The Physics and Algorithms of Enzo

John Wise (Georgia Tech)
Most of the slides are originally written by Greg Bryan (Columbia)

Enzo Workshop – 21 November 2014

Introducing 円相Enzo

- Block-structured AMR + N-body
- Proper or comoving coordinates
- N-body: Adaptive particle-mesh solver
- (Magneto-)Hydrodynamics:
  - High-resolution shock capturing scheme
  - OR Finite differencing
- Chemical network solver (H, He, H_2, HD)
- Star & BH formation and feedback
- Radiative transfer
  - Adaptive (angular) ray tracing
  - OR Flux-limited diffusion

I. Hydrodynamics

Fluid Equations - `grid::SolveHydroEquations`

Mass conservation

$$\frac{\partial \rho}{\partial t} + \frac{1}{a}\mathbf{v}\cdot\nabla\rho = -\frac{1}{a}\rho\nabla\cdot\mathbf{v}$$

Momentum conservation

$$\frac{\partial \mathbf{v}}{\partial t} + \frac{1}{a}(\mathbf{v}\cdot\nabla)\mathbf{v} = -\frac{\dot{a}}{a}\mathbf{v} - \frac{1}{a\rho}\nabla p - \frac{1}{a}\nabla\phi,$$

Energy conservation

$$\frac{\partial E}{\partial t} + \frac{1}{a}\mathbf{v}\cdot\nabla E = -\frac{\dot{a}}{a}(3\frac{p}{\rho} + \mathbf{v}^2) - \frac{1}{a\rho}\nabla\cdot(p\mathbf{v}) - \frac{1}{a}\mathbf{v}\cdot\nabla\phi + \Gamma - \Lambda.$$

$$E = e + \frac{1}{2}\mathbf{v}^2,$$

Ideal Gas EOS

$$e = p/\left[(\gamma - 1)\,\rho\right],$$

Self-gravity

$$\nabla^2\phi = \frac{4\pi G}{a}(\rho_{total} - \rho_0).$$

Field names: `Density`, `Pressure`, `TotalEnergy`, `InternalEnergy`, `Velocity1`, `Velocity2`, `Velocity3`

grid class: accessing the fields – `grid.h`

- In grid class:
  - `BaryonFields[]` – array of pointers to each field
    - Fortran (row-major) ordering within each field
  - `GridRank` – dimensionality of problem
  - `GridDimensions[]` – dimensions of this grid
  - `GridStartIndex[]` – Index of first "active" cell (usually 3)
    - First (and last) three cells are ghost or boundary zones

```
int DensNum = FindField(Density, FieldType, NumberOfBaryonFields);
int Vel1Num = FindField(Velocity1, FieldType, NumberOfBaryonFields);

for (k = GridStartIndex[2]; k <= GridEndIndex[2]; k++) {
 for (j = GridStartIndex[1]; j <= GridEndIndex[1]; j++) {
  for (i = GridStartIndex[0]; i <= GridEndIndex[0]; i++) {
   BaryonField[Vel1Num][GINDEX(i,j,k)] *= BaryonField[DensNum][GINDEX(I,j,k)];
  }
 }
}
```

Enzo file name convention

- General C++ routines:
  - Routine name: EvolveLevel(…)
  - In file: EvolveLevel.C
  - One routine per file!
- grid methods:
  - Routine name: `grid::MyName(…)`
  - In file: `Grid_MyName.C`
- Fortran routines:
  - Routine name: `intvar(…)`
  - In file: `intvar.src`
    - .src is used because routine is fed first through C preprocessor

PPM Solver: `grid::SolvePPM_DE`

- `HydroMethod = 0`
- PPM: e.g. mass conservation equation
  - Flux conservative form:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho v}{\partial x} = 0$$

$$\rho_j^n = \rho(x_j, t^n)$$

Mass flux across j+1/2 boundary

  - In discrete form:

$$\rho_j^{n+1} = \rho_j^n + \Delta t \left(\frac{\overline{\rho}_{j+1/2}\overline{v}_{j+1/2} - \overline{\rho}_{j-1/2}\overline{v}_{j-1/2}}{\Delta x_j} \right)$$

  - How to compute mass flux?
- Note: multi-dimensions handled by operating splitting
  - `grid::xEulerSweep.C`, `grid::yEulerSweep.C`, `grid::zEulerSweep.C`

Grid::SolvePPM_DE

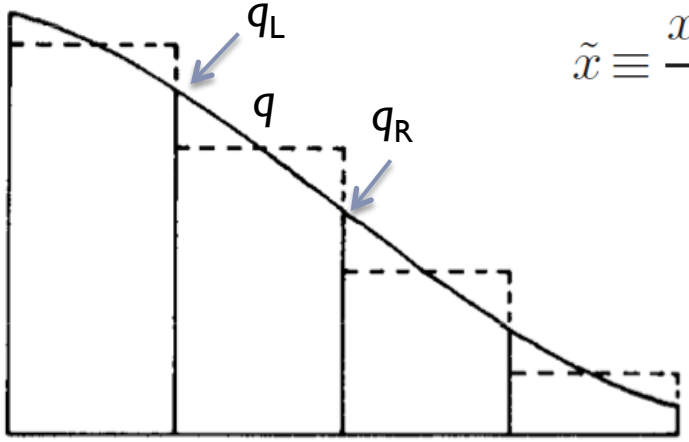
```
 // Update in x-direction
for (k = 0; k < GridDimension[2]; k++) {
    if (this->xEulerSweep(k, NumberOfSubgrids, SubgridFluxes,
                          GridGlobalStart, CellWidthTemp, GravityOn,
                          NumberOfColours, colnum) == FAIL) {
        fprintf(stderr, "Error in xEulerSweep.  k = %d\n", k);
        ENZO_FAIL("");
    }
 } // ENDFOR k

 // Update in y-direction
 for (i = 0; i < GridDimension[0]; i++) {
     if (this->yEulerSweep(i, NumberOfSubgrids, SubgridFluxes,
                          GridGlobalStart, CellWidthTemp, GravityOn,
                          NumberOfColours, colnum) == FAIL) {
      fprintf(stderr, "Error in yEulerSweep.  i = %d\n", i);
      ENZO_FAIL("");
    }
   } // ENDFOR i
```

PPM: 1D hydro update: `grid::xEulerSweep`

- Copy 2D slice out of cube
- Compute pressure on slice (pgas2d)
- Calculate diffusion/steepening coefficients (calcdiss)
- Compute Left and Right states on each cell edge (inteuler)
- Solve Reimann problem at each cell edge (twoshock)
- Compute fluxes of conserved quantities at each cell edge (euler)
- Save fluxes for future use
- Return slice to cube

PPM: reconstruction: `inteuler`

- Piecewise parabolic representation:



$$q_j(x) = q_{L,j} + \tilde{x}(\Delta q_j + q_{6,j}(1 - \tilde{x})),$$

$$\tilde{x} \equiv \frac{x - x_{j-1/2}}{\Delta x_j}, \qquad x_{j-1/2} \le x \le x_{j+1/2}.$$

- Coefficients (Δq and q_6) computed with mean q and q_L, q_R.
- For smooth flow (like shown above), this is fine, but can cause a problem for discontinuities (e.g. shocks)
- q_L, q_R are modified to ensure monotonicity (no *new* extrema)

PLM: reconstruction

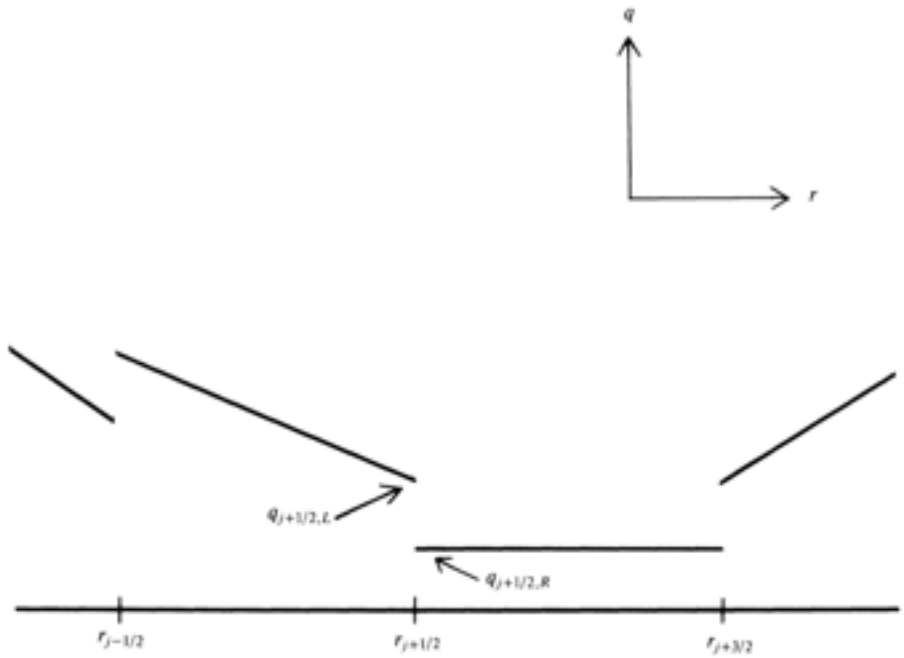
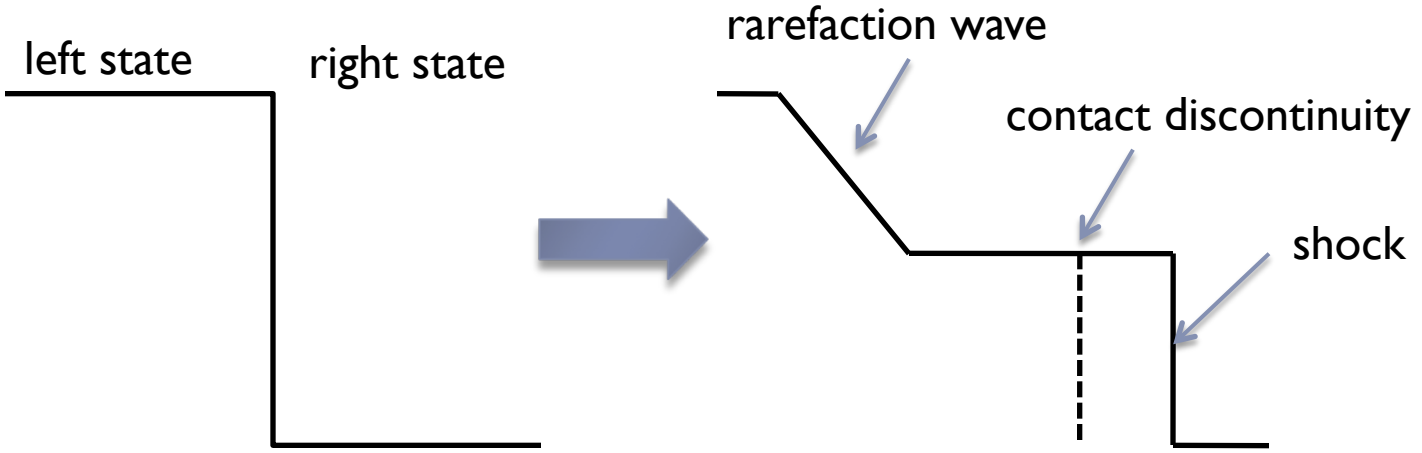


FIG. 1. *Spatial distribution of q at initial time t^n.*

- Piecewise linear method
- More diffusive reconstruction scheme, but more stable.

PPM: Godunov method: twoshock

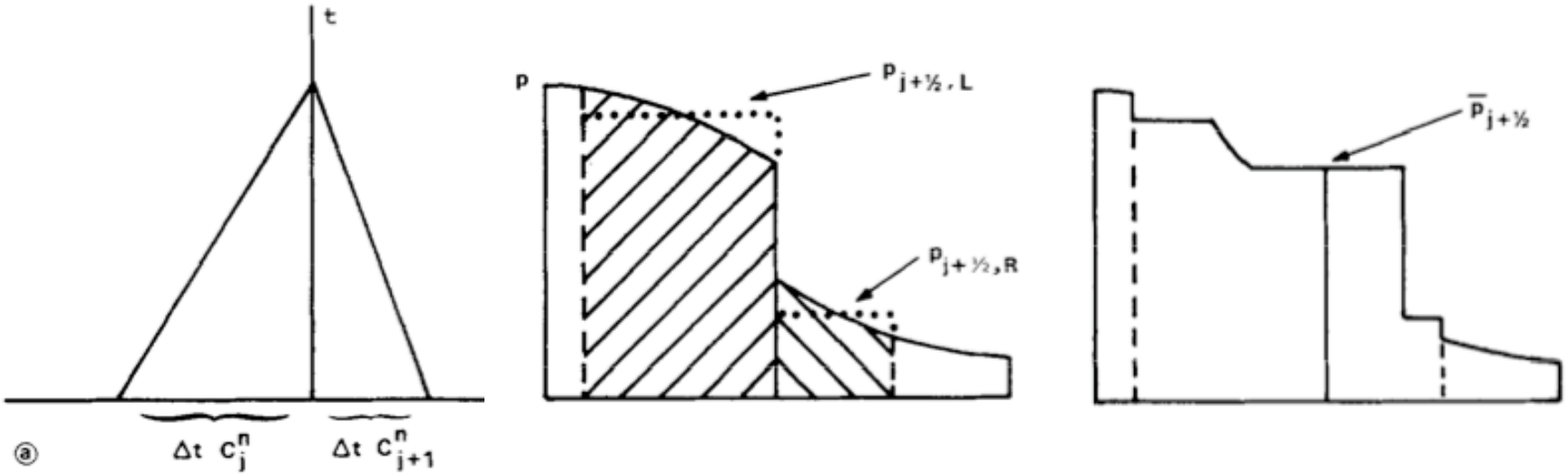
- To compute flux at cell boundary, take two initial constant states and then solve Riemann problem at interface



- Given solution, can compute flux across boundary
- Advantage: correctly satisfies jump conditions for shock

PPM: Godunov method: `inteuler`, `twoshock`

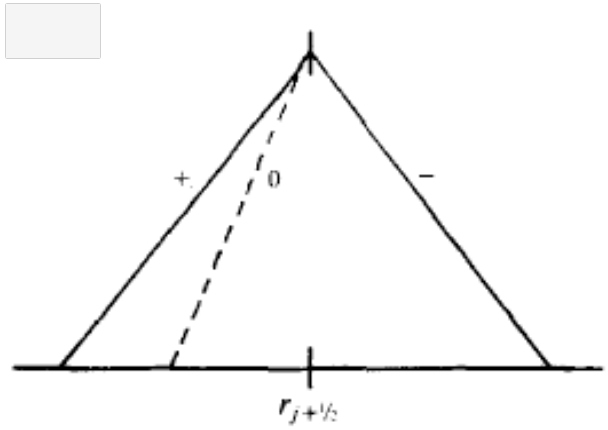
- For PPM, compute left and right states by averaging over characteristic region (causal region for time step Δt)



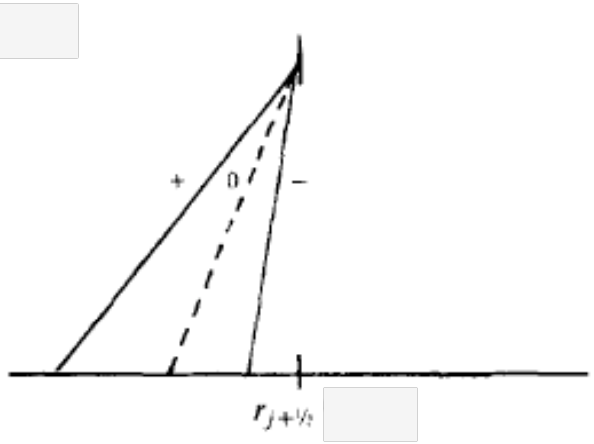
- Average left and right regions become constant regions to be feed into Riemann solver (`twoshock`).

PPM: Eulerian corrections: euler

- Eulerian case more complicated because cell edge is fixed.
  - Characteristic region for fixed cell more complicated:



SUBSONIC CASE



SUPERSONIC CASE

  - Note that solution is not known ahead of time so two-step procedure is used (see Collela & Woodward 1984 for details)

Other Riemann solvers

- HLL: (Harten-Lax-Leer)

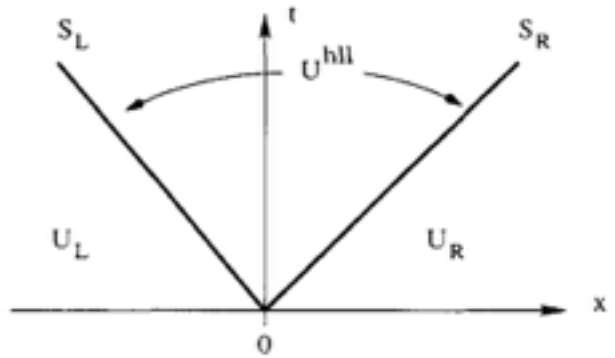


Fig. 10.3. Approximate HLL Riemann solver. Solution in the *Star Region* consists of a single state $\mathbf{U}^{hll}$ separated from data states by two waves of speeds S_L and S_R

- HLLC: HLL but considering the contact wave

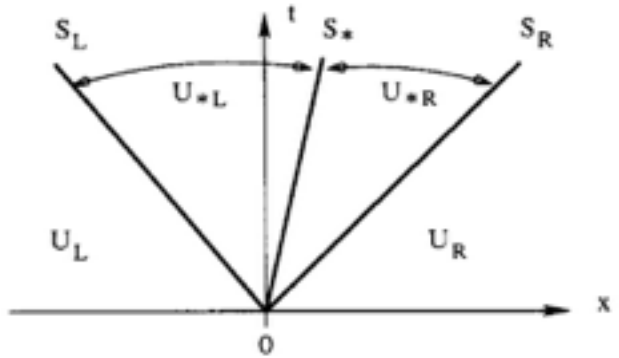


Fig. 10.4. HLLC approximate Riemann solver. Solution in the *Star Region* consists of two constant states separated from each other by a middle wave of speed S_*

Difficulty with very high Mach flows

- PPM is flux conservative so natural variables are mass, momentum, *total* energy
- Internal energy (e) computed from total energy (*E*):

$$e = E - \frac{1}{2}\mathbf{v}^2$$

- Problem can arise in *very* high Mach flows when *E* >> e
  - e is difference between two large numbers
- Not important for flow dynamics since *p* is negligible
  - But can cause problems if we want accurate temperatures since $T \alpha$ e

Dual Energy Formalism:
`grid::ComputePresureDualEnergyFormalism`

- Solution: Also evolve equation for internal energy:

$$\frac{\partial e}{\partial t} + \frac{1}{a}\mathbf{v} \cdot \nabla e = \frac{p}{a\rho}\nabla \cdot \mathbf{v}$$

- Select energy to use depending on ratio e/E:

$$p = \begin{cases} \rho(\gamma - 1)(E - \mathbf{v}^2/2), & (E - \mathbf{v}^2/2)/E > \eta_1; \\ \rho(\gamma - 1)e, & (E - \mathbf{v}^2/2)/E < \eta_1. \end{cases}$$

- Select with `DualEnergyFormalism = 1`
- Use when v/c_s > ~20
- Q: Why not just use e?
  - A: Equation for e is not in conservative form (source term).
  - Source term in internal energy equation causes diffusion

Zeus Solver: `grid::ZeusSolver`

- Traditional finite difference method
  - Artificial viscosity (see Stone & Norman 1992)
- `HydroMethod = 2`
- Source step: `ZeusSource`
  - Pressure (and gravity) update: $$v_j^{n+a} = v_j^n - \frac{\Delta t}{\Delta x_j} \frac{p_j^n - p_{j-1}^n}{(\rho_j^n + \rho_{j-1}^n)/2}$$
  - Artificial viscosity: $$v_j^{n+b} = v_j^{n+a} - \frac{\Delta t}{\Delta x_j} \frac{q_j^{n+a} - q_{j-1}^{n+a}}{(\rho_j^n + \rho_{j-1}^n)/2}$$
    $$q_j = \begin{cases} Q_{\mathrm{AV}} \rho_j (v_{j+1} - v_j)^2 & \mathrm{if} (\mathrm{v_{j+1}} - \mathrm{v_j}) < 0 \\ 0 & \mathrm{otherwise} \end{cases}$$
  - Compression heating: $$e_j^{n+c} = e_j^{n+b} \left(\frac{1 - (\Delta t/2)(\gamma - 1)(\nabla \cdot \mathbf{v})_j}{1 + (\Delta t/2)(\gamma - 1)(\nabla \cdot \mathbf{v})_j} \right)$$

Zeus Solver: `grid::ZeusSolver`

- Transport step: `Zeus_xTransport`

e.g. $$\rho_j^{n+d} = \rho_j^n - \frac{\Delta t}{\Delta x}(v_{j+1/2}^{n+c}\rho_{j+1/2}^* - v_{j-1/2}^{n+c}\rho_{j-1/2}^*)$$

  - Note conservative form (transport part preserves mass)
  - Note v_{j+1} is face-centered so is really at cell-edge, but density needs to be interpolated. Zeus uses an upwinded van Leer (linear) interpolation:

$$q_j(x) = q_{L,j} + \tilde{x}(\Delta q_j$$

  - Similarly for momentum and energy (and y and z)
    - Zeus_yTransport, Zeus_zTransport

Zeus Solver: `grid::ZeusSolver`

- PPM is more accurate, slower but Zeus is faster and more robust.
  - PPM often fails ("dnu < 0" error) when fast cooling generates large density gradients.
  - Try out new hydro solvers in Enzo 2.0!
- Implementation differences with PPM:
  - Internal energy equation only
    - In code, `TotalEnergy` field is really internal energy (ugh!)
  - Velocities are face-centered
    - `BaryonField[Vel1Num][GINDEX(i,j,k)]` really "lives" at `i-1/2`

II. Block Structured AMR

AMR: EvolveHierarchy

- Root grid NxNxN, so Δx = DomainWidth/N
- Level L defined so Δx = DomainWidth/($N2^L$)
- Starting with level 0, grid advanced by Δt
  - Main loop of EvolveHierarchy looks (roughly) like this:

```
InitializeHierarchy
While (Time < StopTime)
begin
  dt = ComputeTimeStep(0)
  EvolveLevel(0, dt)
  Time = Time + dt
  CheckForOutput(Time)
end
```

  - EvolveLevel does the heavy lifting

Time Step: `grid::ComputeTimeStep`

- Timestep on level L is minimum of constraints over all level L grids:

$$\Delta t_{hydro} = min\left(\kappa_{hydro}\frac{a\Delta x}{c_s + |v_x|}\right)_L \qquad \kappa_{hydro} \quad \texttt{CourantSafetyFactor}$$

$$\Delta t_{dm} = min\left(\kappa_{dm}\frac{a\Delta x}{v_{dm,x}}\right)_L, \qquad \kappa_{dm} \quad \texttt{ParticleCourantSafetyFactor}$$

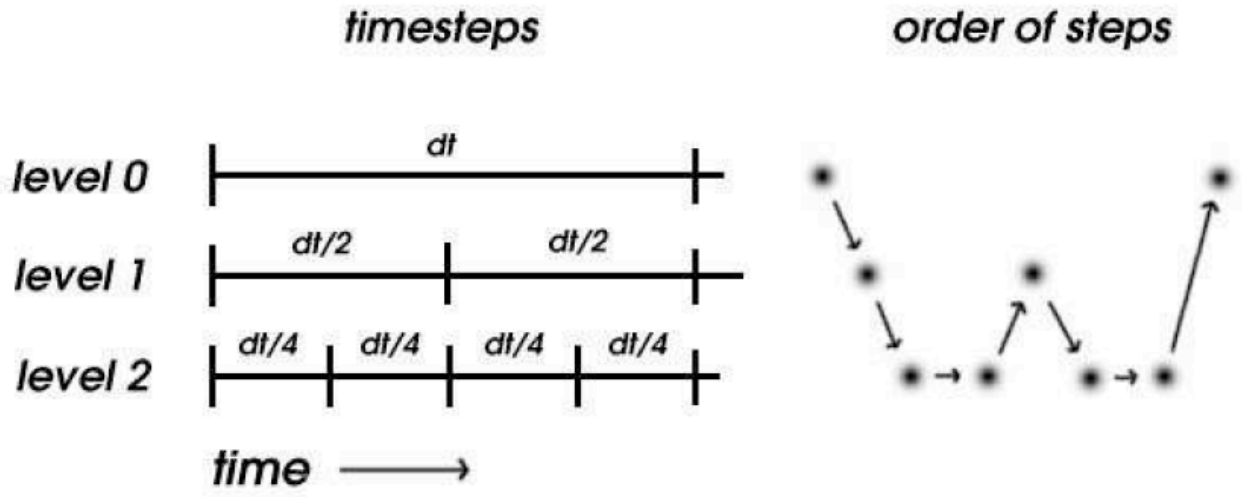
$$\Delta t_{exp} = f_{exp}\left(\frac{a}{\dot{a}}\right), \qquad f_{exp} \quad \texttt{MaximumExpansionFactor}$$

$$\Delta t_{accel} = min\left(\sqrt{\frac{\Delta x}{\vec{g}}}\right)_L$$

- + others (e.g. MHD, FLD, etc.)

AMR: `EvolveLevel`

- Levels advanced as follows:



  - Timesteps may not be integer ratios
    - (Diagram assumes Courant condition dominates and sound speed is constant so: dt α Δx)
  - This algorithm is defined in `EvolveLevel`

Advance grids on level: `EvolveLevel`

- The logic of `EvolveLevel` is given (roughly) as:

```
EvolveLevel(level)
begin
  SetBoundaryValues
  while (Time < ParentTime)
  begin
    dt = ComputeTimeStep(level)
    SolveHydroEquations(dt)
    Time = Time + dt
    SetBoundaryValues
    EvolveLevel(level+1, dt)
    FluxCorrection
    Projection
    RebuildHierarchy(level+1)
  end
end
```

Already talked about this.

recursive

Next, we'll talk about these

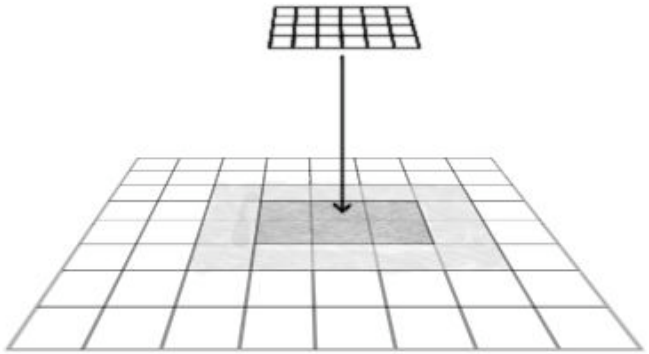
BC's: SetBoundaryConditions

- Setting "ghost" zones around outside of domain
  - `grid::SetExternalBoundaryValues`
  - Choices: `reflecting, outflow, inflow, periodic`
  - Only applied to level 0 grids (except periodic)
- Otherwise, two step procedure:
  - Interpolate ghost (boundary) zones from level L-1 grid
    - `grid::InterpolateBoundaryFromParent`
      - Linear interpolation in time (`OldBaryonFields`)
      - Spatial interpolation controlled by `InterpolationMethod`
        - `SecondOrderA` recommended, default (3D, linear in space, monotonic)
  - Copy ghost zones from sibling grids
    - `grid::CheckForOverlap` and `grid::CopyZonesFromGrid`

Projection: `grid::ProjectSolutionToParentGrid`

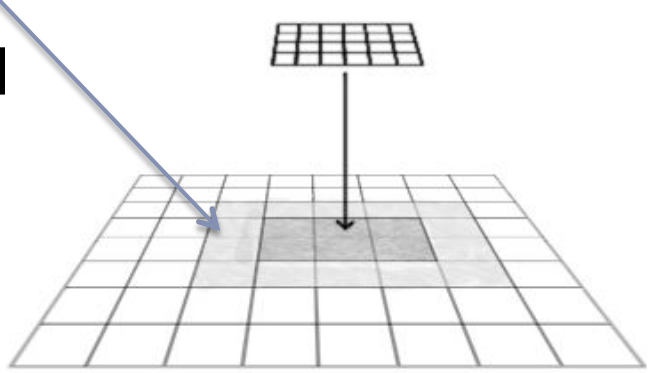
- Structured AMR produces redundancy:
  - coarse and fine grids cover same region
- Need to restore consistency
- Correct coarse cells once grids have all reach the same time:

$$q^{\text{coarse}} = r^{-d} \sum q^{\text{fine}}_{i,j,k}$$



Flux Correction:
`grid::CorrectForRefinedFluxes`

- Mismatch of fluxes occurs around boundary of fine grids
  - Coarse cell just outside boundary used coarse fluxes but coarse cell inside used fine fluxes
- Both fine and coarse fluxes saved from hydro solver



$$q^{\text{coarse}} = \tilde{q}^{\text{coarse}} - \Delta t \left(F^{\text{coarse}} - \sum_{j,k} F^{fine}_{j,k} \right)$$

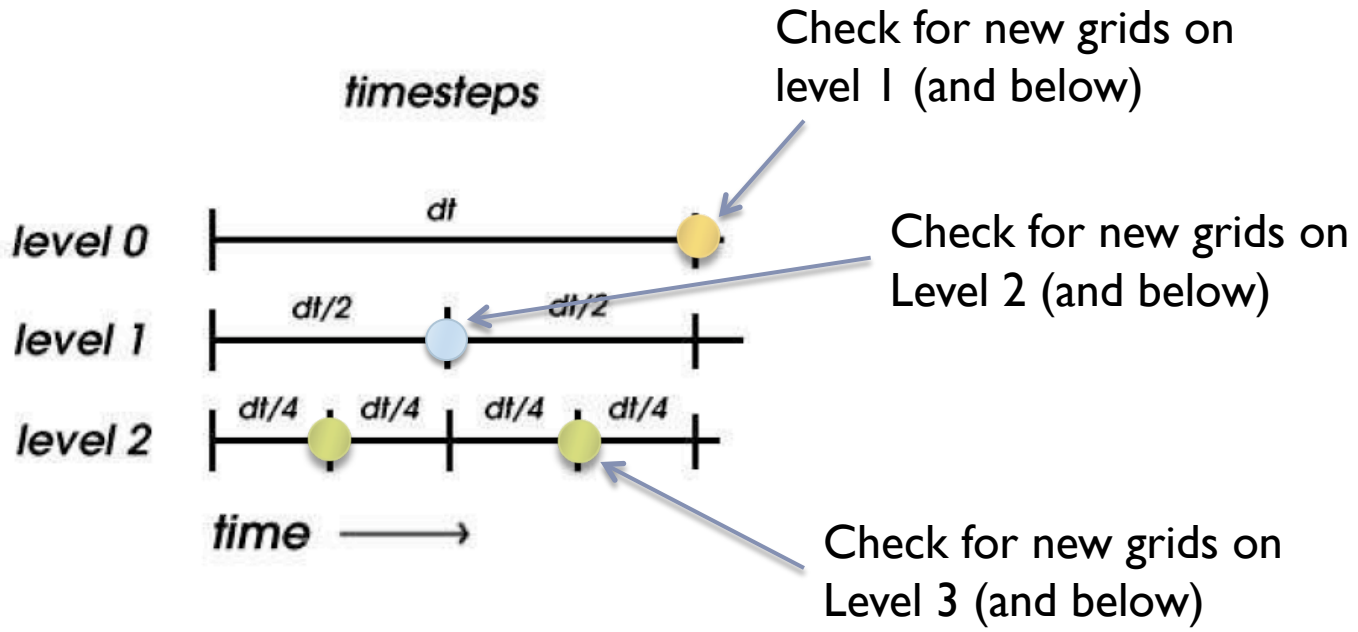
Uncorrected coarse value

Coarse flux across boundary

Sum of fine fluxes Over 4 (in 3D) abutting fine cells

Rebuilding the Hierarchy: `RebuildHierarchy`

- Need to check for cells needing more refinement



Refinement Criteria – `grid::SetFlaggingField`

- Many ways to flag cells for refinement
  - `CellFlaggingMethod =`

```
1 - refine by slope
2 - refine by baryon mass
3 - refine by shocks
4 - refine by particle mass
6 - refine by Jeans length
7 - refine if cooling time < cell width/sound speed
11 - refine by resistive length
12 - refine by defined region "MustRefineRegion"
13 - refine by metallicity
```

- Then rectangular grids must be chosen to cover all flagged cells with minimum "waste"
  - Done with machine vision technique
    - Looks for edges (inflection points in number of flagged cells)
  - `ProtoSubgrid` class

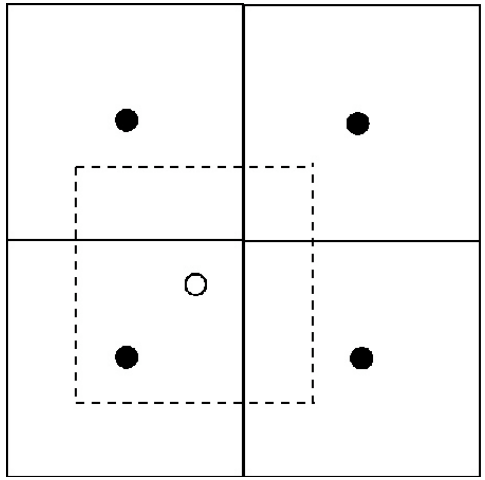
III. Gravity

Self-Gravity (`SelfGravity = 1`)

- Solve Poisson equation
- `PrepareDensityField`
  - `BaryonField[Density]` copied to `GravitatingMassField`
  - Particle mass is deposited in 8 nearest cells (CIC)
    - Particle position advanced by ½ step
    - `DepositParticleMassField`
- Root grid (level 0):
  - Potential solved with FFT

$$\tilde{\phi}(k) = G(k)\tilde{\rho}(k).$$

    - `ComputePotentialFieldLevelZero`
  - Potential differenced to get acceleration
    - `grid::ComputeAccelerationField`



Self-Gravity

- Subgrids:
  - Potential interpolated to boundary from parent
    - `Grid::PreparePotentialField`
  - Each subgrid then solves Poisson equation using multigrid
    - `Grid::SolveForPotential`
  - Note: this has two issues:
    - Interpolation errors on boundary can propagate to fine levels
      - Generally only an issue for steep potentials (point mass)
      - Ameliorated by having 6 ghost zones for gravity grid
    - Subgrids can have inconsistent potential gradients across boundary
      - Improved by copying new boundary conditions from sibilings and resolving the Poisson equation (`PotentialIterations = 4` by default)
    - More accurate methods in development

Other Gravitational sources – grid::ComputeAccelerationFieldExternal

- Can also add fixed potential:
  - `UniformGravity` – constant field
  - `PointSourceGravity` – single point source
  - `ExternalGravity` – NFW profile

IV. Particles

N-body dynamics

- Particles contribute mass to `GravitatingMassField`
- Particles accelerated by `AccelerationField`
  - Interpolated from grid (from 8 nearest cells)
- Particles advanced using leapfrog

$$x^{n+1/2} = x^n + (\Delta t/2)v^n$$
$$v^{n+1} = v^n + \Delta t a^{n+1/2}$$
$$x^{n+1} = x^{n+1/2} + (\Delta t/2)v^{n+1}$$

  - `grid::ComputeAccelerations`
- Particles stored in the locally most-refined grid
  - `ParticlePosition, ParticleVelocity, ParticleMass`
- Tracer particles (massless) also available