Software Design for Particles in Incompressible Flow

Dan Martin and Phil Colella
Applied Numerical Algorithms Group

June 7, 2004

1 Overview

The addition of particles to the existing Incompressible Navier-Stokes code will primarily involve the addition of the forcing function due to the particles ($\mathcal{P}\vec{f}$) to the computation of the provisional velocity field $\vec{u}^*$. This will involve two main additions to the code:

1. The particles themselves will need to be added to the code, in the form of a `LevelData<BinFab<DragParticle> >`, where the `DragParticle` class is our application-specific derivative of the base `BinItem` class.

2. A `ParticleProjector` class will be added to compute an approximation to $\mathcal{P}\vec{f}$ to use as a source term for the velocity update.

In addition to the `ParticleProjector` and `DragParticle` classes mentioned above, we will also define a `discreteDeltaFn` class to encapsulate the discrete δ-function δ_ϵ.

We may also find it useful to define a MLC-solver class to encapsulate the MLC algorithm.

A basic diagram of the class relationships between the `Chombo` and `AMRINS`-particles classes is depicted in Figure 1.

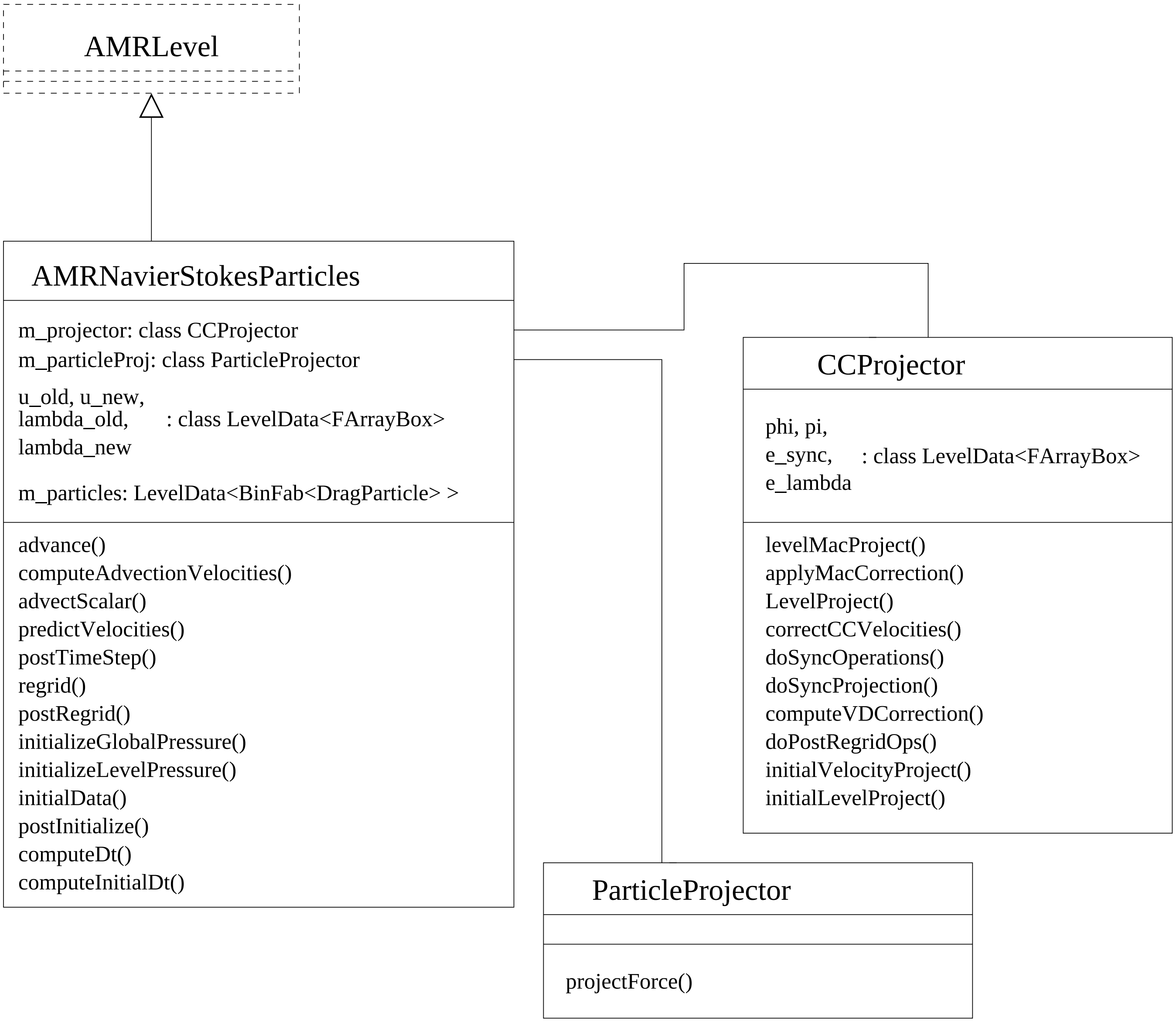

Figure 1: Software configuration diagram for the AMRINS particle code showing basic relationships between AMRINS-particle code classes and Chombo classes

2 Class Outline

2.1 The DiscreteDeltaFn class

The `DiscreteDeltaFn` class will encapsulate the discrete $\delta-$function used to spread the force to the mesh. This class also can compute auxiliary quantities which are functions of the numerical definition of δ_ϵ used.

- `DiscreteDeltaFn* clone() const`
  Create a clone of this DiscreteDeltaFn, with the same properties.

- `Real evaluateDelta(Real a_radius)`
  Evaluate the discrete δ-function $\delta_\epsilon(\boldsymbol{r})$.

- `Real integralDelta(Real a_radius)`
  Returns the integral of δ_ϵ, also known as Q:

  $$Q(r) = \int_0^r \delta_\epsilon(s) s^{D-1} ds,$$

  where D is `SpaceDim`.

- `Real computeK(RealVect a_radius, int a_idir, int a_jdir)`
  Returns the kernel $K_{ij}(\boldsymbol{r})$.

- `Real computeLapDelta(Real a_radius)`
  Returns $\Delta\delta_\epsilon(r)$. Not sure if I really need this, but it's included for completeness at the moment.

- ```
  void sumForce(FArrayBox& a_sum,
                const RealVect& a_force,
                const Box& a_box,
                const RealVect& a_position,
                const RealVect& a_origin,
                Real a_dx)
  ```

  Computes $\sum_j f_j K_{ij}$ over `box` and places it in `a_sum`, which is a `SpaceDim`-component `FArrayBox`. More efficient than calling `computeK` on a cell-by-cell basis.

- ```
  void sumForce(FArrayBox& a_sum,
                const RealVect& a_force,
  ```

```
const Box& a_box,
int a_dir,
int a_destComp,
const RealVect& a_position,
const RealVect& a_origin,
Real a_dx)
```

Computes $\sum_j f_j K_{ij}$ over `box` and places it in `a_sum`, which is a `SpaceDim`-component `FArrayBox`. More efficient than calling `computeK` on a cell-by-cell basis; this version computes a single (`a_dir`) component and places it in the `a_destComp` component of `a_sum`.

2.2 The `PolynomialDelta:public DiscreteDeltaFn` class

Derived class which instantiates the `DiscreteDeltaFn` class using a Polynomial.

2.3 The `DragParticle` class

The `DragParticle` class will encapsulate the definition of the particles used for ths application. It contains a `DiscreteDeltaFn` object to specify the spreading function. Data members include $\vec{f}$, the force vector for the particle, and the position, velocity, and mass of the particle, $\boldsymbol{x}$, $\vec{u}$, and m along with the local fluid velocity and a body force (weight).

Functions include the following:

- ```
  DragParticle(const RealVect& a_position,
               const RealVect& a_velocity,
               const DiscreteDeltaFn* a_deltaFnPtr)
  ```
  Full constructor.

- `void setVel(RealVect& a_vel)`
  sets the velocity field $\vec{u}^{(k)}$ of the particle

- `void setFluidVel(const RealVect& a_vel)`
  sets the local fluid velocity

- `void setBodyForce(const RealVect& a_force)`
  Sets the body force (due to gravity, for example) which is added to any computed drag force.
  ```

- `setMass(Real a_mass)`
  sets the mass of the particle.

- `void updatePosition(RealVect& a_position)`
  updates the position $\boldsymbol{x}^{(k)}$ of the particle.

- `void computeDragForce(RealVect& a_flowVelocity)`
  computes the drag force $\vec{f}^{(k)}$ based on the flow velocity and the particle velocity.

- `Real computeK(RealVect a_x, int a_idir, int a_jdir)`
  computes $K_{ij}^{(k)}(x)$.

- `Real computeProjForce(RealVect a_x, int a_idir)`
  returns
  $$\sum_{j=0}^{D-1} f_{drag,j}^{(k)} K_{ij}^{(k)}(x)$$
  for this particle. Note that this computes the drag force exerted *by* the particle on the surrounding fluid.

- ```
  void computeProjForce(FArrayBox& a_force,
                        const Box& a_box,
                        Real a_dx,
                        RealVect& a_origin) const
  ```
  increments `a_force` over `a_box` with $\sum_{j=0}^{D-1} f_{drag,j}^{(k)} K_{ij}^{(k)}(x)$ for this particle. More efficient than calling the pointwise version of this function. Note that this computes the drag force exerted *by* the particle on the surrounding fluid.

- `RealVect totalForce() const`
  returns the total force on the particle

- `RealVect dragForce() const`
  returns the drag force on the particle

- `RealVect bodyForce() const`
  returns the body force on the particle.

- `Real mass() const`
  returns the mass of the particle.
  ```

- `DragParticle* clone() const`
  creates a clone of this particle, with the same properties (drag coefficient, discrete delta function, mass) as this one.

2.4 The `ParticleProjector` class

The `ParticleProjector` class encapsulates the functionality needed to take the individual forces in the particles and apply them to the mesh in an approximation to $\mathcal{P}\vec{f}$, which may then be used as a source term for the Navier-Stokes advance.

Public Functions:

- ```
  void define(const DisjointBoxLayout& a_grids,
              const DisjointBoxLayout& a_crseGrids,
              const ProblemDomain& a_domain,
              int a_nRefCrse, Real a_dx)
  ```

  Defines class object.

- ```
  void projectForce(LevelData<FArrayBox>& a_force,
                    LevelData<BinFab<DragParticle> >& a_particles)
  ```

  Given the collection of `DragParticles` in `a_particles`, returns the projection of the force at cell centers in `a_force`, suitable for use as a source term for the INS advance.

- `void setSpreadingRadius(const Real a_rad)`
  Sets spreading radius for MLC part of algorithm

- `void setCorrectionRadius(const Real a_rad)`
  Sets correction radius for MLC part of algorithm.

Protected Functions:

- ```
  void computeD(LevelData<FArrayBox>& a_D,
                LevelData<BinFab<DragParticle> >& a_particles)
  ```

- ```
  void solveForProjForce(LevelData<FArrayBox>& a_projectedForce,
                         LevelData<FArrayBox>& a_D,
                         const LevelData<BinFab<DragParticle> >& a_particles)
  ```

- ```
  void addImageEffects(FArrayBox& a_rhs,
                       int a_buffer,
                       int a_dir) const
  ```

- ```
  void doInfiniteDomainSolve(FArrayBox& a_phi,
                             const FArrayBox& a_rhs) const
  ```