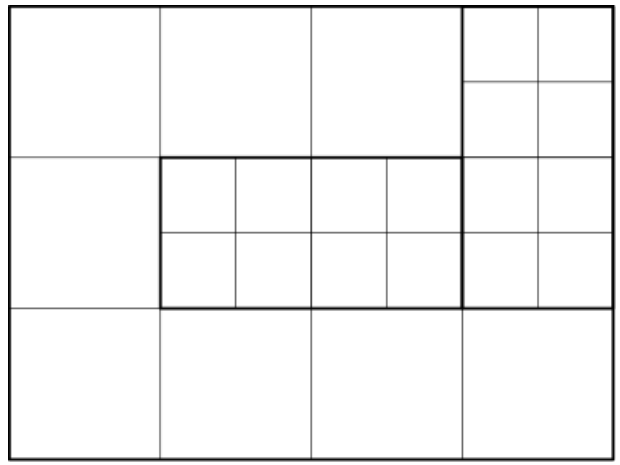
Freie Universität Berlin



# Ein Vergleich von SAMRAI und AMReX

Maikel Nadolski
Freie Universität Berlin

28. Februar 2019

Beide Bibliotheken haben überschneidende aber unterschiedliche Daten Verwaltung für die gleichen Konzepte.

**Dimensionalität**

| SAMRAI_MAXIMUM_DIMENSION | AMREX_SPACEDIM |
|---|---|

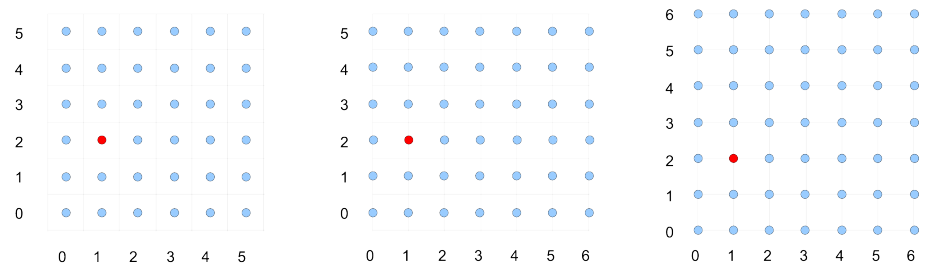
Freie Universität Berlin

## Indexraum

| | |
|---|---|
| SAMRAI::hier::BlockId | |
| SAMRAI::hier::Index | amrex::IntVect |
| SAMRAI::hier::IntVector | |
| | amrex::IndexType |
| SAMRAI::hier::Box | amrex::Box |
| SAMRAI::hier::BoxContainer | amrex::BoxArray |
| SAMRAI::hier::BoxLevel | amrex::BoxArray<br>+ amrex::DistributionMapping |

# Polymorphe AMReX Boxen

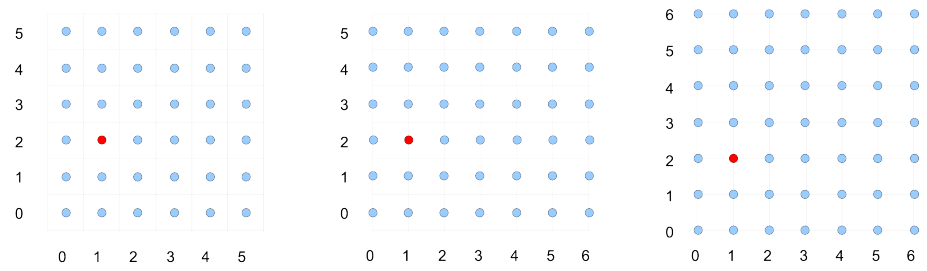
Boxen in AMReX sind in ihrem Indextyp polymorph.




```
1 // By default, Box is cell-centered.
2 amrex::Box cell_box({0, 0}, {5, 5});
3 // Construct a face-centered Box with normal X direction
4 amrex::Box face_box({0, 0}, {6, 5}, {1, 0});
5 // Construct a nodal Box.
6 amrex::Box node_box({0, 0}, {6, 6}, {1, 1});
7
8 // Conversion is possible
9 assert(amrex::convert(cell_box, {1, 0}) == face_box);
```

# Polymorphe AMReX Boxen

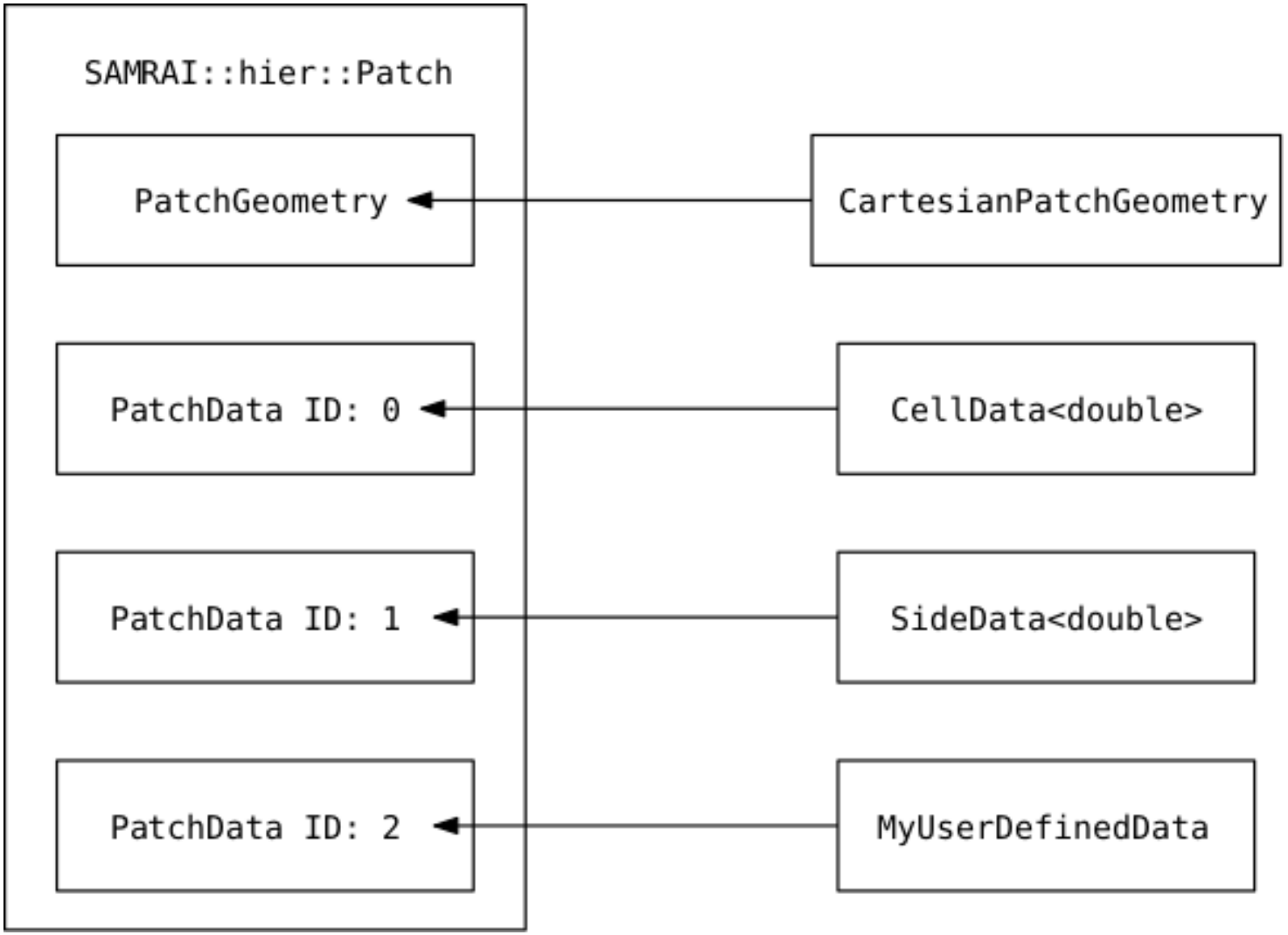
Boxen in AMReX sind in ihrem Indextyp polymorph.



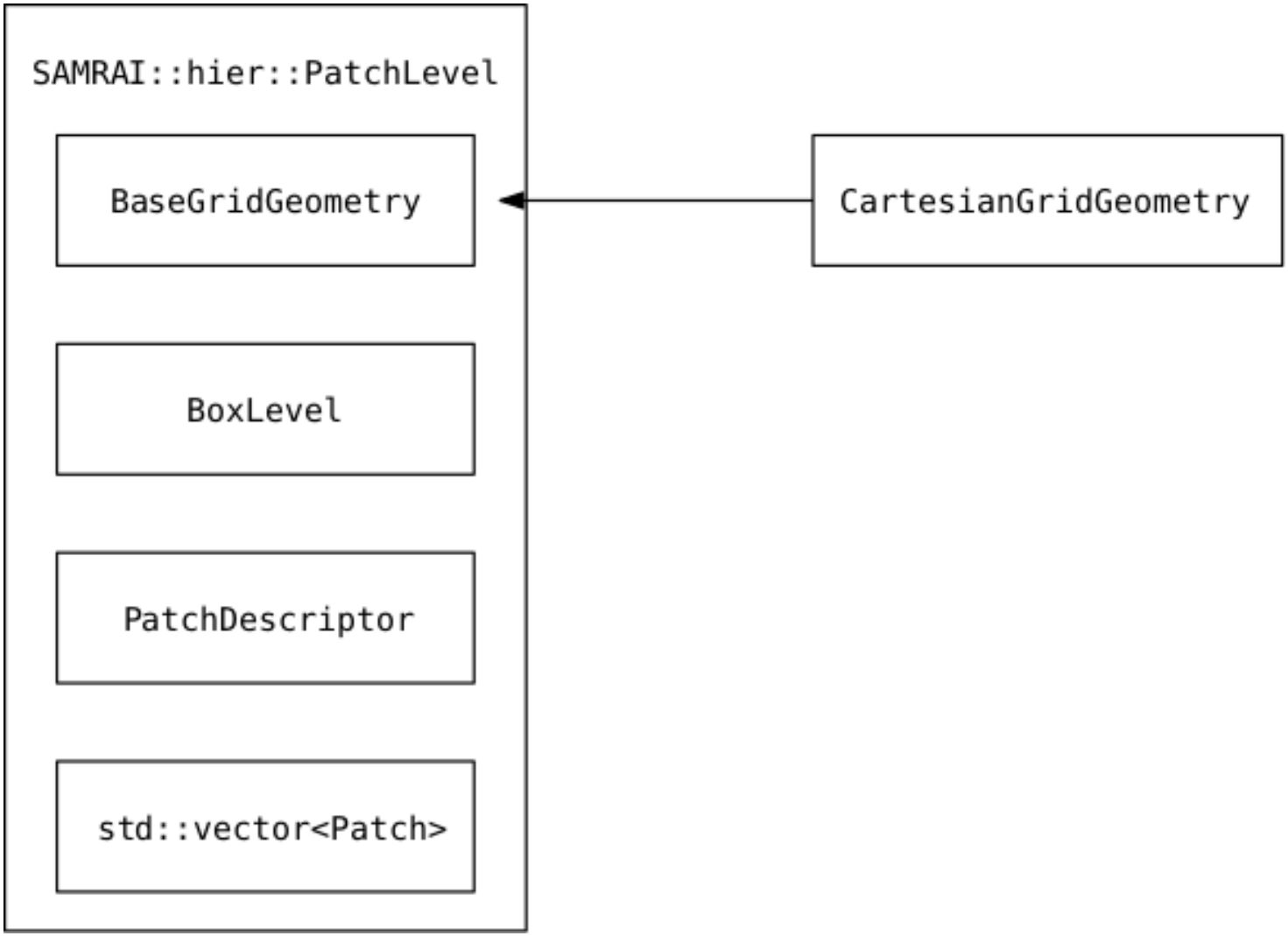

```
1 // By default, Box is cell-centered.
2 amrex::Box cell_box({0, 0}, {5, 5});
3 // Construct a face-centered Box with normal X direction
4 amrex::Box face_box({0, 0}, {6, 5}, {1, 0});
5 // Construct a nodal Box.
6 amrex::Box node_box({0, 0}, {6, 6}, {1, 1});
7
8 // Conversion is possible
9 assert(amrex::convert(cell_box, {1, 0}) == face_box);
```

*SAMRAI::hier::Box ist immer Zell-zentriert.*

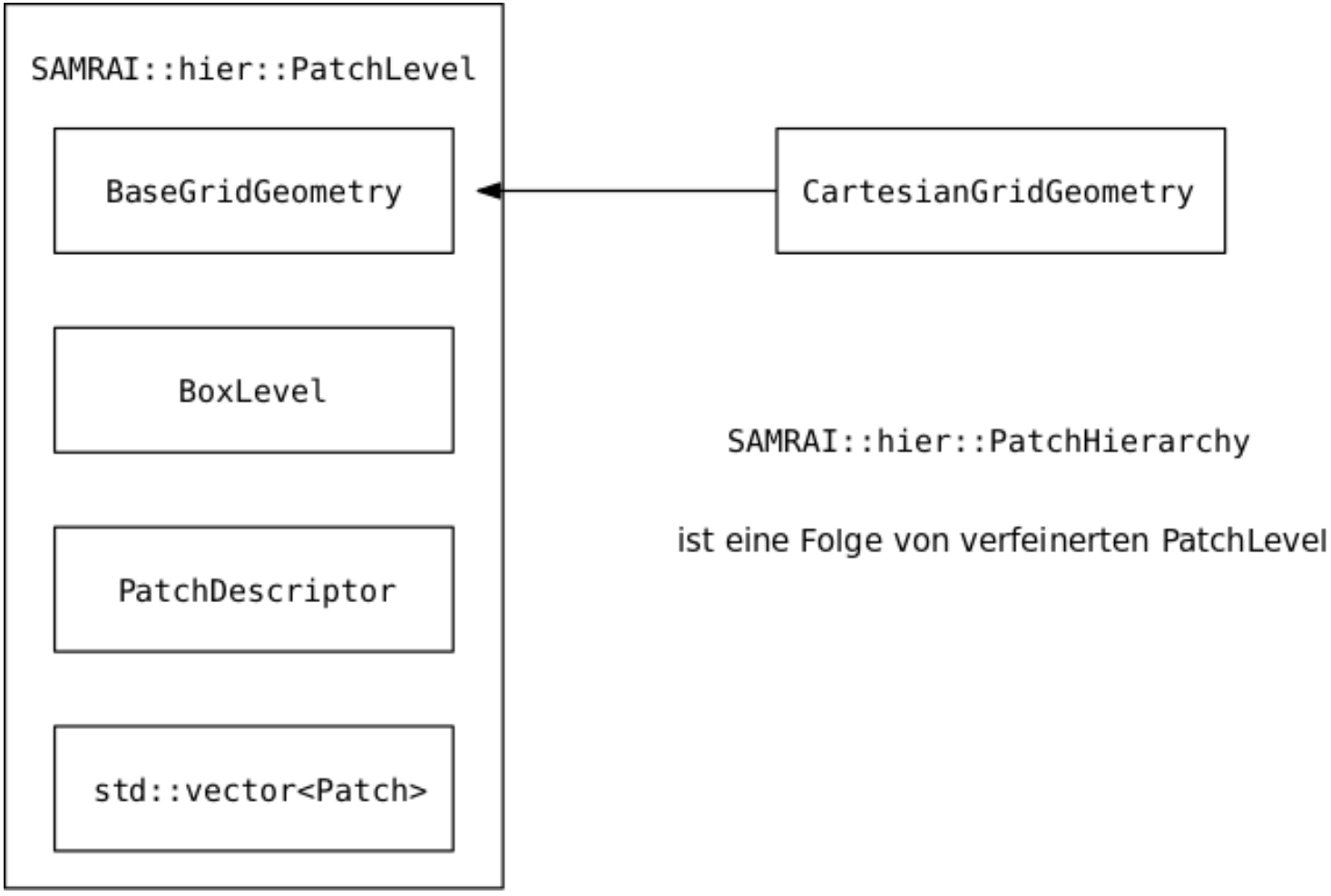
# Daten Verwaltung



# Daten Verwaltung



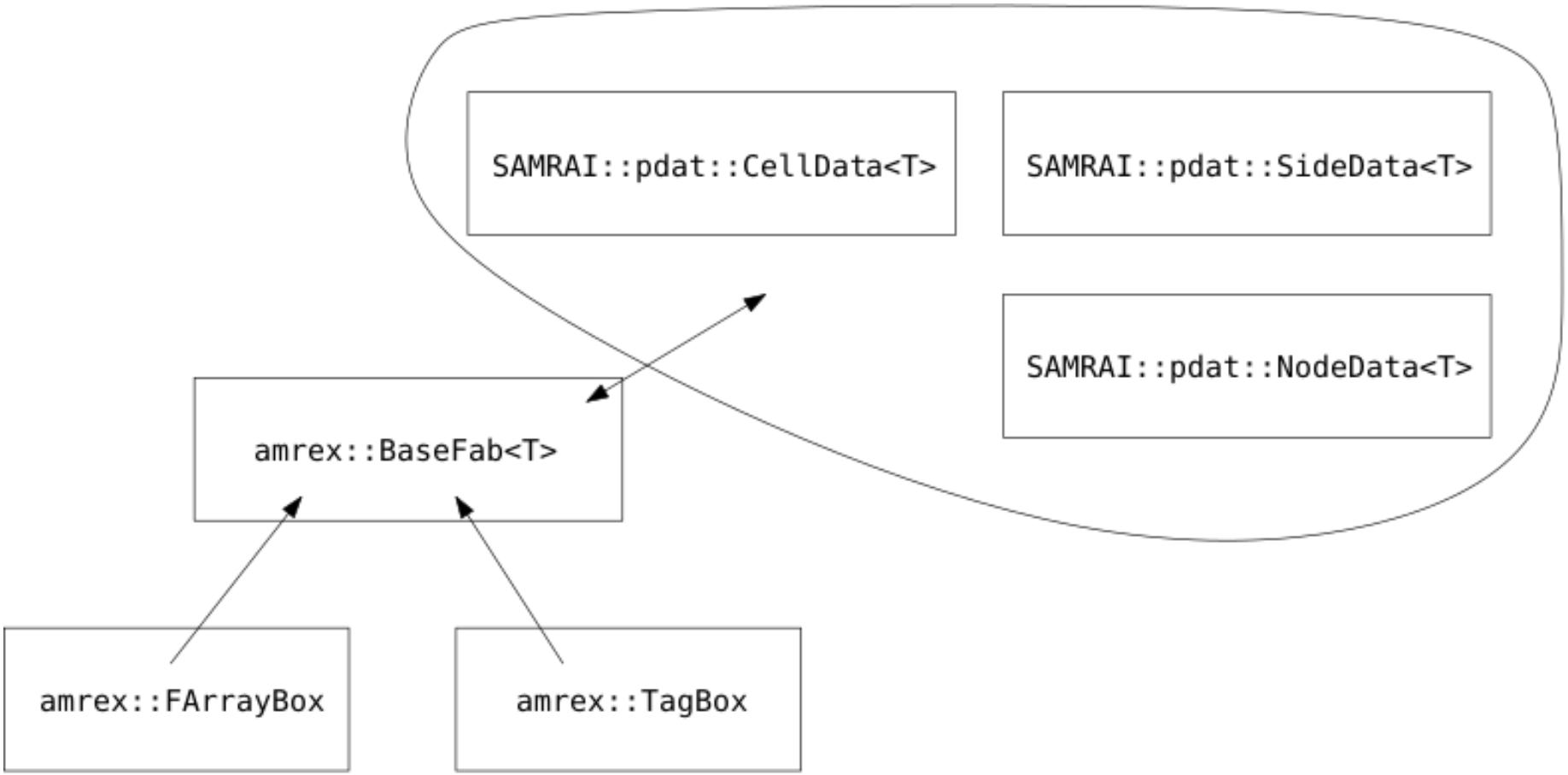
# Daten Verwaltung




SAMRAI::hier::PatchHierarchy

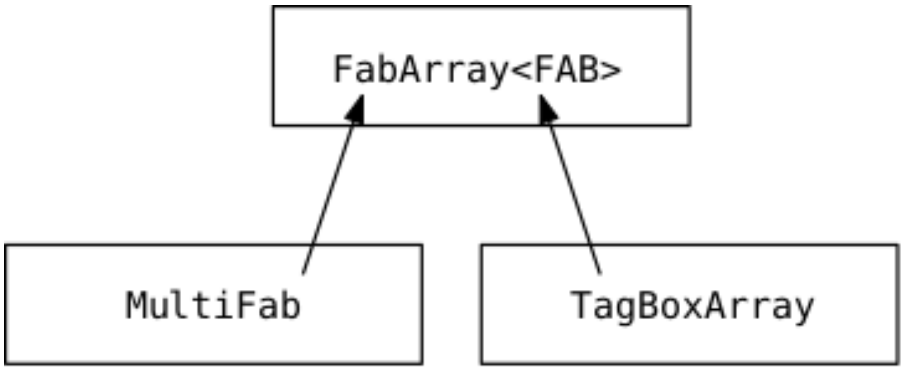
ist eine Folge von verfeinerten PatchLevel

# Daten Verwaltung




BaseFab<T> übernimmt in AMReX die Rolle von PatchData Typen in SAMRAI

# Daten Verwaltung

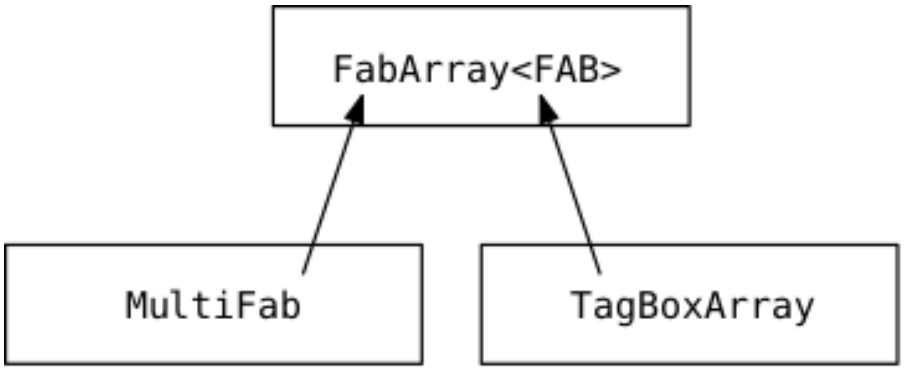



`FabArray` definiert Patch Daten für alle lokalen Boxen eines `(BoxArray, DistributionMapping)`-Paares.

```
1 // ba is amrex::BoxArray
2 // dm is amrex::DistributionMapping
3 int ncomp = 4;
4 int ngrow = 1;
5 amrex::MultiFab mf(ba, dm, ncomp, ngrow);
```

`mf` hat eine Ghost-Zellweite von 1 in jede Raumrichtung und 4 Komponenten.

# Daten Verwaltung




`FabArray` definiert Patch Daten für alle lokalen Boxen eines `(BoxArray, DistributionMapping)`-Paares.

```
1 // ba is amrex::BoxArray
2 // dm is amrex::DistributionMapping
3 int ncomp = 4;
4 int ngrow = 1;
5 amrex::MultiFab mf(ba, dm, ncomp, ngrow);
```

`mf` hat eine Ghost-Zellweite von 1 in jede Raumrichtung und 4 Komponenten.

`amrex::Geometry` entspricht der `SAMRAI::geom::CartesianGridGeometry` und wird unabhängig von den `MultiFabs` einer Applikation verwaltet.

# Daten Verwaltung

In SAMRAI ist es viel komplizierter Datenarrays für ein BoxLevel anzulegen! Das liegt an der starken Koppelung zu SAMRAI::hier::VariableDatabase.

```
1  SAMRAI::tbox::Dimension dim(2);
2  int ncomp = 4;
3  int ngrow = 1;
4
5  // VariableDatabase is a Singleton object
6  SAMRAI::hier::VariableDatabase* vardb = SAMRAI::hier::VariableDatabase::GetDatabase();
7
8  // Register Cell Variable + Context with ncomp components and given ghost cell width
9  std::shared_ptr<SAMRAI::hier::VariableContext> context =
10     vardb->getContext("Your_Unique_Module_Context_Name");
11
12 std::shared_ptr<SAMRAI::pdat::CellVariable<double>> mass =
13     std::make_shared<SAMRAI::pdat::CellVariable<double>>(dim, "mass", ncomp);
14
15 // This line crashes the application if the variable + context pair already exists.
16 int mass_id = vardb->registerVariableAndContext(
17     mass, context, std::vector<int>{ngrow, ngrow});
18
19 // box_level is SAMRAI::hier::BoxLevel
20 // geom is std::shared_ptr<BaseGridGeometry>
21
22 SAMRAI::hier::PatchLevel level(box_level, geom);
23 level.allocatePatchData(mass_id);
```

# Gitter Generieren

Wie werden in AMReX Gitter generiert?

# Gitter Generieren

Wie werden in AMReX Gitter generiert?
Durch Vererbung von `amrex::AmrCore`!

# Gitter Generieren

Wie werden in AMReX Gitter generiert?
Durch Vererbung von `amrex::AmrCore`!

```
1  //! Tag cells for refinement.  TagBoxArray tags is built on level lev grids.
2  virtual void ErrorEst (int lev, TagBoxArray& tags, Real time,
3                         int ngrow) override = 0;
4
5  //! Make a new level from scratch using provided BoxArray and DistributionMapping.
6  //! Only used during initialization.
7  virtual void MakeNewLevelFromScratch (int lev, Real time, const BoxArray& ba,
8                                        const DistributionMapping& dm) override = 0;
9
10 //! Make a new level using provided BoxArray and DistributionMapping and fill
11 //  with interpolated coarse level data.
12 virtual void MakeNewLevelFromCoarse (int lev, Real time, const BoxArray& ba,
13                                      const DistributionMapping& dm) = 0;
14
15 //! Remake an existing level using provided BoxArray and DistributionMapping
16 //  and fill with existing fine and coarse data.
17 virtual void RemakeLevel (int lev, Real time, const BoxArray& ba,
18                           const DistributionMapping& dm) = 0;
19
20 //! Delete level data
21 virtual void ClearLevel (int lev) = 0;
```

# Gitter Generieren

Wie werden in SAMRAI Gitter generiert?

# Gitter Generieren

Wie werden in `SAMRAI` Gitter generiert?
Durch verwendung von `SAMRAI::mesh::GriddingAlgorithm` und Vererbung von `SAMRAI::mesh::TagAndInitializeStrategy`!

# Gitter Generieren

Wie werden in SAMRAI Gitter generiert?
Durch verwendung von SAMRAI::mesh::GriddingAlgorithm und Vererbung von SAMRAI::mesh::TagAndInitializeStrategy!

```
1  //! Initialize data on a new level after it is inserted into an AMR patch
2  //! hierarchy by the gridding algorithm.
3  virtual void
4  initializeLevelData(const std::shared_ptr<hier::PatchHierarchy>& hierarchy,
5                      const int level_number, const double init_data_time,
6                      const bool can_be_refined, const bool initial_time,
7                      const std::shared_ptr<hier::PatchLevel>& old_level,
8                      const bool allocate_data) = 0;
9
10 //! Set integer tags to "one" on the given level to identify where refinement of
11 //! that level should occur.
12 virtual void
13 tagCellsForRefinement(const std::shared_ptr<hier::PatchHierarchy>& hierarchy,
14                       const int level_number, const int regrid_cycle,
15                       const double regrid_time, const int tag_index,
16                       const bool initial_time, const bool coarsest_sync_level,
17                       const bool can_be_refined,
18                       const double regrid_start_time) = 0;
19
20 //! After hierarchy levels have changed and data has been initialized on the new
21 //! levels, this routine can be used to reset any information needed by the
22 //! solution method that is particular to the hierarchy configuration.
23 virtual void resetHierarchyConfiguration(
24     const std::shared_ptr<hier::PatchHierarchy>& hierarchy,
25     const int coarsest_level, const int finest_level) = 0;
```

# Daten Kommunikation

In AMReX gibt es freie Funktionen für die Datenkommunikation von Ghost-Zellen.

```
1  void FillPatchSingleLevel(MultiFab& mf, Real time, const Vector<MultiFab*>& smf,
2                            const Vector<Real>& stime, int scomp, int dcomp,
3                            int ncomp, const Geometry& geom,
4                            PhysBCFunctBase& physbcf, int bcfcomp);
5
6  void FillPatchTwoLevels(MultiFab& mf, Real time, const Vector<MultiFab*>& cmf,
7                          const Vector<Real>& ct, const Vector<MultiFab*>& fmf,
8                          const Vector<Real>& ft, int scomp, int dcomp, int ncomp,
9                          const Geometry& cgeom, const Geometry& fgeom,
10                         PhysBCFunctBase& cbc, int cbccomp, PhysBCFunctBase& fbc,
11                         int fbccomp, const IntVect& ratio, Interpolater* mapper,
12                         const Vector<BCRec>& bcs, int bcscomp,
13                         const InterpHook& pre_interp = NullInterpHook(),
14                         const InterpHook& post_interp = NullInterpHook());
```

# Daten Kommunikation

In SAMRAI ist die Kommunikation in `SAMRAI::xfer::RefineAlgorithm` und `SAMRAI::xfer::RefineSchedule` geteilt.

```
1  std::vector<int> data_ids{/* ... */};
2  std::vector<int> scratch_ids{/* ... */};
3  assert(data_ids.size() == scratch_ids.size());
4  const int n_components = data_ids.size();
5
6  // You do not need to rebuild the algorithm if a hierarchy is regrid
7  auto algorithm = std::make_shared<SAMRAI::xfer::RefineAlgorithm>();
8  for (int component = 0; component < n_components; ++component) {
9    algorithm->registerRefine(
10       scratch[component], data_ids[component], scratch[component],
11       std::make_shared<SAMRAI::pdat::CellDoubleConstantRefine>());
12 }
13
14 // Whenever you change a PatchLevel you have to rebuild the RefineSchedule
15 std::shared_ptr<SAMRAI::hier::PatchLevel> = patch_level = hierarchy->getPatchLevel(level);
16 std::shared_ptr<SAMRAI::xfer::RefineSchedule> schedule =
17         algorithm->createSchedule(patch_level, level - 1, hierarchy, &boundary_condition);
18
19 // Whenever you want to fill the ghost layer
20 schedule->fillData(time_point);
```

# Daten Kommunikation

In SAMRAI scheint es auf den ersten Blick komplizierter zu sein, aber die Parameter für die freien Funktionen `amrex::FillPatchSingleLevel` und `amrex::FillPatchTwoLevels` zu sammeln, macht auch Arbeit.

```
1  void HyperbolicSplitIntegratorContext::FillGhostLayerSingleLevel(
2      int level, Direction dir, BoundaryCondition boundary) {
3    ::amrex::Vector<::amrex::BCRec> bcr(2 * AMREX_SPACEDIM);
4    ::amrex::MultiFab& scratch = GetScratch(level, dir);
5    const int nc = scratch.nComp();
6    const ::amrex::Vector<::amrex::MultiFab*> smf{&GetData(level)};
7    const ::amrex::Vector<double> stime{GetTimePoint(level, dir).count()};
8    const ::amrex::Geometry& geom = GetGeometry(level);
9    AdaptBoundaryCondition condition(boundary, geom, level);
10   ::amrex::FillPatchSingleLevel(scratch, stime[0], smf, stime, 0, 0, nc, geom,
11                                 condition, 0);
12 }
```

# Daten Kommunikation

In SAMRAI scheint es auf den ersten Blick komplizierter zu sein, aber die Parameter für die freien Funktionen `amrex::FillPatchSingleLevel` und `amrex::FillPatchTwoLevels` zu sammeln, macht auch Arbeit.

```
1  void HyperbolicSplitIntegratorContext::FillGhostLayerTwoLevels(
2      int fine, int coarse, Direction dir, BoundaryCondition boundary) {
3    FUB_ASSERT(coarse >= 0 && fine > coarse);
4    ::amrex::Vector<::amrex::BCRec> bcr(2 * AMREX_SPACEDIM);
5    ::amrex::MultiFab& scratch = GetScratch(fine, dir);
6    const int nc = scratch.nComp();
7    const ::amrex::Vector<::amrex::MultiFab*> cmf{&GetData(coarse)};
8    const ::amrex::Vector<::amrex::MultiFab*> fmf{&GetData(fine)};
9    const ::amrex::Vector<double> ct{GetTimePoint(coarse, dir).count()};
10   const ::amrex::Vector<double> ft{GetTimePoint(fine, dir).count()};
11   const ::amrex::Geometry& cgeom = GetGeometry(coarse);
12   const ::amrex::Geometry& fgeom = GetGeometry(fine);
13   const ::amrex::IntVect ratio =
14       GetRefineRatioToCoarserLevel(fine) * ::amrex::IntVect::TheUnitVector();
15   ::amrex::Interpolater* mapper = &::amrex::pc_interp;
16   AdaptBoundaryCondition fine_condition(boundary, fgeom, fine);
17   AdaptBoundaryCondition coarse_condition(boundary, cgeom, coarse);
18   ::amrex::FillPatchTwoLevels(scratch, ft[0], cmf, ct, fmf, ft, 0, 0, nc, cgeom,
19                               fgeom, coarse_condition, 0, fine_condition, 0,
20                               ratio, mapper, bcr, 0);
21 }
```

# Daten Kommunikation

Gitterdaten werden in jedem Zeitschritt von den feineren Leveln vergröbert. In AMReX gibt es für diesen Zweck verschiedene averave_down_XXX Funktionen.

```
1  void HyperbolicSplitIntegratorContext::CoarsenConservatively(int fine_level,
2                                                                int coarse_level,
3                                                                Direction dir) {
4    const int first =
5        GetPatchHierarchy()->GetDataDescription().first_cons_component;
6    const int size = GetPatchHierarchy()->GetDataDescription().n_cons_components;
7    ::amrex::average_down(GetScratch(fine_level, dir),
8                          GetScratch(coarse_level, dir), GetGeometry(fine_level),
9                          GetGeometry(coarse_level), first, size, 2);
10 }
```

# Daten Kommunikation

In SAMRAI ist die Kommunikation in `SAMRAI::xfer::CoarsenAlgorithm` und `SAMRAI::xfer::CoarsenSchedule` geteilt.

```
1  std::vector<int> scratch_ids{/* ... */};
2  std::vector<int> cons_indices{/* indices into scratch_ids */};
3
4  // You do not need to rebuild the algorithm if a hierarchy is regrid
5  auto algorithm = std::make_shared<SAMRAI::xfer::CoarsenAlgorithm>(dim);
6  for (int cons : cons_indices) {
7    algorithm->registerCoarsen(
8        scratch[cons], scratch[cons],
9        std::make_shared<SAMRAI::geom::CartesianCellDoubleWeightedAverage>());
10 }
11
12 // Whenever you change a PatchLevel you have to rebuild the RefineSchedule
13 std::shared_ptr<SAMRAI::hier::PatchLevel> coarse = hierarchy->getPatchLevel(lvl - 1);
14 std::shared_ptr<SAMRAI::hier::PatchLevel> fine = hierarchy->getPatchLevel(lvl);
15 std::shared_ptr<SAMRAI::xfer::CoarsenSchedule> schedule =
16     algorithm.createSchedule(coarse, fine);
17
18 // Whenever you want to coarsen your data
19 schedule->fill(time_point);
```