    typename TrackCloudType::parcelType::trackingData& ttd =
        static_cast<typename TrackCloudType::parcelType::trackingData&>(td);

    ttd.switchProcessor = false;
    ttd.keepParticle = true;

    //- control the Lagrangian time step dt by using the p.stepFraction()
```

```
    const scalarField& cellLengthScale = cloud.cellLengthScale();
    const scalar maxCo = cloud.solution().maxCo();

    while (ttd.keepParticle && !ttd.switchProcessor && p.stepFraction() < 1)
    {
        // Cache the current position, cell and step-fraction
        const point start = p.position();
        const scalar sfrac = p.stepFraction();

        // Total displacement over the time-step
        const vector s = trackTime*U_;

        // Cell length scale
        const scalar l = cellLengthScale[p.cell()];

        // Deviation from the mesh centre for reduced-D cases
        const vector d = p.deviationFromMeshCentre();

        // Fraction of the displacement to track in this loop. This is limited
        // to ensure that the both the time and distance tracked is less than
        // maxCo times the total value.
        scalar f = 1 - p.stepFraction();
        f = min(f, maxCo);
        f = min(f, maxCo*l/max(small*l, mag(s)));
        if (p.active())
        {
            // Track to the next face
            p.trackToFace(f*s - d, f);
        }
        else
        {
            // At present the only thing that sets active_ to false is a stick
            // wall interaction. We want the position of the particle to remain
            // the same relative to the face that it is on. The local
            // coordinates therefore do not change. We still advance in time and
            // perform the relevant interactions with the fixed particle.
            p.stepFraction() += f;
        }

        const scalar dt = (p.stepFraction() - sfrac)*trackTime;

        // Avoid problems with extremely small timesteps
        if (dt > rootVSmall)
        {
            // Update cell based properties
            p.setCellValues(cloud, ttd);

            //- update the instantaneous fluid velocity that parcel can see.
```

```
            //- this is working with the RAS turbulence model because the velocity
field is time averaged, not the instantaneous fluid velocity. The dispersion model is
related with turbulence related properties.

            //- the calcualtion of dispersionmodel is in the OpenFOAM-
8/src/lagrangian/turbulence/submodels/Kinematic/DispersionModel/StochasticDispersionRAS
/StochasticDispersionRAS.C

            p.calcDispersion(cloud, ttd, dt);

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

            p.calc(cloud, ttd, dt);
        }

        p.age() += dt;

        if (p.active() && p.onFace())
        {
            cloud.functions().postFace(p, ttd.keepParticle);
        }

        //- run the cloud function postMove function
        cloud.functions().postMove(p, dt, start, ttd.keepParticle);

        if (p.active() && p.onFace() && ttd.keepParticle)
        {
            p.hitFace(f*s - d, f, cloud, ttd);
        }
    }

    return ttd.keepParticle;
}
```

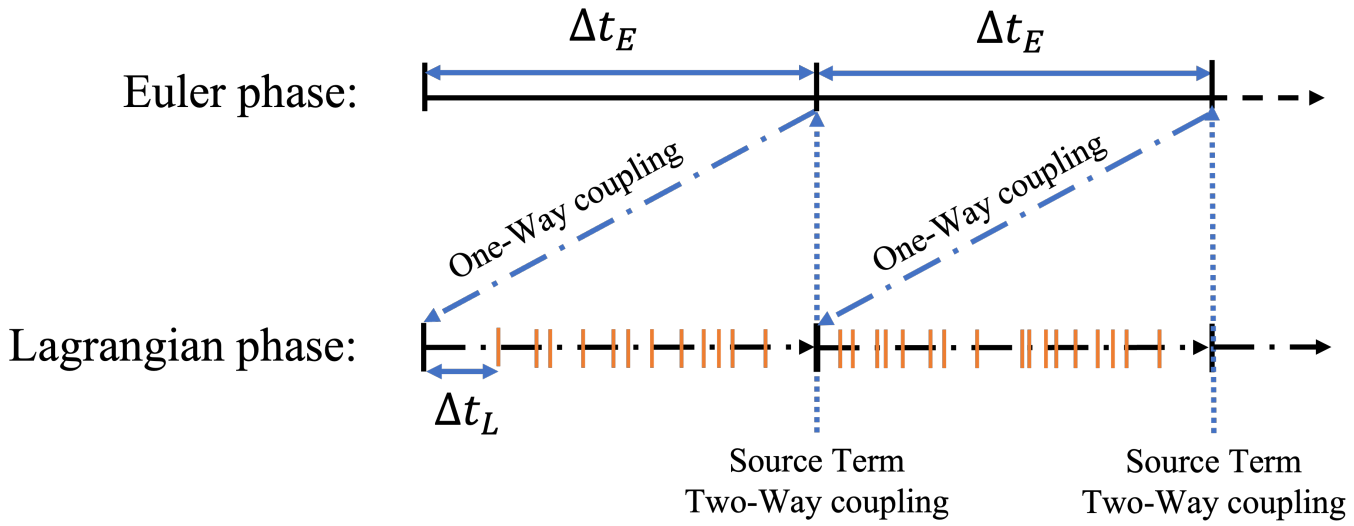
In the following figure, it explain the relationship of Euler Time step and Lagrange Time step.

After solving Euler phase in Euler time step, the calculation of lagrangian phase will start which contains several lagrangian time step dt, which is controled by the p.stepFraction() in each parcel. and the feedback effect is related to one-way coupling and two-way coupling or even four-way.

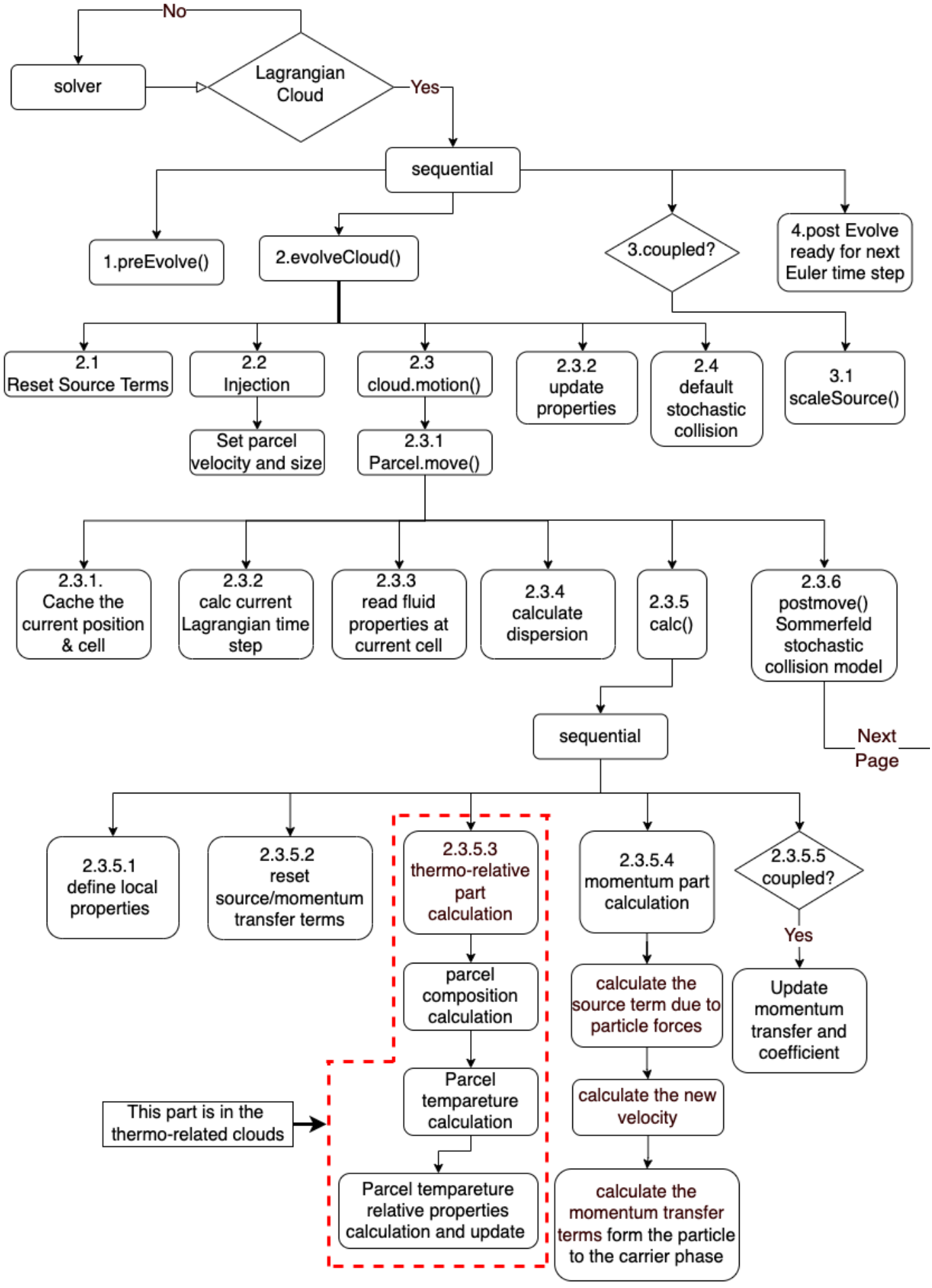
one-way coupling: only parcel can see the surrounding fluid properties and get influenced

two-way coupling: parcel would also feedback to the fluid.

four-way coupling: it contains the parcel-parcel effect, like parcel-parcel collision.



This is the flow chart of the Lagrangian calculation. if you want to change something, please check the part of the code.


No
solver
Lagrangian Cloud
Yes
sequential
1.preEvolve()
2.evolveCloud()
3.coupled?
4.post Evolve ready for next Euler time step
2.1 Reset Source Terms
2.2 Injection
2.3 cloud.motion()
2.3.2 update properties
2.4 default stochastic collision
3.1 scaleSource()
Set parcel velocity and size
2.3.1 Parcel.move()
2.3.1. Cache the current position & cell
2.3.2 calc current Lagrangian time step
2.3.3 read fluid properties at current cell
2.3.4 calculate dispersion
2.3.5 calc()
2.3.6 postmove() Sommerfeld stochastic collision model
Next Page
sequential
2.3.5.1 define local properties
2.3.5.2 reset source/momentum transfer terms
2.3.5.3 thermo-relative part calculation
2.3.5.4 momentum part calculation
2.3.5.5 coupled?
Yes
parcel composition calculation
Parcel tempareture calculation
Parcel tempareture relative properties calculation and update
This part is in the thermo-related clouds
calculate the source term due to particle forces
calculate the new velocity
calculate the momentum transfer terms form the particle to the carrier phase
Update momentum transfer and coefficient