Software Design for Particles in Incompressible Flow, non-subcycled case

Dan Martin and Phil Colella
Applied Numerical Algorithms Group

February 8, 2005

1 Overview

To implement an AMR incompressible Navier-Stokes with particles algorithm, we have decided to use a non-subcycled algorithm to simplify the implementation of the particle drag forcing term. This requires a fairly broad redesign of the software from what was presented in [1], since we will no longer be using the `AMR/AMRLevel` base classes to manage the AMR hierarchy. The new classes map on to the functionality of the classes in the original design in a fairly straightforward way, as illustrated in Table 1. The new `PAmrNS` class takes on the functionality of the `ParticleAMRNS class` in the earlier implementation, along with the functionality of the `AMR` and `AmrLevel` classes in the `Chombo AMRTimeDependent` library. The new `AmrProjector` class replaces the original `CCProjector` class, while the new `AMRParticleProjector` class replaces the original `ParticleProjector` class.

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

The `PAmrNS` class will manage the AMR hierarchy and the non-subcycled advance. The non-subcycled advance is much simpler than the subcycled case, both in terms of algorithmic complexity (no need for synchronization projections, etc) and in terms of software implementation.

The `AMRParticleProjector` will do the particle projection originally implemented in the `ParticleProjector` class on an AMR hierarchy, including all image particle effects. The rest of the implementation (`DragParticle`, etc) will be the same as in the original software design.

Old Implementation	New Implementation
ParticleAMRNS AMR AmrLevel	PAmrNS
CCProjector	AmrProjector
ParticleProjector	AMRParticleProjector

Table 1: Mapping of functionality of classes in new implementation to those in the original Navier-Stokes with particles implementation.

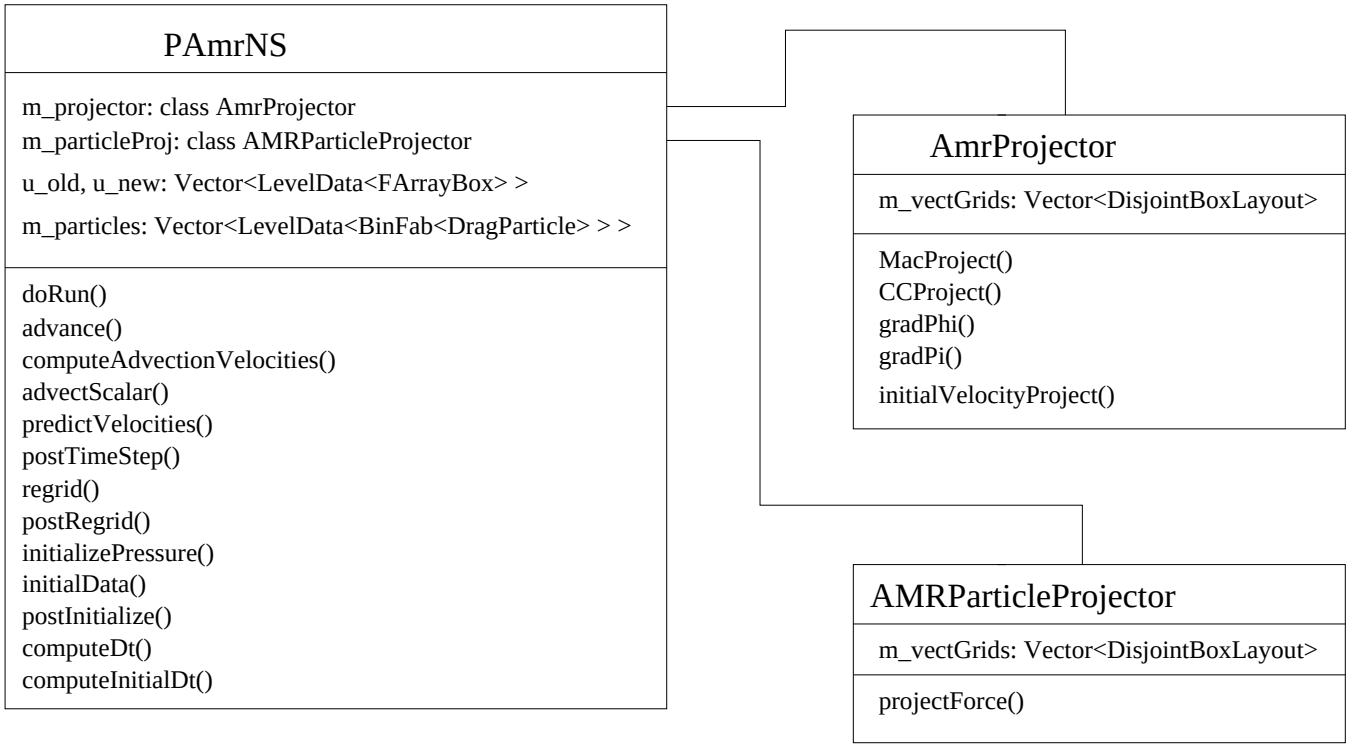

Figure 1: Software configuration diagram for the AMRINS particle code showing basic relationships between AMRINS-particle code classes

Since much of the functionality and internal storage in the `CCProjector` class in the original `AMRINS` code is devoted to subcycling-related functionality, the `AmrCCProjector` is a stripped-down version of the `CCProjector` which only contains the functionality needed to do the multilevel cell-centered and face-centered projections.

2 Class Outline

2.1 The `AMRParticleProjector` class

The `AMRParticleProjector` 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. Note that the particles and projected force may be on different grid hierarchies; in parallel this allows the use of different grid distributions which are load balanced for particles and for the fluid (the force will generally be on the same grids as the fluid).

Public Functions:

- ```
  void define(const Vector<DisjointBoxLayout>& a_vectGrids,
              const Vector<DisjointBoxLayout>& a_vectParticleGrids,
              const Vector<ProblemDomain>& a_vectDomain,
              const Vector<int>& a_vectRefRatio,
              const Vector<Real>& a_vectDx,
              const Vector<LevelData<BinFab<DragParticle> >* >& a_vectParticles
              Real a_spreadingRadius,
              Real a_correctionRadius = 2,
              int a_gridsGrow=0);
  ```

  Defines class object.

  - `a_vectGrids` – grid hierarchy for projected particle drag force.
  - `a_vectParticleGrids` – grid hierarchy which contains particles.
  - `a_vectDomain` – Problem domains for grid hierarchy.
  - `a_vectRefRatio` – refinement ratios.
  - `a_vectDx` – cell spacings.
  - `a_vectParticles` – current particles (needed to generate grown grids).
  ```

- a_spreadingRadius
- a_correctionRadius
- a_gridsGrow – amount of padding to add to grown grids to account for particle motion between regridding steps.

- void projectForce(Vector<LevelData<FArrayBox>* >& a_force,
  const Vector<LevelData<BinFab<DragParticle> >* >& a_particl

  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(Vector<LevelData<FArrayBox>* >& a_D,
  const LevelData<BinFab<DragParticle> >* >& a_particles)

- void solveForProjForce(Vector<LevelData<FArrayBox>* >& a_projectedForce,
  const Vector<LevelData<FArrayBox>* >& a_D,
  int a_lbase);

- void defineImages(List<DragParticle>& a_imageParticles,
  const Vector<LevelData<BinFab<DragParticle> >* >& a_particl

  void defineGrownGrids(const Vector<DisjointBoxLayout>& a_grids,
  const Vector<LevelData<BinFab<DragParticle> >* >& a_vec

2.2 The PAmrNS class

This PAmrNS class manages the AMR hierarchy and the non-subcycled advance of the solution. In terms of functionality, it combines the functionality of the AMR and AMRLevel base classes, but with much simplified functionality due to the non-subcycled advance.

Public Functions:

- `PAmrNS(const ProblemDomain& a_baseDomain,`
  `const Vector<int>& a_vectRefRatio,`
  `const RealVect& a_domainLength,`
  `int a_maxLevel)`

  Full constructor (calls matching define function).

  - `a_baseDomain` – problem domain for the coarsest level in the AMR hierarchy
  - `a_vectRefRatio` – refinement ratios for the AMR hierarchy
  - `a_domainLength` – size of the physical domain.
  - `a_maxLevel` – finest allowable level in the AMR hierarchy

- `void setUpForFixedHiearchyRun(Vector<Vector<Box> > a_vectGrids)`

  Set up grids, initialize data, etc for fixed hierachy run

  - `a_vectGrids` – boxes for AMR grid hierarchy

- `void setUpForAmrRun()`

  Set up grids, initialize data, etc for AMR run

- `void ~PAmrNS()`

  destructor

- `Real doRun(Real a_maxTime, int a_maxStep)`

  advance solution – returns time of final solution.

  - `a_maxTime` – maximum time to advance solution to
  - `a_maxStep` – maximum number of steps to advance solution

- `int writeCheckpointFile(const string& a_fname)`

  writes checkpoint file suitable for restarting. Returns status code (0 == OK).

- `int writeCheckpointFile(HDF5Handle& a_handle)`

  writes checkpoint file for restarting

- `int writePlotFile()`

  write plotfile using default filename construction.

- `int writePlotFileName(const string& a_fname)`
- `int writePlotFileHDF5(HDF5Handle& a_handle)`

Protected Functions:

- `Real advance(Real a_dt);`
- `void postTimeStep();`
- `void tagCells(Vector<IntVectSet>& a_tags);`
- `void tagCellsLevel(IntVectSet& a_tags);`
- `void tagCellsInit(IntVectSet& a_tags);`
- `void regrid(const Vector<Vector<Box> >& a_new_grids);`
- `void postRegrid();`
- `void initialGrid(const Vector<Box>& a_new_grids);`
- `void initialData();`
- `void postInitialize();`
- `void writeParticleData(HDF5Handle& a_handle) const;`
- `int writeParticlesHDF5(HDF5Handle& a_handle,`
  `const List<DragParticle>& a_particles) const;`
- `Real computeDt();`
- `Real computeInitialDt();`
- `void CFL(Real a_cfl);`
- `void particleCFL(Real a_particle_cfl);`
- `void refinementThreshold(Real a_refine_threshold);`
- `void limitSolverCoarsening(bool a_limitSolverCoarsening);`
- `int finestLevel() const;`

- `int maxLevel() const;`
- `bool isEmpty(a_level) const;`
- `void computeVorticity(LevelData<FArrayBox>& a_vorticity, int a_level) const;`
- `void computeKineticEnergy(LevelData<FArrayBox>& a_energy,`
  `int a_level) const;`
- `Real particleKineticEnergy() const;`
- `void dumpParticlesASCII(std::ostream& os ) const;`

2.3 The `AmrProjector` class

The `AmrProjector` class manages the enforcement of the divergence constraint for both cell-centered and face-centered (MAC) velocities. The class also differs from the original `CCProjector` class in that it owns no data itself. This simplifies its design considerably.

Public Functions:

- `AmrProjector(const Vector<DisjointBoxLayout>& a_vectGrids,`
  `const Vector<ProblemDomain>& a_vectDomain,`
  `const Vector<Real>&  a_vectDx,`
  `const Vector<int>& a_vectRefRatio,`
  `int a_lbase,`
  `const PhysBCUtil& a_physBC)`

  Full constructor – calls matching define function.

  - `a_vectGrids` – AMR grid hierachy
  - `a_vectDomain` – problem domains
  - `a_vectDx` – cell spacings for the AMR hierarchy
  - `a_vectRefRatio` – refinement ratios
  - `a_lbase` – base level
  - `a_physBC` – object containing physical BC info.

- `void regrid(const Vector<DisjointBoxLayout>& a_newGrids)`

  Redefine grid hierarchy for projection after regridding

- `void init(const AmrProjector& a_oldProj)`

  Initialize new projection with data from old projection

- ```
  void MacProject(Vector<LevelData<FluxBox>* > & a_uEdge,
                  Vector<LevelData<FArrayBox>* >& a_vectPhi,
                  Real a_oldTime, Real a_dt)
  ```

  Do a multilevel face-centered projection of uEdge.

  - `a_uEdge` – face-centered velocities.
  - `a_vectPhi` – correction field generated by the projection.
  - `a_oldTime` – time at the beginning of the timestep.
  - `a_dt` – timestep used for advance.

  ```
  void CCProject(Vector<LevelData<FArrayBox>* >& a_velocity,
                 Vector<LevelData<FArrayBox>* >& a_vectPi,
                 const Real a_newTime, const Real a_dt)
  ```

  Performs a multilevel cell-centered projection of `a_velocity`.

  - `a_velocity` – cell-centered velocity field.
  - `a_vectPi` – correction field (pressure) generated by the projection.
  - `a_newTime` – time at the end of the timestep.
  - `a_dt` – timestep.

- `void initialVelocityProject(Vector<LevelData<FArrayBox>* >& a_velocity)`

  Project `a_velocity` using a multilevel cell-centered projection. Differs from CCProject in that a correction is not returned.

- ```
  void gradPhi(LevelData<FArrayBox>& a_gradPhi,
               const Vector<LevelData<FArrayBox>* >& a_vectPhi,
               int a_dir, int a_level) const
  ```

  Returns face-centered grad(phi) in direction dir.

  - `a_gradPhi` – dir-component of face-centered Grad(phi).
  - `a_vectPhi`

- `a_dir` – component of gradient to compute
- `a_level` – which AMR level on which to compute grad(phi)

- `void gradPhi(LevelData<FluxBox>& a_gradPhi, const Vector<LevelData<FArrayBox>* >& a_vectPhi, int a_level) const`

  Returns all components of grad(phi) (gradPhi should be correct size). This includes transverse components.

  - `a_gradPhi` – face-centered grad(phi), which should have `SpaceDim` components on each face.
  - `a_vectPhi`
  - `a_level` – which AMR level on which to compute grad(phi)

- `void gradPi(LevelData<FArrayBox>& a_gradPi, const Vector<LevelData<FArrayBox>* >& a_vectPi, int a_dir, int a_level) const`

  Computes grad(pi) in direction dir (similar to `gradPhi` function).

- `void gradPi(LevelData<FArrayBox>& a_gradPi, const Vector<LevelData<FArrayBox>* >& a_vectPi, int a_level) const`

  returns grad(pi) in all directions into SpaceDim-dimensioned gradPi.

References

[1] Dan Martin and Phil Colella. *Software Design for Particles in Incompressible Flow.* Applied Numerical Algorithms Group, Lawrence Berkeley Laboratory, 2003.