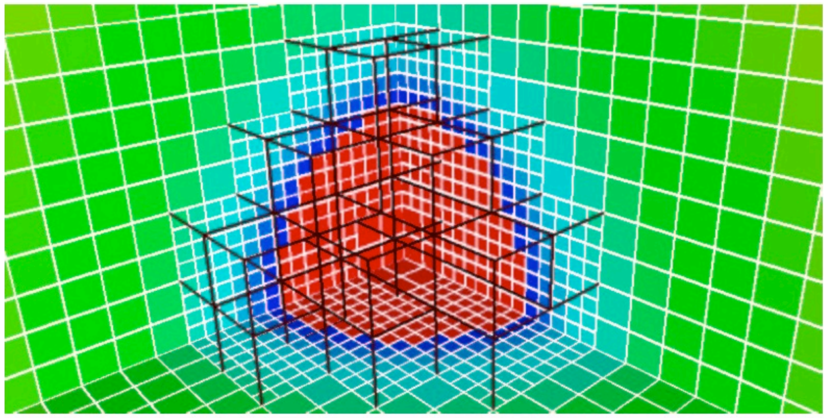
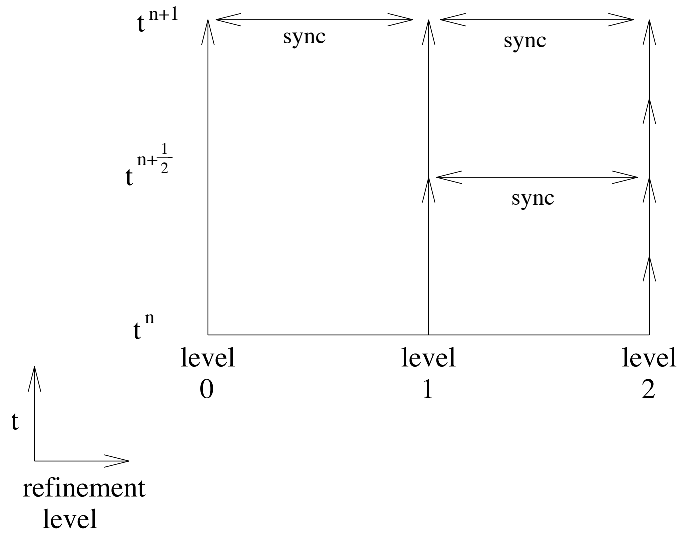
Block Structured AMR Libraries
(plus a note on PETSc Interface)

Brian Van Straalen

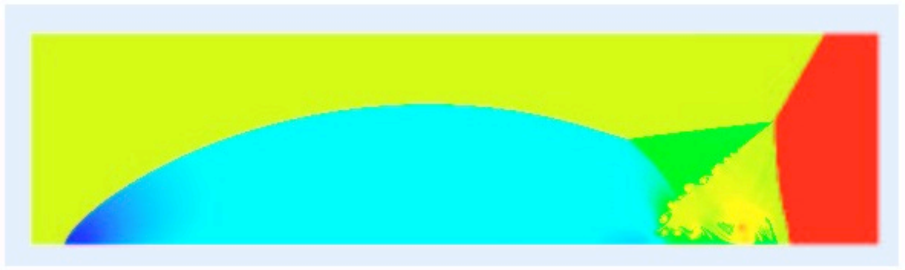
FASTMath SciDAC Institute

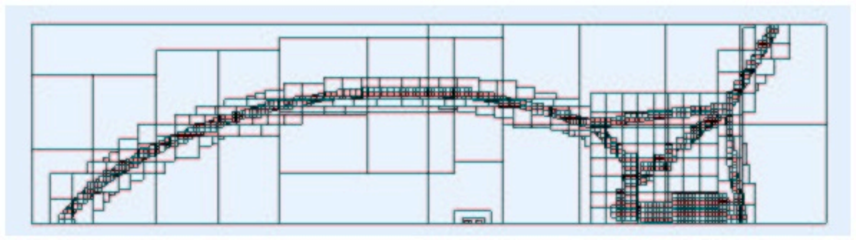
Block-Structured Local Refinement (Berger and Oliger, 1984)





Refined regions are organized into logically-rectangular patches.
Refinement is performed in time as well as in space.





Why use AMR and When ?

- Think of AMR as a compression technique for the discretized mesh

- Apply higher resolution in the domain only where it is needed

- When should you use AMR:
  - When you have a multi-scale problem
  - When a uniformly spaced grid is going to use more memory than you have available to achieve the resolution you need

- You cannot always use AMR even when the above conditions are met
- When should you not use AMR:
  - When the overhead costs start to exceed gains from compression
  - When fine-coarse boundaries compromise the solution accuracy beyond acceptability

Much as using any tool in scientific computing, you should know what are the benefits and limits of the technologies you are planning to use

The Flip Side - Complexity

- Machinery needed for computations :
  - Interpolation, coarsening, flux corrections and other needed resolutions at fine-coarse boundaries

- Machinery needed for house keeping :
  - The relationships between entities at the same resolution levels
  - The relationships between entities at different resolution levels

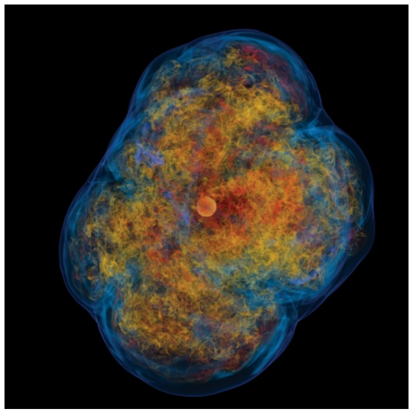
- Machinery needed for parallelization :
  - Domain decomposition and distribution among processors
    - Sometimes conflicting goals of maintaining proximity and load balance
  - Redistribution of computational entities when the grid changes due to refinement
  - Gets more complicated when the solution method moves away from explicit solves

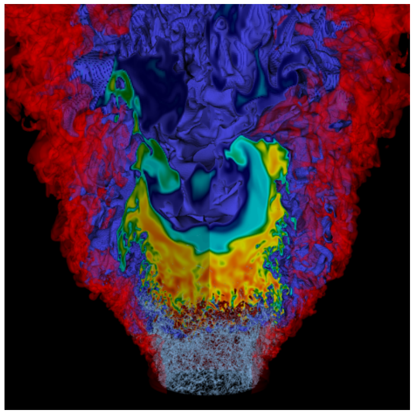
Chombo: Application Codes

- Large Eddy simulations of wind turbines (PI, R. Samtaney, KAUST, S. Guzik CSU).
- **CHARM** - Compressible CFD + collisionless particle cosmology simulations (PI F. Miniati, ETH Zurich)
- **MS_FLUKSS** - Physics of the Solar Wind and Its Interaction with the Interstellar medium using compressible hyperbolic CFD + electromagnetic and kinetic effects (G. Zank. and N. Pogorelov UA-Huntsville)
- **PLUTO** - General astrophysics modeling (PI A. Mignone)
- **ORION** - Astrophysical MHD turbulence PI C. McKee / R. Klein, UCB
- **REALM** - SF Bay and Delta Hydrology modeling shallow water (PI P. Schwartz, LBNL and CA Department of Water Resources)
- Plasma-wakefield accelerators -- compressible viscous flow (PI D. Graves, LBNL).
- **ChomboCrunc**h - Pore-scale subsurface reacting flow code (PI D. Trebotich, LBNL)
- Conjugate heat transfer in nuclear reactors (PI R. Crockett, LBNL, TechX)
- **COGENT** - 4D gyrokinetic models of tokamak edge (PI J. Hittinger, LLNL)
- **BISICLES** - Land ice model for climate simulation (PI D. Martin, LBNL)

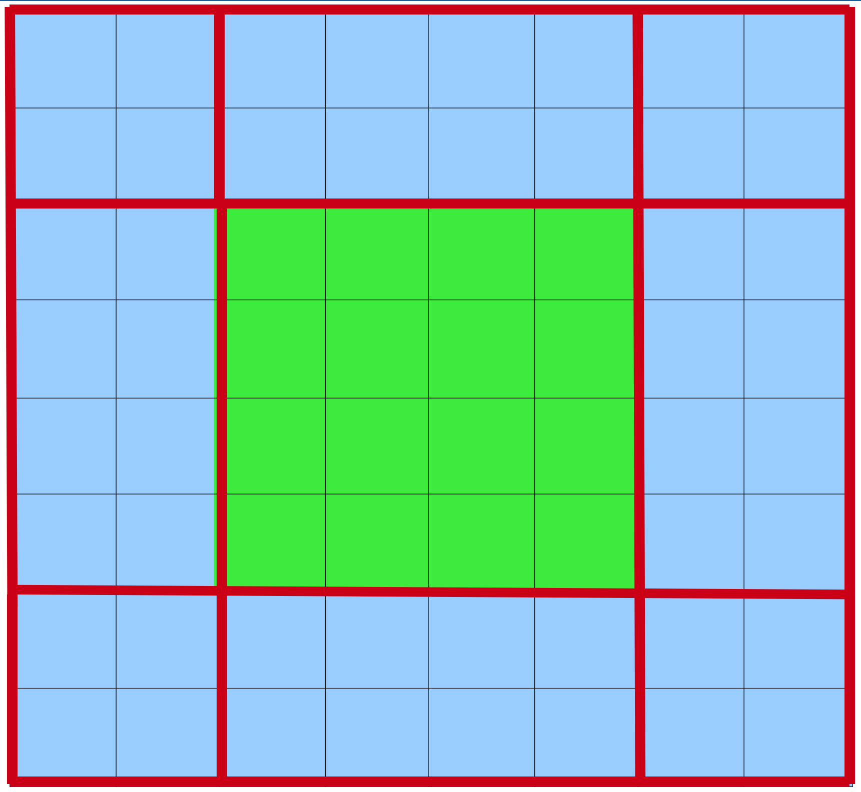
BoxLib-based Application Codes

- **BoxLib-based codes are being used in active research, including:**
  - MAESTRO – low Mach number astrophysics
  - CASTRO – compressible radiation/hydrodynamics
  - Nyx – cosmology (baryon plus dark matter evolution)
  - LMC – low Mach number combustion
  - RNS – higher-order method for compressible reacting flow
- **Shared features between many of these codes include:**
  - Explicit advection solvers
  - Semi-implicit diffusion solvers (using multigrid)
  - Projection operations (using multigrid)
  - Pointwise solution of ODE's (e.g. for chemical reactions)
  - Multilevel synchronization operations
  - Active and passive particles
- **Hybrid parallelism: MPI + OpenMP with logical tiling**
- **Demonstrated scaling to O(10K – 100K) cores**


Simulation of core-collapse supernova using CASTRO.


Simulation of low-swirl burner using LMC.

Abstraction for Data Localization



- Data is distributed
  - across the MPP machine
  - throughout the data hierarchy
- Data is redundantly localized for computation: *halo/ghost cells*
- The halo cells may come from same level exchanges or from a coarser level through interpolation
- If there is no sub-cycling, the interface is simple, all patches can get their halos filled simultaneously
- With sub-cycling either the application or the infrastructure can

Most structured AMR methods use the same abstraction for semi-implicit solvers such as multigrid, in the sense they operate on a block/box at a time, the operations in between and the orchestration gets more complicated

The Two Packages: BoxLib and Chombo

- Mixed-language model: C++ for higher-level data structures, Fortran for regular single-grid calculations.
- Reuseable components. Component design based on mapping of mathematical abstractions to classes.
- Build on public-domain standards: MPI and OpenMP
    - Chombo also uses HDF5
- Interoperability with other tools: VisIt, PETSc, hypre, SUNDIALS,LAPACK.
- The lowest levels are very similar – they had the same origin
- Downloads
    - BoxLib
        - https://ccse.lbl.gov/Downloads/downloadBoxLib.html
    - Chombo:
        - https://anag-repo.lbl.gov/
- Examples from Chombo

Software Reuse by Templating Dataholders

Classes can be parameterized by types, using the class template language feature in C++.

- `BaseFAB<T>` is a multidimensional array for any type `T`.
- `FArrayBox: public BaseFAB<Real>`
- `LevelData<T>, T` can be any type that “looks like” a multidimensional array.
    - Ordinary multidimensional arrays: `LevelData<FArrayBox>`
    - Binsorted lists of particles: `LevelData<BinFAB<ParticleType>>`
    - Data structures for embedded boundary methods.

FArrayBox Class

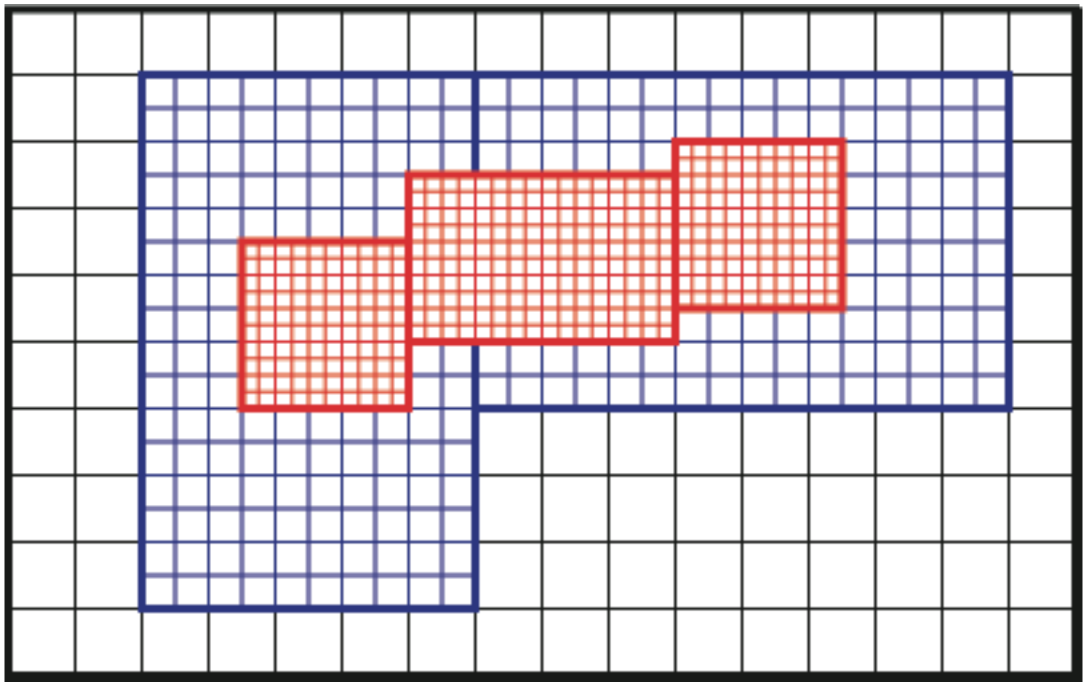
Specialized `BaseFab<T>` with additional floating-point operations – can add, multiply, divide, compute norms.

Example:

```
Box domain;
FArrayBox afab(domain, 1);   // define single-component FAB's
FArrayBox bfab(domain, 1);

afab.setVal(1.0);           // afab = 1 everywhere
bfab.setVal(2.0);           // set bfab to be 2

afab.mult(2.0);             // multiply all values in afab by 2
afab.plus(bfab);            // add bfab to afab (modifying afab)
```

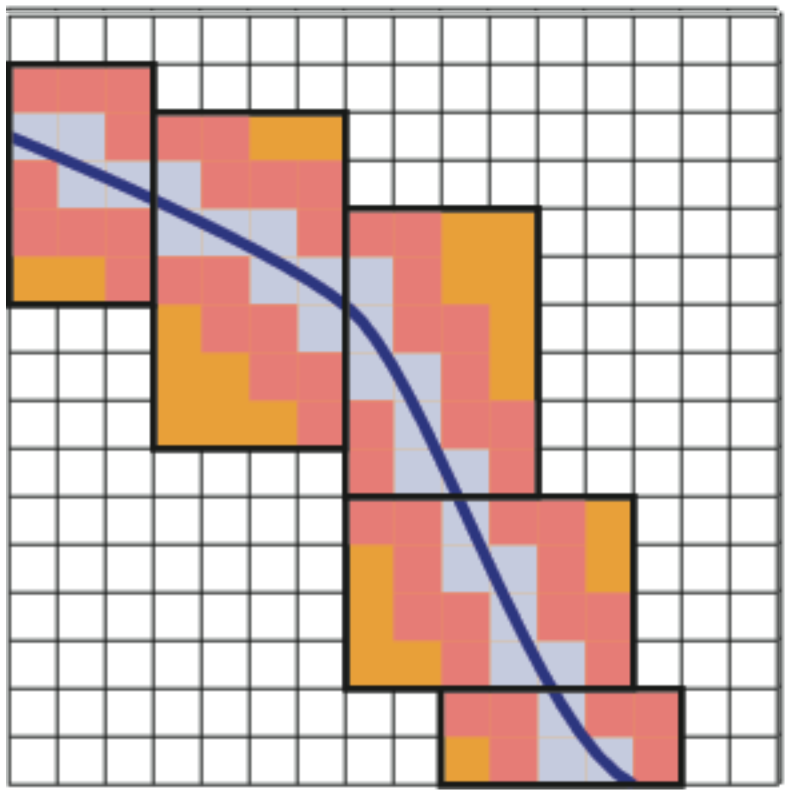
Approach

- Locally refine patches where needed to improve the solution.
- Each patch is a logically rectangular structured grid.
  - Better efficiency of data access.
  - Can amortize overhead of irregular operations over large number of regular operations.
- Refined grids are dynamically created and destroyed.



Building the Initial Hierarchy

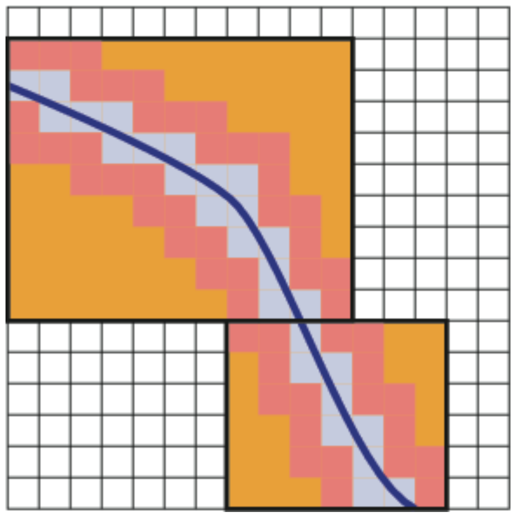
- *Boostrapping*
- Fill data at level 0
- Estimate where refinement is needed
- Group cells into patches according to constraints (refinement levels, grid efficiency etc)
- Repeat for the next level
- Maintains proper nesting

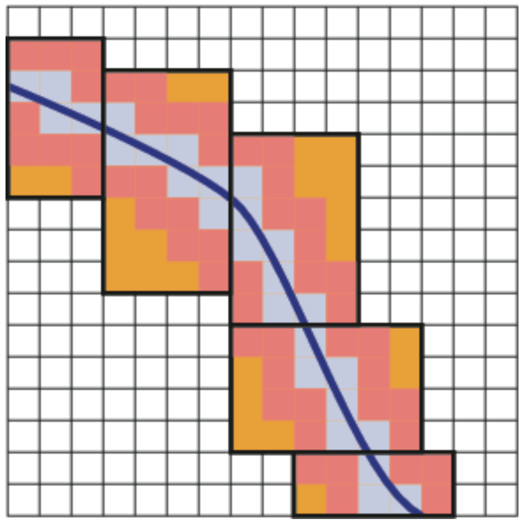


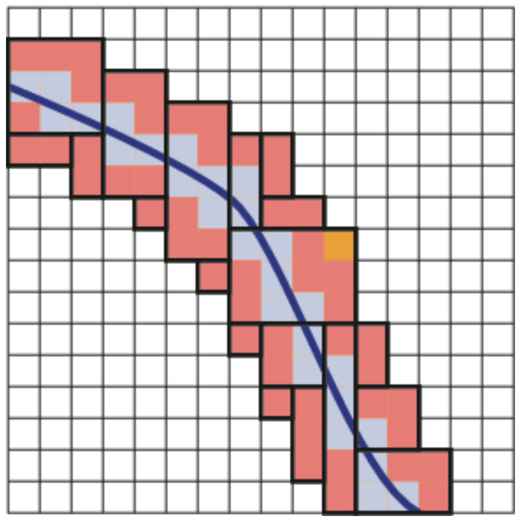
```
class AMR {
   Vector<AMRLevel*> m_levels;
};
class AMRLevel {
  virtual    void tagCells(IntVectSet& a_tags) = 0;
}
```

FASTMATH

How Efficiency Affects the Grid


Efficiency=0.5


Efficiency=0.7


Efficiency=0.9

each box dimension satisfies

```
    n%blocking_factor == 0
    n<=MaxBoxSize
```

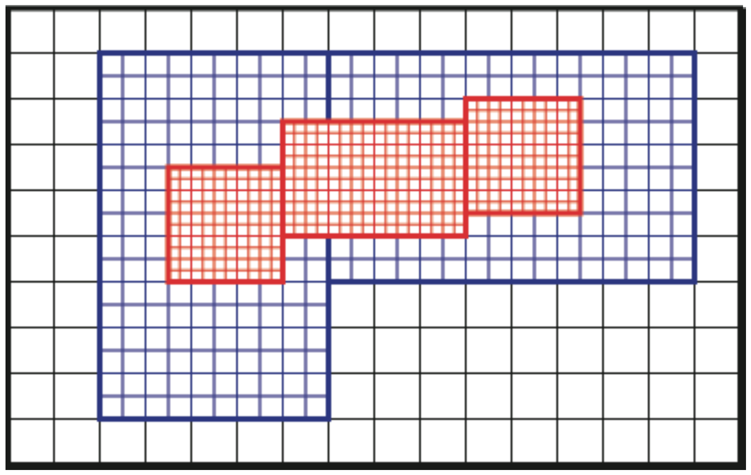
Argonne NATIONAL LABORATORY · BERKELEY LAB · Sandia National Laboratories · Berkeley University of California · Columbia University · Rensselaer · SMU · UBC THE UNIVERSITY OF BRITISH COLUMBIA

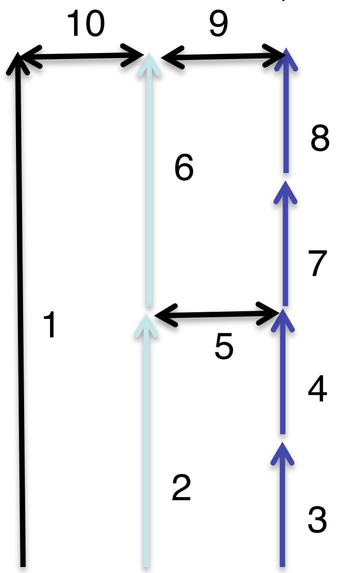
Adaptive in Time

- Consider two levels, coarse and fine with refinement ratio r

$$\Delta x_f = \Delta x_c / r \quad , \quad \Delta t_f = \Delta t_c / r,$$

- Berger-Oliger timestepping
  - Advance current level
  - Advance finer grids r times
  - Synchronize fine and coarse data
  - Apply recursively to all refinement levels
- Variations provided in Chombo Backward-Euler, Crank-Nicolson,TGA, RK4, ARK4. Can use SUNDIALS





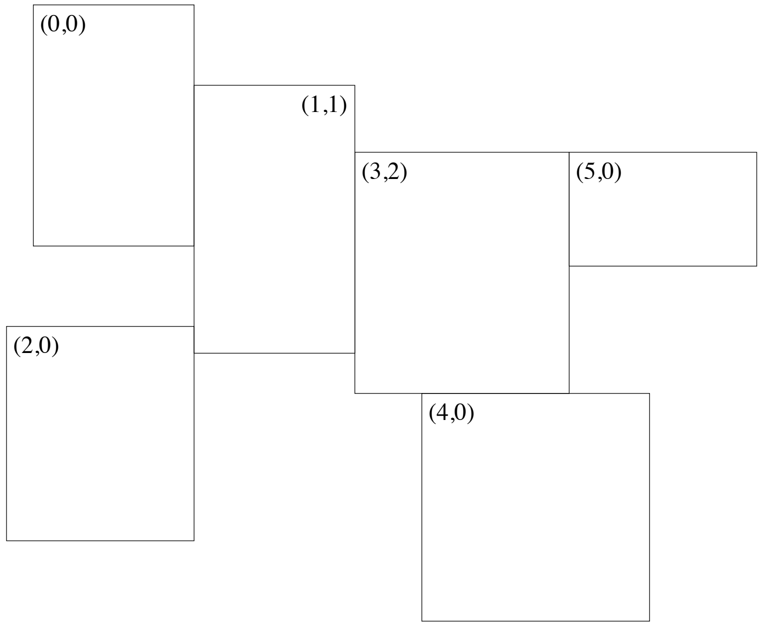
`BoxLayout` Operations

Set of `Boxes` which comprise the valid regions on a single level, including processor assignments for each rectangular region.

Two ways to iterate through a `BoxLayout`

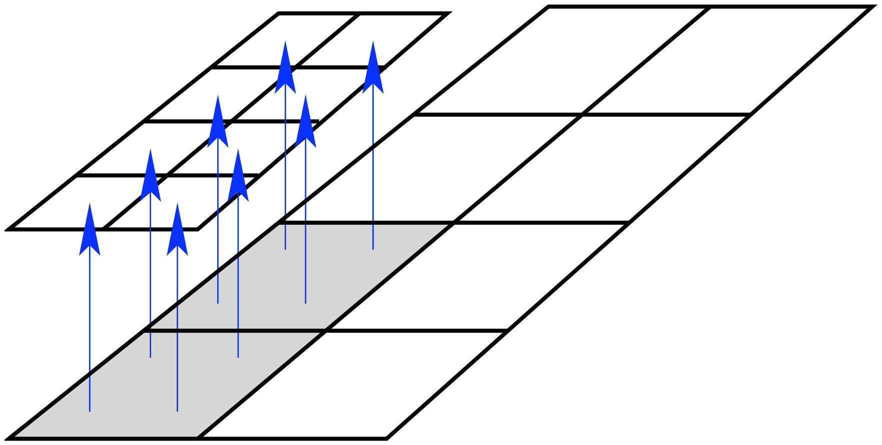
• `LayoutIterator` – iterates through **all** boxes in the `BoxLayout`, regardless of which processor they are assigned to.

• `DataIterator` – iterates only through the boxes on **this** processor.



Interpolation from coarse to fine

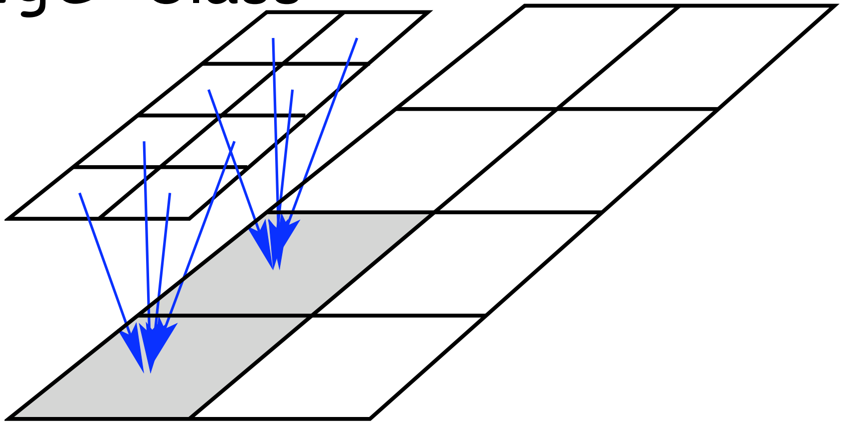
- Linearly interpolates data from coarse cells to the overlaying fine cells.
- Useful when initializing newly-refined regions after regridding.



Example:

```
ProblemDomain fineDomain;
DisjointBoxLayout coarseGrids, fineGrids;
int refinementRatio, nComp;
LevelData<FArrayBox> coarseData(coarseGrids, nComp);
LevelData<FArrayBox> fineData(fineGrids, nComp);

FineInterp interpolator(fineGrids, nComp, refinementRatio,
                        fineDomain);

// fineData is filled with linearly interpolated coarseData
interpolator.interpToFine(fineData, coarseData);
```

CoarseAverage Class



- Averages data from finer levels to covered regions in the next coarser level.
- Used for bringing coarse levels into sync with refined grids covering them.

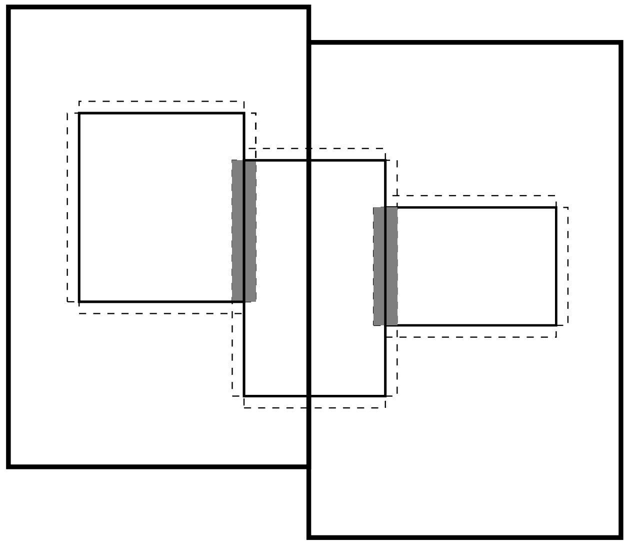
Example:

```
DisjointBoxLayout fineGrids;
DisjointBoxLayout crseGrids;
int nComp, refRatio;

LevelData<FArrayBox> fineData(fineGrids, nComp);
LevelData<FArrayBox> crseData(crseGrids, nComp);

CoarseAverage averager(fineGrids, crseGrids, nComp,
refRatio);

averager.averageToCoarse(crseData, fineData);
```

Coarse-Fine Interactions (`AMRTools`)

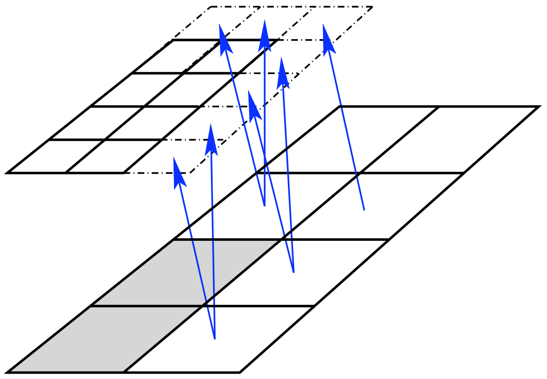
The operations that couple different levels of refinement are among the most difficult to implement, as they typically involve a combination of interprocessor communication and irregular computation.

- Interpolation between levels (`FineInterp`).
- Averaging down to coarser grids (`CoarseAverage`).
- Interpolation of boundary conditions (`PiecewiseLinearFillpatch`, `QuadCFInterp`, higher-order extensions).
- Managing conservation at refinement boundaries (`LevelFluxRegister`).



`PiecewiseLinearFillPatch` Class

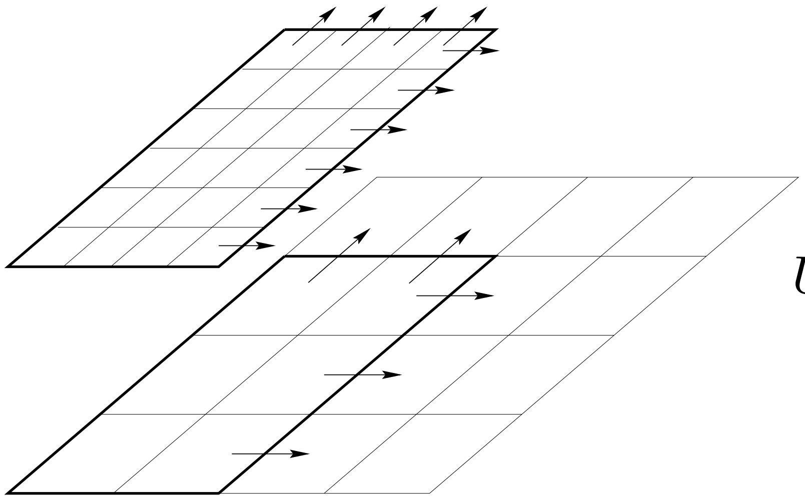
Linear interpolation of coarse-level data (in time and space) into fine-level ghost cells.



Example:

```
ProblemDomain crseDomain;
DisjointBoxLayout crseGrids, fineGrids;
int nComp, refRatio, nGhost;
Real oldCrseTime, newCrseTime, fineTime;

LevelData<FArrayBox> fineData(fineGrids, nComp,
nGhost*IntVect::Unit);
LevelData<FArrayBox> oldCrseData(crseGrids, nComp);
LevelData<FArrayBox> newCrseData(crseGrids, nComp);

PiecewiseLinearFillPatch filler(fineGrids, coarseGrids, nComp,
                                crseDomain, refRatio, nGhost);
Real alpha = (fineTime-oldCrseTime)/(newCrseTime-oldCrseTime);
filler.fillInterp(fineData, oldCrseData, newCrseData, alpha,
                  0, 0, nComp);
```

`LevelFluxRegister` Class



$$U^c := U^c + \Delta t^c (F^{c,s}_{\boldsymbol{i}^c - \frac{1}{2}\boldsymbol{e}} - \frac{1}{Z} \sum_{\boldsymbol{i}^f} F^{f,s}_{\boldsymbol{i}^f - \frac{1}{2}\boldsymbol{e}})$$

The coarse and fine fluxes are computed at different points in the program, and on different processors. We rewrite the process in the following steps.

$$\delta F = 0$$

$$\delta F := \delta F - F^c$$

$$\delta F := \delta F + < F^f >$$

$$U^c := U^c + D_R(\delta F)$$

Dimension-independence with ChomboFortran

Advantages to `ChomboFortran`:

- Enables fast (2D) prototyping, and nearly immediate extension to 3D.
- Simplifies code maintenance and duplication by reducing the replication of dimension-specific code.
- Automatic C++/Fortran interface generation and type-safety.

Replace

```
do j = nlreg(2), nhreg(2)
   do i = nlreg(1), nhreg(1)
      phi(i,j) = phi(i,j) + nu*dt*lphi(i,j)
   enddo
enddo
```

with

```
CHF_MULTIDO[dombox; i;j;k]
       phi(CHF_IX[i;j;k]) = phi(CHF_IX[i;j;k])
    &                     + nu*dt*lphi(CHF_IX[i;j;k])
     CHF_ENDDO
```

Prior to compilation, ChomboFortran replaces the indexing and looping macros with code appropriate to the dimensionality of the problem.

Example

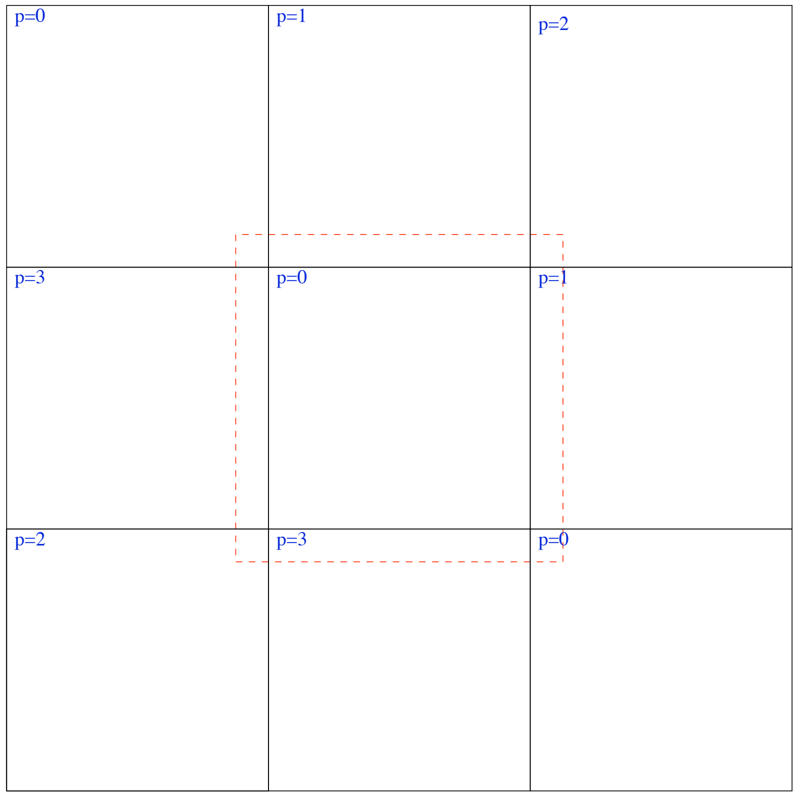
```
Vector<Box> boxes;         // boxes and processor assignments
Vector<int> procAssign;
ProblemDomain domain;
DisjointBoxLayout dbl(boxes, procAssign, domain);  // define dbl
// access _all_ boxes
LayoutIterator lit = dbl.layoutIterator();
for (lit.begin(); lit.ok(); ++lit)
{
  const Box thisBox = dbl[lit];
}
// access only local boxes
DataIterator dit =   dbl.dataIterator();
for (dit.begin(); dit.ok(); ++dit)

{

  const Box thisLocalBox = dbl[dit];

}
```

Example

```
DisjointBoxLayout grids;
int nComp = 2;                                              // two
data components
IntVect ghostVect(IntVect::Unit)     // one layer of ghost cells

LevelData<FArrayBox> ldf(grids, nComp, ghostVect);

              // Real distributed data.
LevelData<FArrayBox> ldf2(grids, nComp, ghostVect);

DataIterator dit = grids.dataIterator(); // iterate local data
for (dit.begin(); dit.ok(); ++dit)
{
  FArrayBox& thisFAB = ldf[dit];
  thisFAB.setVal(procID());
}

// fill ghost cells with "valid" data from neighboring grids
ldf.exchange();

ldf.copyTo(ldf2); \\ copy from ldf->ldf2
```

Example Explicit Heat Equation Solver, Parallel Case

Want to apply the same algorithm as before, except that the data for the domain is decomposed into pieces and distributed to processors.



Example Explicit Heat Equation Solver, Parallel Case

```
// C++ code:
  Box domain(IntVect:Zero,(nx-1)*IntVect:Unit);
  DisjointBoxLayout dbl;
// Break domain into blocks, and construct the DisjointBoxLayout.
  makeGrids(domain,dbl,nx);

  LevelData<FArrayBox> phi(dbl, 1, IntVect::TheUnitVector());

  for (int nstep = 0;nstep < 100;nstep++)
  {
…
// Apply one time step of explicit heat solver: fill ghost cell values,
// and apply the operator to data on each of the Boxes owned by this
// processor.

   phi.exchange();
```

Example Explicit Heat Equation Solver, Parallel Case

```
DataIterator dit = dbl.dataIterator();
   for (dit.reset();dit.ok();++dit)
   {
     FArrayBox& soln = phi[dit()];
     Box& region = dbl[dit()];
     FORT_HEATSUB(CHF_FRA(soln),
                  CHF_BOX(region),
                  CHF_BOX(domain),
                  CHF_REAL(dt), CHF_REAL(dx), CHF_REAL(nu));
  }
 }
```

`LevelFluxRegister` Class

A `LevelFluxRegister` object encapsulates these operations.

- `LevelFluxRegister::setToZero()`
- `LevelFluxRegister::incrementCoarse`: given a flux in a direction for one of the patches at the coarse level, increment the flux register for that direction.
- `LevelFluxRegister::incrementFine`: given a flux in a direction for one of the patches at the fine level, increment the flux register with the average of that flux onto the coarser level for that direction.
- `LevelFluxRegister::reflux`: given the data for the entire coarse level, increment the solution with the flux register data for all of the coordinate directions.

Layer 3 Classes: Reusing control structures via inheritance (`AMRTimeDependent`, `AMRElliptic`)

AMR has multilevel control structures that are largely independent of the operations and data.

• Berger-Oliger timestepping (refinement in time).

• Various linear solver operations, such as multigrid on a single level, multigrid on an AMR hierarchy.

To separate the control structure from the the details of the operation that are being controlled, we use C++ inheritance in the form of *interface classes*.

Matrix-Free Geometric Multigrid in Chombo

- `Chombo/src/AMRElliptic/`
  - `AMRLevelOp`
    - `AMRPoissonOp, VCAMRPoissonOp, ViscousTensorOp`
  - `AMRMultigrid`
- `Chombo/releasedExamples/AMRPoisson`
  - `execNode` `execVariableCoefficient`
  - `execCell` `execPETSc` `execViscousTensor`

- What happens in that odd one, execPETSc?

Matrix representation of operators

- We have seen how construct AMR operator in Chombo as series of sub-operations
  - Coarse interpolation, fine interpolation, boundary conditions, etc.

- Matrix-free operators
  - Low memory: good for performance and memory complexity
  - Can use same technology to construct matrix-free equation solvers
    - Operator inverse
    - Use geometric multigrid (GMG)
    - Inherently somewhat isotropic

- Some applications have complex geometry and/or anisotropy
  - GMG looses efficacy
  - Solution: algebraic multigrid (AMG)

- Need explicit matrix representation of operator
  - Somewhat complex bookkeeping task but pretty mechanical
  - Recently developed infrastructure in Chombo support matrix construction
  - Apply series of transformations to matrix or stencil
    - Similar to operator but operating matrix/stencil instead of field data
  - Stencil: list of <Real weight, <cell, level>>
    - Stencil + map <cell, level> to global equation number: row of matrix
  - Start with A^0 : initial operator matrix
    - Eg, 1D 3-point stencil: {<-1.0, <i-1,lev>, <2.0, <i,lev>, <-1.0, <i+1,lev>}

PETSc Composite Grid Solvers: `PetscCompGrid`

```
class PetscCompGrid
{
 virtual void define(const ProblemDomain &a_cdomain,
                     Vector<DisjointBoxLayout> &a_grids,
                     Vector<int> &a_refratios, BCHolder a_bc …)
 PetscErrorCode putChomboInPetsc(Vector<LevelData<FArrayBox>*> &rhs, Vec b )const;
 PetscErrorCode putPetscInChombo(Vec b, Vector<LevelData<FArrayBox>*> &rhs )const;
 Vector<RefCountedPtr<LevelData<BaseFab<PetscInt> > > > m_GIDs;
};
class PetscCompGridPois : public PetscCompGrid
{
  void createOpStencil(IntVect,int,const DataIndex&, StencilTensor&);
  PetscErrorCode createMatrix(int a_makePmat=0);
  Mat getMatrix() const { return m_mat; }
};
```

PETSc Composite Grid Example: `PetscCompGridPois`

```
void
PetscCompGridPois::createOpStencil( IntVect a_iv, int a_ilev,const
DataIndex &a_di_dummy, StencilTensor &a_sten)
{
  Real dx=m_dxs[a_ilev][0],idx2=1./(dx*dx);
  StencilTensorValue &v0 = a_sten[IndexML(a_iv,a_ilev)];
  v0.define(1);
  v0.setValue(0,0,m_alpha - m_beta*2.*SpaceDim*idx2);
  for (int dir=0; dir<CH_SPACEDIM; ++dir) {
   for (SideIterator sit; sit.ok(); ++sit) {
     int isign = sign(sit());
     IntVect jiv(a_iv); jiv.shift(dir,isign);
     StencilTensorValue &v1 = a_sten[IndexML(jiv,a_ilev)];
     v1.define(1);
     v1.setValue(0,0,m_beta*idx2);
}}}
```

PETSc AMR Solver Example: releasedExamples/AMRPoisson/execPETSc

```
Solve(Vector<DisjointBoxLayout> grids, Vector<LevelData<FArrayBox> *> phi, Vector<LevelData<FArrayBox> *> rhs)
{
 PetscCompGridPois petscop(0.,-1.,s_order);
 RefCountedPtr<ConstDiriBC> bcfunc =
                       RefCountedPtr<ConstDiriBC>(new
                       ConstDiriBC(1,petscop.getGhostVect()));
 BCHolder bc(bcfunc);
 petscop.define( cdomains, grids, refratios, bc, cdx*RealVect::Unit );
 ierr = petscop.createMatrix(); CHKERRQ(ierr);
 Mat A = petscop.getMatrix();
 ierr = MatGetVecs(A,&x,&b); CHKERRQ(ierr);
 ierr = petscop.putChomboInPetsc(rhs,b); CHKERRQ(ierr);
 ierr = KSPCreate(PETSC_COMM_WORLD, &ksp); CHKERRQ(ierr);
 ierr = KSPSetOperators(ksp, A, A); CHKERRQ(ierr);
 ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
 ierr = KSPSolve(ksp, b, x); CHKERRQ(ierr);
 ierr = KSPDestroy(&ksp); CHKERRQ(ierr);
 ierr = a_petscop.putPetscInChombo(x, phi); CHKERRQ(ierr);
}
```

Polytropic Gas Example

- Demonstrates integration of conservative laws (e.g., the Euler equations of gas dynamics) on an AMR grid hierarchy.
- Uses explicit unsplit, second-order Godunov method.
- One of the released examples in Chombo distribution
- Look under $CHOMBO_HOME/releasedExamples/AMRGodonov/execPolytropic
- Source code:
    - AMRLevel specialized for this set of problems in ../srcPolytropic
    - Main in ./amrGodunov.cpp
    - We use TutorialRamp.inputs to provide runtime parameters
- Problem is a Mach 10 oblique shock impacting a rigid planar surface.
    - Solution has curved shocks, rarefactions, and mach lines
    - classic experiment and code stress problem
        - *2-D double Mach reflection shock wedge test*

Parameters and late time results

Length of the run
- godunov.max_step = 100
- godunov.max_time = 0.064

Shape of the patch
- # godunov.num_cells = 32 8 4
- godunov.num_cells = 64 16 8

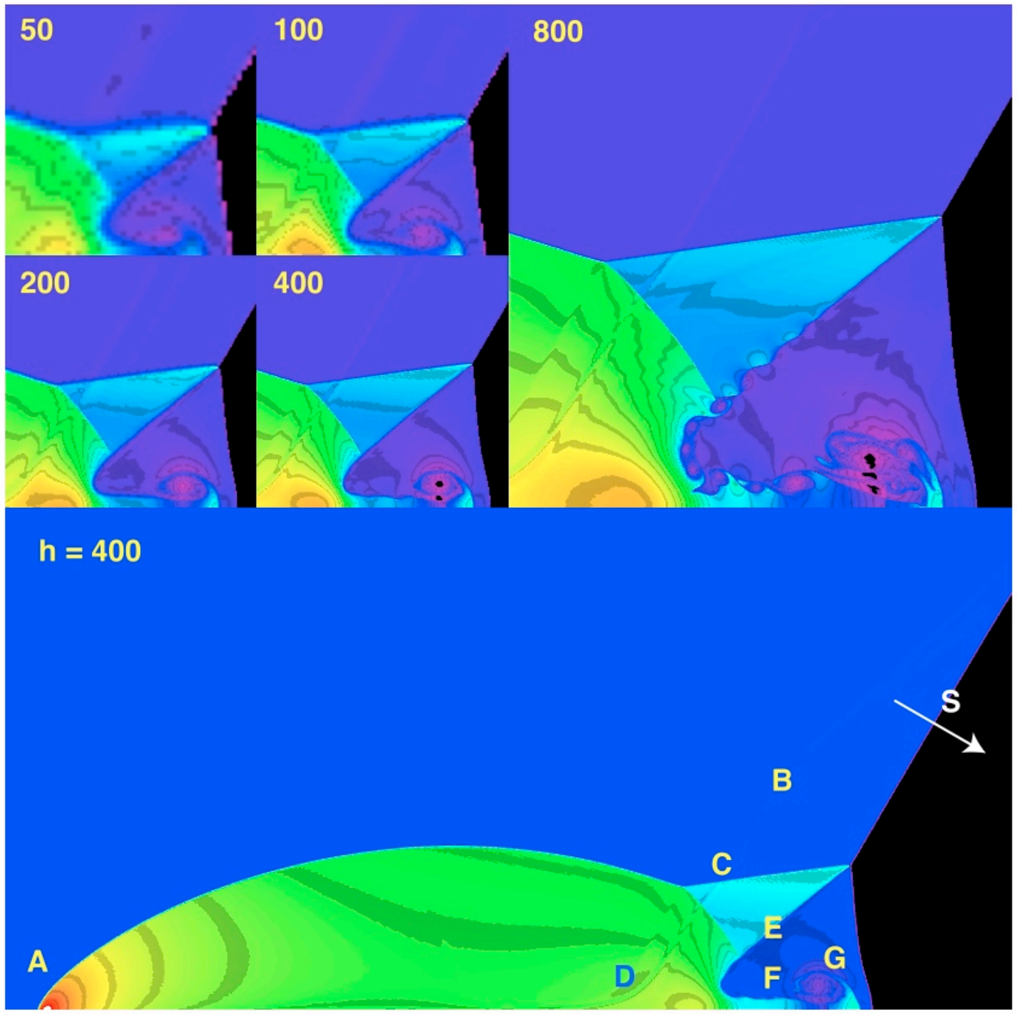
Grid refinement parameters
- godunov.max_level = 2

# For 2D
- godunov.ref_ratio = 4 4 4 4 4

# For 3D
- # godunov.ref_ratio = 2 2 2 2 2

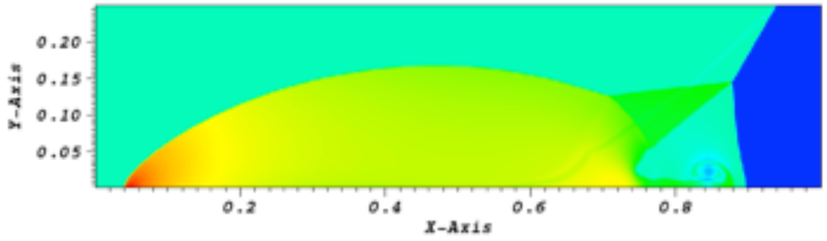
Regridding parameters
- godunov.regrid_interval = 2 2 2 2 2 2
- godunov.tag_buffer_size = 3
- godunov.refine_thresh = 0.015

Grid generation parameters
- godunov.block_factor = 4
- godunov.max_grid_size = 32
- godunov.fill_ratio = 0.75

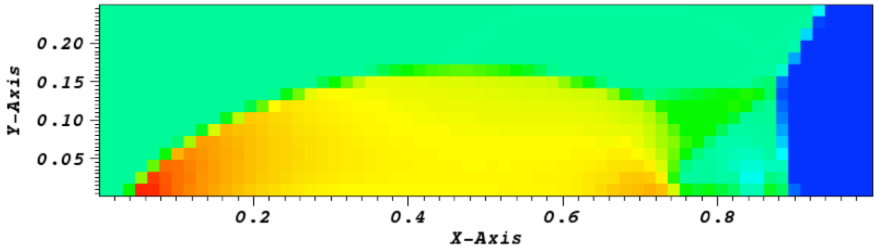


Experimenting with Parameters

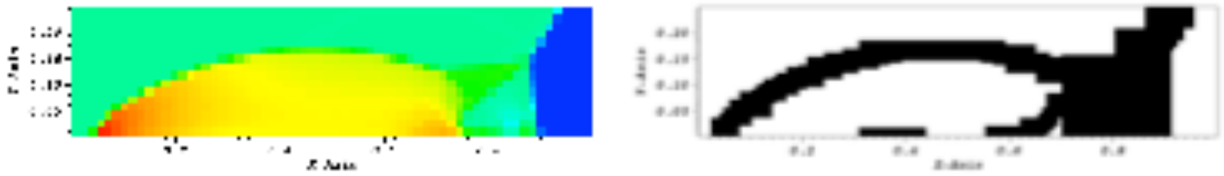
Default



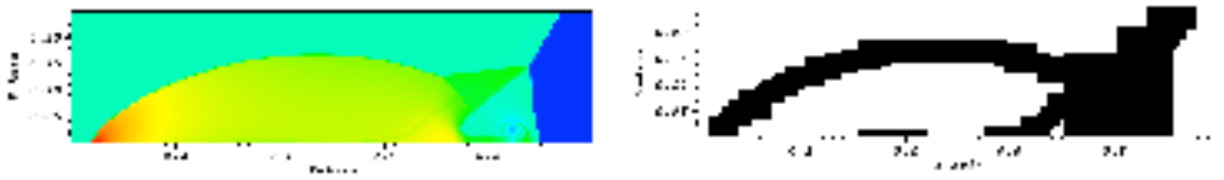
Change ref_ratio to 2



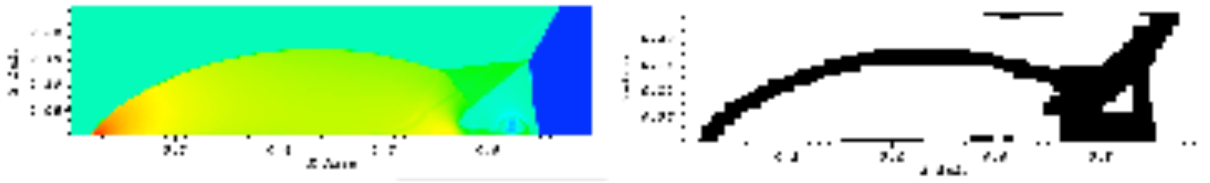
2 levels, refinement ratio of 2



2 levels, refinement ratio of 4



4 levels, refinement ratio of 2



Variations in output with change in refinement parameters