Amanzi: Developers' Guide

November 20, 2024

AMANZI-SDG, Revision 1.0

United States Department of Energy

K. Lipnikov, LANL D. Moulton, LANL E. Coon, ORNL S. Naranjo, OSU

LA-UR-20-22491

DISCLAIMER

Table of Contents

1 Introduction

1.1 Purpose and Scope of this Document

This document provides short description of Amanzi libraries, summarizes main design concepts, provides step-by-step instructions and detailed comments for solving an abstract PDE, provides essentials on creation of unit tests and web-pages for the user guide.

The intended audience includes (a) new developers of Amanzi and (b) software developers who want to use the general purpose C++ Amanzi libraries in their work.

This document is not designed for the end-users who want to use Amanzi as a subsurface simulator for environmental applications.

2 Design of Amanzi

In short, the guiding principles described below are related to code readability, modularity, and extensibility. We facilitate community code development and code review, we follow closely, but not exactly, the Google C++ coding style (https://google.github.io/styleguide/cppguide.html). The Amanzi's version of the coding style is located here

`doc/standards/cxx/cppguide.html`

for more detail. In this section, we describe high-level principles and elaborate some of them in the subsequent section. We use different fonts to distinguish between a `Class` name, its *Methods()*, and its *variables*. Global CONSTANTS are capitalized.

Disclaimer. Amanzi's initial code implementation of new models and algorithms does not always comply with the formulated design principles, but it is getting there with each code re-factory.

2.1 State

State is a simple data manager. It allows process kernels (PK) to require, read, and write various variables (such as physical fields). It guarantees data protection by providing both const and non-const data pointers for variables. It provides some initialization capability – this is where all independent variables can be initialized – since independent variables are typically owned by the state, not by a process kernel. A few initialization tools are supported: a space-time function, initialization from an Exodus file and initialization from an HDF5 file.

2.2 PK and MPC PK

PK stands for the Process Kernel. MPC stands for the Multi-Process Coupler. Each PK and MPC PK does little actual numerical work. Instead, PK administrates discretization schemes, time integrators, and solvers. Each PK may represent a single equation (e.g. the Poisson equation for the Darcy flow) or system of strongly connected equations (e.g. the Navier-Stokes flow).

An MPC PK couples multiple physical processes which have their respected PKs. One example is the Darcy flow and dispersive transport of chemical components. An MPC PK may often be fully automated with no knowledge of the underlying PKs. Since an MPC PK has the same interface as a PK, it is also a process kernel which allows us to build a hierarchy of physical models with various degree of coupling ranging from a weak coupling to an iterative coupling to a strong coupling.

Much of the work in a PK is delegated to field evaluators, which implement various physical and mathematical models, such as the equations of state, or boundary conditions, or mesh deformation. For these reasons, it is appropriate to call them variable evaluators. The available variable evaluators are classified as follows:

1. Independent variable evaluators are the user-provided functions of spatial and temporal coordinates and has no dependencies. They could be used to compute (analytic or tabulated) boundary conditions, source terms, and initial conditions.

2. Primary variable evaluators are related to the fields solved for within a PK. Examples are pressure and temperature fields. Typically these evaluators are used internally to track change in fields state and inform the dependency tree about this.

3. Secondary variable evaluators are derived either from primary variable evaluators or other secondary variables. There are two types of the secondary variable evaluators used to evaluate either a single or multiple variables. A model for a secondary variable can be anything from a constitutive relation to a discrete operator (apply a divergence operator to a velocity given a mesh and discretization) to a summation operator (add the divergence of Darcy fluxes to a source term to determine the mass balance). Quite often, the secondary field/variable evaluators are created by high-level PKs during the setup phase and inserted automatically in the list of evaluators.

The evaluator is much like a functor or function; it stores no actual data, only meta-data and a few parameters or constants. It accesses data using a data manager, which controls access for both read-only and read/write modes.

All evaluators are stored in a dependency graph, which is a directed, acyclic graph (DAG) describing the functional relationship of each variable in the state. End nodes in the dependency graph are either independent variables or primary variables. All other nodes in the graph are secondary variables.

The combination of a data manager and a dependency graph enables dynamic definition of each variable's model and data, and splits complex equations into manageable chunks. It also allows lazy evaluation, where nodes in the graph are updated (re-calculated) only if their dependencies have changed, resulting in a managed, automated evaluation process with fewer bugs and inefficiencies. For more details, we refer to **?**. A developer may easily modify behavior of evaluators by overriding virtual member functions. The example below shows implementation of one function in an abstract product evaluator of type $\Pi_{i=1}^{N} f_i^{p_i}$ where p_i is either 1 or -1.

```
void ProductEvaluator::EvaluateField_(
    const Teuchos::Ptr<State>& S,
    const Teuchos::Ptr<CompositeVector>& result)
{
  auto& result_c = *result->ViewComponent("cell");
  int ncells = result_c.MyLength();

  int n(0);
  result_c.PutScalar(1.0);
  for (auto it = dependencies_.begin(); it != dependencies_.end(); ++it) {
    const auto& field = *S->GetFieldData(*it)->ViewComponent("cell");
    if (powers_[n] == 1)
      for (int c = 0; c != ncells; ++c) result_c[0][c] *= field[0][c];
    else if (powers_[n] == -1)
      for (int c = 0; c != ncells; ++c) result_c[0][c] /= field[0][c];
    n++;
  }
}
```

2.3 CompositeVector

Class `CompositeVector` is an implementation of an improved `Epetra_MultiVector` (from Trilinos suite of packages) which spans multiple components and knows how to communicate itself. A composite vector is a collection of vectors defined on a common mesh and communicator. Each vector, or component, has a name (typically, a mesh entity) and a number of degrees of freedom. This meta data is stored in class `CompositeVectorSpace`. For instance, the field *total_component_concentration* is the cell-centered field with as many degrees of freedom as there exist chemical components.

Ghost cell updates are managed by the class `CompositeVector`. Design of the parallel communication strategy is driven by two observations:

- The need for updated ghost cell information is typically known by the user just prior to being used, not just after the master values are updated.
- Occasionally multiple functions need ghost values, but no changes to owned data have been made between these functions. However, it is not always possible for the second call to know, for certain, that the first call did the communication. Versatility means many code paths may be followed.

2.3.1 Parallel communications

To avoid unnecessary parallel communication the following algorithms were implemented but are not active now. This may change in the future.

Each time the vector values are changed, an internal flag is marked to record that the ghost values are stale. Each time ghost cells are needed, that flag is checked and communication is done, if needed. Keeping this flag correct is therefore critical. To do this, access to vectors must follow the rigid pattern. The following modifications tag the flag:

1. Any of the usual *PutScalar()*, *Apply()*, etc methods.
2. Non-const calls of *ViewComponent()*.
3. Call of *GatherMasterToGhosted()* and *ChangedValues()*.
4. *Scatter()* called in a non-INSERT mode.

There exist known ways to break this paradigm. One is to store a non-const pointer to the underlying `Epetra_MultiVector`. The fix is simple as this: never store a pointer to the underlying data, just keep pointers to the composite vector itself.

The other one is when one grabs a non-const pointer, calls *Scatter()*, then changes the values of the local data. This is the nasty one, because it is both subtle and reasonable usage. When you access a non-const pointer, the data is flagged as changed. Then you call *Scatter()* and the data is flagged as unchanged. Then you change the data from your old non-const pointer, so that the data is changed, but not flagged. The first fix is to always call *ViewComponent()* after *Scatter()*

and before changing values. Another way to protect yourself is to put non-const references in their own scope. For instance, the following practice is encourage:

```
CompositeVector my_cv;
{ // unnamed scope for my_vec
  Epetra_MultiVector& my_vec = *my_cv.ViewComponent("cell", false);
  my_vec[0][0] = 12;
} // close scope of my_vec

my_cv.ScatterMasterToGhosted()

// Reference to my_vec is now gone, so we cannot use it and screw things up!

{ // unnamed scope for my_vec
  // This is now safe!
  Epetra_MultiVector& my_vec = *my_cv.ViewComponent("cell", true);
  my_vec[0][0] = my_vec[0][ghost_index] + ...
} // close scope of my_vec
```

The final way to break the parallel machinery is to use *const_cast()* and then change the values. Const-correctness is your friend. Keep your PKs const-correct, and you will never have this problem.

Note that the non-INSERT modes of scatter are never skipped because of the flag state, and the flag is always tagged as changed. This is because subsequent calls with different modes would break the code.

2.4 TreeVector

The class `TreeVector` implements a nested, hierarchical data structure that mimics that for PK hierarchies. It is an extendable collection of composite vectors. This vector allows each physical PK to use composite vector to store their solution, and allows MPCs to push back vectors in a tree format.

This class provides the standard vector interface (extended ring algebra) and may be used with time integrators and nonlinear solvers.

2.5 Linear operators

The idea behind the design of Amanzi operators is to separate three functionalities that are frequently placed in a single class in other C++ packages.

1. Containers of local matrices (classes prefixed with `Op`) and data layout *schemas*.
2. Linear operators and elemental operations with them: assembly of a global matrix (e.g. Jacobian), matrix-vector product, inversion, and calculation of the Schur complement.
3. Discrete PDEs: populate values in local matrices, add nonlinear coefficients, create specialized preconditioners, and impose special boundary conditions.

2.5.1 Op

Class `Op_*` is a container of local matrices. A series of such classes (e.g. `Op_Cell_FaceCell` and `Op_Cell_Schema`) handle data layout. The second word in the class name indicates the container size (the number of mesh cells here). The third word specifies location of degrees of freedom: in cells and on faces in the first example; specified by a complex schema in the second example. These are really just structs of vectors of dense matrices of doubles, and simply provide a type. They are derived from the virtual class `Op`.

A key concept of an `Op` is the schema. The old design of the schema includes one enum representing the dofs associated with the Operator's domain and range, and one enum for the contained size. This is a major limitation for implementing complex discretization schemes. Also a single schema implies that the domain and range of the operator are the same. The new design (which is backward compatible) includes two schemas that are also more detailed. A the first enum is replaced with the list of enums to represent various possible collections of degrees of freedom including non-standard degrees of freedom as as derivatives and moments. Additional list specifies multiplicity of these degrees of freedom. The second enum in the new schema specifies (as before) the geometric entity over which the local matrices are assembled.

Existence of two (simple and complex) schemas provides some flexibility for code development. For instance, a developed could create surface matrices, and then assemble them into a subsurface matrix by introducing a new `Op` class (for surface discretization) with a simple schema. Alternatively, it could be done using the class `Op_Cell_Schema` with a complex schema.

The general schema makes it trivial to assemble a global matrix (e.g. in a coupled flow-energy system) from sub-block operators. Finally, the new schema supports rectangular matrices which is useful for saddle-point type systems.

`Op_*` works via a visitor pattern. Matrix assembly, *Apply()*, application of boundary conditions, and symbolic assembly are implemented by the virtual class `Operator` calling a dispatch to the virtual class `Op`, which then dispatches back to the derived class `Operator_*` so that type information of both the `Operator_*` (i.e. global matrix info) and the `Op_*` (i.e. local matrix info) are known.

A container of local matrices (i.e. instantiation of a `Op_*`) can be shared by multiple `Operator_*`. Sharing is indicated by the variable *ops_properties*. In combination with *CopyShadowToMaster()* and *Rescale()*, a developer has a room for a variety of optimized implementations. The key parameters have prefix OPERATOR_PROPERTY and described in file *Operators_Defs.hh*.

2.5.2 Operator

An operator represents a map from linear space X to linear space Y. Typically, this map is a linear map; however, it can be used also to calculate a nonlinear residual. The spaces X and Y are coded using class `CompositeVectorSpace`. A few concrete maps $X \to Y$ are already implemented in the code.

Typically the forward operator is applied using only local Ops. The inverse operator typically requires assembling a matrix, which may represent the entire operator or may be only its Schur complement.

The class `Operator` performs actions summarized in the second bullet above. Amanzi has a few derived classes such as `Operator_Cell`, `Operator_Node`, `Operator_FaceCellSff`, where the suffix `_X` indicates the specific map, see class `Operator_Schema` for a general map. These classes are derived from the virtual class `Operator` which stores a schema and a pointer to a global operator.

Concrete maps use the old schema which is an integer variable. Their are now superseded by the new flexible schema which is a class variable. Each operator stores a list of containers of local matrices, more precisely a list of pointers to the variables of class `Op`.

The only potentially confusing part is the use of the visitor pattern (i.e. double dispatch in this case) to resolve all types. For instance to assemble a matrix, we may use the following pseudocode

```
// Operator
AssembleMatrix(Matrix A) {
  for each op {
    op->AssembleMatrix(this, Matrix A);
  }
}

virtual AssembleMatrixOp(Op_Cell_FaceCell& op) {
  // throw error, not implemented
}

// Op
AssembleMatrix(Operator* global_op, Matrix& A) = 0;

// Op_Cell_FaceCell
AssembleMatrix(Operator* global_op, Matrix& A) {
  global_op->AssembleMatrixOp(*this, A);
}

// Operator_FaceCell
AssembleMatrixOp(Op_Cell_FaceCell& op, Matrix& A) {
  // This method now know both local schema and the matrix's dofs,
  // and assembles the face+cell local matrices into the matrix.
}
```

The reason for the double dispatch is to get the types specifically without a ton of statements like this one "if (schema == schema1) { assemble one way } else { assemble another way}".

2.5.3 PDE

A "discrete" PDE consists of (a) a single global operator, (2) an optional global assembled matrix, and (3) an un-ordered additive collection of lower-rank (or equal rank) local operators, hereafter called *ops*. During its construction, a PDE can grow by assimilating more *ops*. The global operator knows how (a) to perform the matrix-vector product, the corresponding function is called *Apply()*, and (b) to assemble *ops* into a global matrix. Each `PDE_*` class knows how to apply boundary conditions and to create a preconditioner.

The classes `PDE_Diffusion`, `PDE_Advection`,`PDE_Accumulation`, etc create operators of the specified type (for instance `Operator_FaceCell` or `Operator_Schema`), populate their

values, and apply boundary conditions. They are in some sense physics based generalization of operators and may perform complex actions such as an approximation of Newton correction terms.

A collection of PDEs that store a pointer to the same global operator form an additive PDE. Application of boundary conditions is done independently by each PDE in this collection. The result is gathered into a single right-hand side vector.

Discretization of a simple PDE (i.e. diffusion) is not done directly. Instead, a helper class that contains methods for creating and populating the *ops* within the `Operator` is used. The helper class can be used to discretize a simple PDE, such as the diffusion equation. A more complex PDE, such as the advection-diffusion equation, can be discretized by creating two "discrete" PDEs for diffusion and advection processes.

2.6 TreeOperator

Class `TreeOperator` is the block analogue of linear operators and provides a linear operator acting on a `TreeVectorSpace`. In short, it is a matrix of operators. Currently this structure is used for things like multiphase flows, thermal Richards, coupled matrix-fracture flow and transport, etc.

2.7 Linear solvers

Native and third-party solvers are handled through a single factory and the uniform interface. Direct and iterative solvers from Trilinos is a part of this factory. Native re-implementation of some iterative solvers supplied by Trilinos is due to lack of capabilities needed for subsurface solvers. One example is the neccisety to perform at least one iteration even when a norm of the linear residual is below the requested tolerance.

2.8 Nonlinear solvers

A factory of nonlinear solvers includes sever solvers ranging from the Newton method to inexact Newton methods to continuation methods. The solvers are templated on classes `Vector` and `VectorSpace`.

The nonlinear Krylov accelerator solvers **?** implements inexact Newton's method, where the correction equation of Newton's method is only approximately solved because the Jacobian matrix is approximated and/or the linear system is not solved exactly. Placed in the iteration loop, this black-box accelerator listens to the sequence of inexact corrections and replaces them with accelerated corrections; the resulting method is a type of accelerated inexact Newton method. Note that an inexact Newton iteration is merely a standard fixed point iteration for a preconditioned system, and so this accelerator is more generally applicable to fixed point iterations.

3 Selected Amanzi libraries

This section describes selected Amanzi libraries, their limitations and possible ways for extensions.

3.1 WhetStone

This library implements primarily local matrices for various discretizations frameworks including finite volumes, nonlinear finite volumes, mimetic finite differences, virtual elements, and discontinuous Galerkin. Conceptual design of a part of the library is presented in Fig. **??**. Classes derived from class `MFD3D` cover a huge spectrum of PDEs.

The library is under extensive development. At the moment, there exist both a unified approach to discretization schemes of arbitrary order and the optimized implementation of the low-order schemes.

Additional functionality included in this library supports

1. the ring algebra of scalar, vector and matrix polynomials;
2. quadrature rules on simplices;
3. numerical integration algorithms based on the Euler homogeneous theorem;
4. coordinate transformations including parameterization of mesh faces and edges.

A few comments on the design principles. Polynomial coefficients are represented by a linear array. A polynomial iterator class allow us to access information about monomial terms of a given polynomial in a for-type loop:

```
Polynomial poly(3, 2);
for (auto it = poly.begin(); it < poly.end(); ++it) {
  int i = it.PolynomialPosition();
  int k = it.MonomialSetOrder();
  const int* idx = it.multi_index();
  double ci = poly(i);
}
```

Each step of this loop extracts information about monomial $c_i\, x^{idx_0} y^{idx_1} z^{idx_2}$ of degree $k = idx_0 + idx_1 + idx_2$ in a quadratic polynomial.

Quadrature rules on simplexes have positive weights for stability of numerical schemes. Integration formulas based on Euler's homogeneous theorem can be used for integrating polynomials over polytopal cells. To integrate polynomial and non-polynomial functions using a single interface a simple base class `WhetStoneFunction` is used.

Coordinate transformation allows us to treat a 3D mesh face as a 2D polygon. This is used in (a) hierarchical construction of high-order virtual element and mimetic schemes, and (b) projection of polynomials on a low-dimension manifold and an reserve (non-unique) lifting operation.

Finally this library contains a factory of discretization schemes that could be extended by including users schemes via a simple interface. Example of such an extension is available in directory *operators/test*.

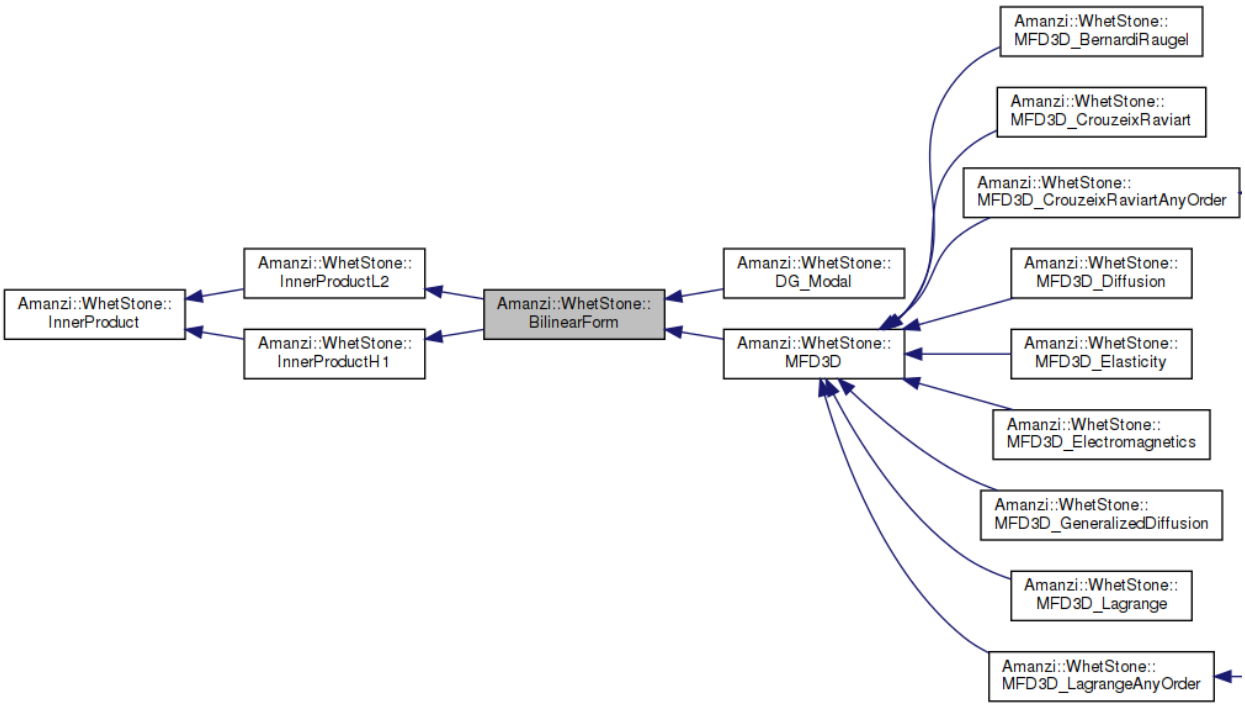

Figure 1: Partial dependency tree for library WhetStone.

3.2 Operators

This is a high-level library that supports global matrices and various assembly patterns. Conceptual design of a part of this library is presented in Fig. **??**. Classes derived from a helper class `PDE_HelperDiscretization` cover a range of parabolic and hyperbolic problems.

Additional functionality included in this library supports

1. cell-based remap schemes;
2. upwind algorithms for cell-centered fields;
3. reconstruction of slopes from cell-based data and their limiting.

To create a preconditioner from an assembled matrix, we need a contiguous vector space. Two classes `SuperMapLumped` and `SuperMap` in directory `data_structures` takes non-contiguous data structures, such as the `CompositeVector` and `TreeVector` and converts them into a single map. Unfortunately, un-rolling vectors requires to copy data using functions described in `OperatorUtils.hh`.

This library was re-factored a few times. Implementation of new schemes, most certainly will require an additional re-factory; however, backward compatibility should be preserved.

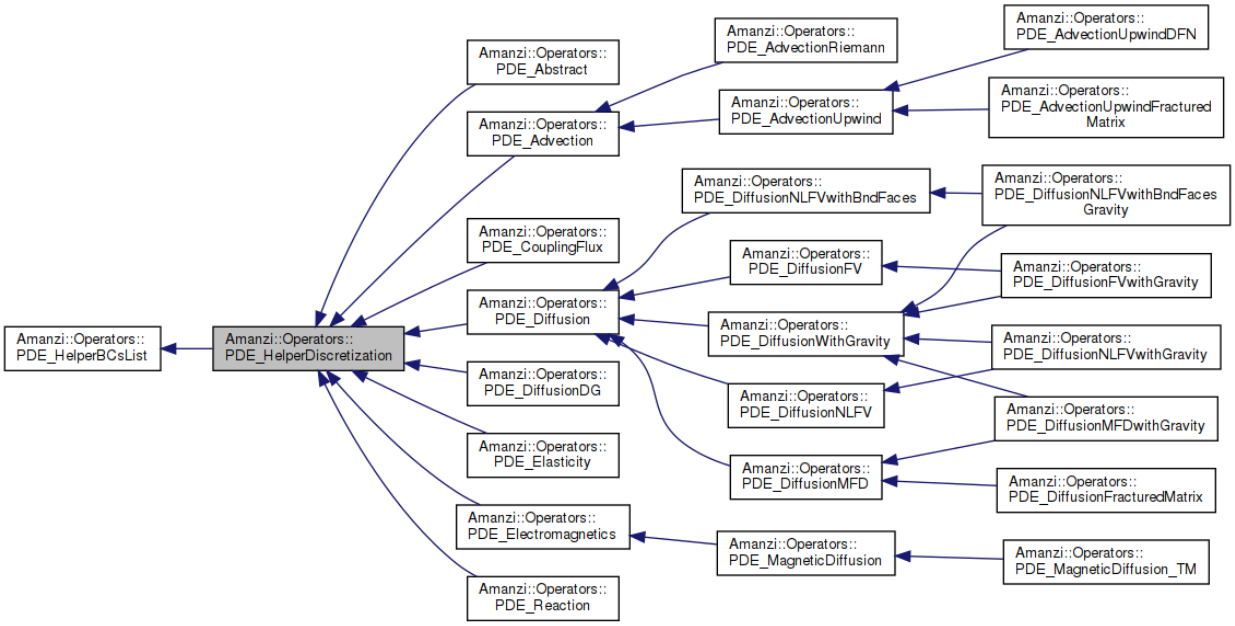

Figure 2: Partial dependency tree for library Operators.

3.3 Data structures

This library describes parallel vectors used in Amanzi. This includes classes `CompositeVector` and `TreeVector` described above. Additional functionality includes implementation of algorithms that close shortcomings of various Trilinos interfaces. For instance, from a user perspective, parallel communications should be the integral part of a parallel vector. This is done via Amanzi's wrapper classes `CompositeVector` and `TreeVector`.

Classes `GraphFE`, and `MatrixFE` provides capabilities for better assembly practices for Epetra-based implementations. They provide a plausibly scalable matrix for use in FE-like systems, where assembly must be done into rows of ghost entities as well as owned entities. These classes uses the "construct, insert, complete fill" paradigm of all Epetra graphs and CRS matrices. The only real difference is the use of *InserMyIndices()* and *SumIntoMyValues()* which may now take local indices from the ghosted map, not the true row map.

3.4 PKs

This library provides plausibly abstract classes for implementation of boundary conditions and source terms in the physical PKs, see Fig. **??**.

Multiple subdirectories contain implementation of various process kernels and MPC PKs. A PK factory is used for self-registering of PKs from the global input spec. To register an new PK, the developer must add a private, static member of type `RegisteredPKFactory` to the class declaration, and write a special `_reg.hh` file that instantiates the static registry:

```
// pk_implementation.hh
#include "PK.hh"
#include "PK_Factory.hh"
class DerivedPK : public Amanzi::PK {
 private:
  static Amanzi::RegisteredPKFactory<DerivedPK> factory_;
};

// pk_implementation_reg.hh
#include "pk_implementation.hh"
template<>
Amanzi::RegisteredPKFactory<DerivedPK> DerivedPK::factory_("pk unique id");
```

Each PK must implement the standard PK interface, as well as interfaces to either an explicit time integrator or an implicit solver. A few comments are below, see the code for the complete description of these interfaces:

- Function *Setup()* is used to create fields (e.g. pressure), field evaluators (e.g. water content), and variables (e.g. absolute permeability) and to register them with the state. This function should avoid any initialization work unless it is really needed for the state registration process.
- Function *Initialize()* is used to initialize primary state fields for the PK, and miscalleneous structures for discrete PDE operators, time integrators, boundary conditions, source terms, etc.

- Function *AdvanceStep()* is used to advance PK from time one timestep to another. For an implicit time discretization, it calls an implicit solver. For an explicit time discretization, it calls a Runge-Kutta solver or any other ODE integrator.
- Function *CommitStep()* is used to update any needed secondary variables at the next time slice under assumption that the timestep was sucessful.
- Function *FunctionalResidual()* is a part of the implict solver interface. It computes the non-linear functional at the given solution approximation.
- Function *ModifyPredictor()* is a part of the implict solver interface. It modifies optionally the extrapolated guess for the predictor that is going to be used as a starting value for the nonlinear solve in the time integrator.
- Function *FunctionalTimeDerivative()* is a part of the explicit time integration interface. It calculates the right-hand side of an ODE system at the given state vector.

3.4.1 Setup

The following example shows how to register the evaluator that computes the total water content. First, we define a key for the field to avoid usage of hard-coded names in the code. The key takes a name of a computational domain and the field name. Next, we populate names of the dependends fields: pressure, saturation, and porosity. Finally, we create a secondary variable field evaluator and link it to the field name.

```
Key wc_key = Keys::getKey(domain, "water_content");

S->RequireField(wc_key, wc_key)->SetMesh(mesh)->SetGhosted(true)
  ->SetComponent("cell", AmanziMesh::CELL, 1);

Teuchos::ParameterList elist;
elist.set<std::string>("pressure key", pressure_key_)
     .set<std::string>("saturation key", saturation_liquid_key_)
     .set<std::string>("porosity key", porosity_key_);

auto eval = Teuchos::rcp(new VWContentEvaluator(elist));
S->SetFieldEvaluator(water_content_key_, eval);
```

3.4.2 Boundary conditions

To help implementing boundary conditions, this library provides a few helper classes derived from the base class `PK_DomainFunction`. Each PK has typically its own boundary class derived (optionally) from the base class, e.g. `FlowBoundaryFunction`. This class parses the XML sublist *boundary conditions* and creates distributed arrays with boundary data. The key variable is typically named *bcs_* and can be found in a main PK class. Explicit Transport uses *bcs_* directly. Other PKs take *bcs_* and populate mesh-size arrays (see class `BCs`) in order to use operators' machinery for imposing boundary conditions.

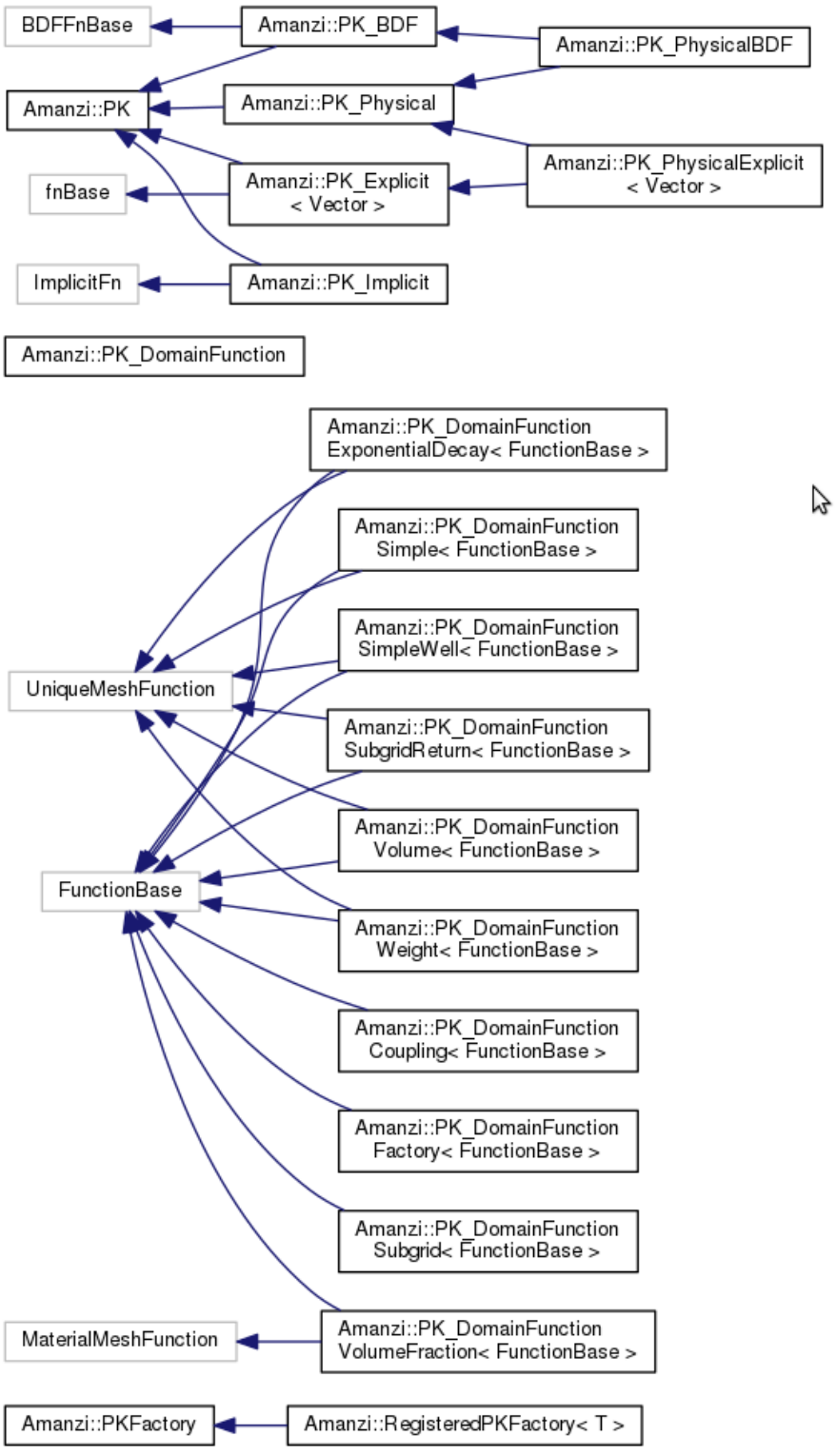

Figure 3: Partial dependency tree for library PKs.

Another approach to imposing simple boundary condition is useful for unit tests:

```
// get mesh maps with ghosts entities
const auto& fmap = mesh->face_map(true);
const auto& bmap = mesh->exterior_face_map(true);

// loop over boundary faces
for (int bf = 0; bf < bmap.NumMyElements(); ++bf) {
  // local boundary face id -> global face id -> local face id
  int f = fmap.LID(bmap.GID(bf));
  const Point& xf = mesh->face_centroid(f);
  ...
}
```

3.5 General purpose factory

In many cases, the developer may have multiple options that inherit a common (likely purely) virtual class. For instance, many implementations of the equations of state class will provide a basic method for $\rho(T, p)$, including both real "fits" to data, analytic expressions, and fake EOS classes for testing. We would like to be able to:

- choose the implementation at run time
- easily add new implementations

To do the first, we use a factory design pattern. Like most factories, an implementation must be "registered" with the factory. To do the second, this registration must NOT be done in the factory's source code itself.

This is made a little easier by the fact that nearly all of these things will be constructed using a single interface for the constructor, which (explicitly) takes a single argument – a variable of class `ParameterList` – and parses that list for its actual parameters. While it is usually a good idea to have a factory take the input list, do the parsing, and call the model's constructor with the parameters, that would require every model implementation to have its own factory. To simply things for scientists writing these models, we choose to do the parsing within the constructor/initialization.

The obvious exception to this is the model type parameter, which must get read by a factory and mapped to an implementation's constructor.

The general purpose factory is templated to take a single base class. Implementations of that base class then "register" themselves with the factory instance (which is stored statically since we cannot correctly manage the cleanup). This factory assumes all constructors for all implementations of all base classes take a single variable of class `ParameterList` as an argument. An EOS example:

```
// eos_factory.cc  (no .hh file necessary)
#include "eos.hh" // header for class EOS, a purely virtual base class
#include "factory.hh" // this file
template <> Factory<EOS>::map_type* Factory<EOS>::map_; // explicitly
                                                        // instantiate the
                                                        // static registry
// eos_implementation.hh
#include "eos.hh"
class DerivedEOS : public EOS {
  DerivedEOS(Teuchos::ParameterList& plist);

 private:
  static RegisteredFactory<EOS,DerivedEOS> factory_; // my factory
};

// pk_using_an_eos.cc
#include "eos.hh"

void init(...) {
  Factory<EOS> eos_factory;
  my_eos_ = eos_factory.CreateInstance("my_eos_type", eos_plist);
}
```

4 From first-time users of Amanzi

The purpose of this section is to describe first-hand experience in solving square systems using Amanzi. Square systems are those that can be written as

$$\text{Find } u \in V : \ \forall v \in V \quad a(u, v) = f(v). \tag{4.1}$$

The types of problems these systems include well-known differential systems like Poisson, advection-diffusion, magneto- and electro-statics. In summary, the process of solving these types of systems is as follows, begin by creating a mesh using the `MeshFactory`, then create a class for your PDE type as a derived class from `PDE_HelperDiscretization`, this will give access to a container for local matrices and a global operator that assembles these matrices as well as routines for applying Dirichlet-type boundary conditions. Thus, the next step is to populate the entries in the container for mass matrices, then assemble the global system and the right-hand side and apply a linear solver.

4.1 Defining your PDE class

The PDE class defined must be a derived class of PDE_HelperDiscretization, this immediately gives new class access to two important variables, a global operator and a container for local matrices and a series of useful routines to apply boundary conditions, assemble global systems, etc. This all comes with a caveat: you must define a member function called it UpdateMatrices() which is usually used to populate the local matrices, failure to do so will result in an abstract class with no possibility for instantiation. The header for a class to solve the Poisson equation in second order form will look like

```
class PDE_SecondOrderPoisson: public PDE_HelperDiscretization {
 public:
  PDE_SecondOrderPoisson(const Teuchos::RCP<const AmanziMesh::Mesh>& mesh);
  ~PDE_SecondOrderPoisson() {};

  // populate container of local matrices
  virtual
  void UpdateMatrices(const Teuchos::Ptr<const CompositeVector>& u,
                      const Teuchos::Ptr<const CompositeVector>& p);

  // postprocessing: calculate flux u from potential p
  virtual
  void UpdateFlux(const Teuchos::Ptr<const CompositeVector>& p,
                  const Teuchos::Ptr<CompositeVector>& u);

  // accessors
  Teuchos::RCP<CompositeVectorSpace> GetCVS() { return cvs_; }

 public:
  Teuchos::RCP<CompositeVectorSpace> cvs_;
};
```

Notice that in this class we have, additionally, defined a composite vector space as a class variable. Composite vector spaces are factories for composite vectors which is an enhanced EPetra Multi-Vector. This factory can help us create vectors for our trial or test spaces which in the case of a square system are the same or at least have the same set of degrees of freedom. In what follows we will explain how to define the global operator, we note that this is all done in the constructor of the PDE class.

4.1.1 Constructing the local stiffness matrix

We will approximate, for a cell P, the bilinear form a as

$$a_P(p,q) := \int_P \nabla p \cdot \nabla q \approx p^I \cdot M_P q^I \tag{4.2}$$

where the superscript I refers to the vector of degrees of freedom of p and q. The matrix M is given by

$$M_P = R(R^T N)^\dagger R^T + \lambda(I - N(N^T N)^{-1} N^T), \quad \lambda = \frac{2}{\#nodes}\text{tr}\left(R(R^T N)^\dagger R^T\right), \tag{4.3}$$

for

$$N = \begin{pmatrix} 1 & x_{v_1} - x_P & y_{v_1} - y_P \\ 1 & x_{v_2} - x_P & y_{v_2} - y_P \\ 1 & x_{v_3} - x_P & y_{v_3} - y_P \\ 1 & x_{v_4} - x_P & y_{v_4} - y_P \end{pmatrix} \quad R = \frac{1}{2}\begin{pmatrix} 0 & |e_4|n_x^{(4)} + |e_1|n_x^{(1)} & |e_4|n_y^{(4)} + |e_1|n_y^{(1)} \\ 0 & |e_1|n_x^{(1)} + |e_2|n_x^{(2)} & |e_1|n_y^{(1)} + |e_2|n_y^{(2)} \\ 0 & |e_2|n_x^{(2)} + |e_3|n_x^{(3)} & |e_2|n_y^{(2)} + |e_3|n_y^{(3)} \\ 0 & |e_3|n_x^{(3)} + |e_4|n_x^{(4)} & |e_3|n_y^{(3)} + |e_4|n_y^{(4)} \end{pmatrix} \tag{4.4}$$

where $\{(x_{v_i}, y_{v_i}) : 1 \le i \le 4\}$ is the set of vertices of the rectangle P, $\left\{\left(n_x^{(i)}, n_y^{(i)}\right) : 1 \le i \le 4\right\}$ makes a set of outward normal vectors to the edges of P. This derivation is an example of a mimetic method presented in chapter 4 of **?**.

4.1.2 Populating the local matrices and defining the global operator

The first step in populating the local matrices is to define a schema. There should be one schema for the test space and one for the trial space, in the case of square systems the same can be used for both. Schemas define the different aspects of a variable and its discretization. Schemas require two inputs: the base and and an item. The base describes what type of assembly is required for this variable the choices include cells, faces, edges and nodes all part of the AmanziMesh namespace. Selecting, for example, faces as the base will imply that the local matrices are associated with the faces of the mesh. Moreover, item defines the type of degrees of freedom that are used to discretize the variable in question. Items require three inputs: a part of the topology of the mesh like a node or an edge which defines where the degrees of freedom are placed, the type of quantity the degree of freedom, whether it is scalar of vector valued and the number of degrees of freedom of this type. For a classic finite element method to solve the Poisson equation the definition of its schema will look something like

```
Schema p_schema;
// the assembly should run over the cells thus its base are the cells
p_schema.SetBase(AmanziMesh::CELL);

// the pressure dofs are cell-based and scalars
p_schema.AddItem(AmanziMesh::NODE, WhetStone::DOF_Type::SCALAR, 1);
p_schema.Finalize(mesh);  // computes the starting position of the dof ids
```

Feeding the mesh to the schema, as shown in the last step, will create important variables used in the eventual assembly. Once the necessary schemas are defined the local operator can be initialized and populated in a fairly straight-forward way. It is a matter of feeding the schemas to the local operator and defining the necessary matrices. For example:

```
local_op_ = Teuchos::rcp(new Op_Cell_Schema(p_schema,p_schema,mesh));
// populate the local matrices
for (int c = 0; c < ncells_owned ; c++) {
  Mcell(0,0) =  3, Mcell(0,1) = -1, Mcell(0,2) = -1, Mcell(0,3) = -1;
  Mcell(1,0) = -1, Mcell(1,1) =  3, Mcell(1,2) = -1, Mcell(1,3) = -1;
  Mcell(2,0) = -1, Mcell(2,1) = -1, Mcell(2,2) =  3, Mcell(2,3) = -1;
  Mcell(3,0) = -1, Mcell(3,1) = -1, Mcell(3,2) = -1, Mcell(3,3) =  3;
  local_op_->matrices[c] = Mcell;
}
```

The above demostrates how one can take an existing matrix and update the local operator. In practice one must also build the required matrix. Initally, say to define the matrix in (**??**) one requires to find geometric features of the mesh like the coordinates of the nodes, the length of edges or vectors that are orthogonal to the boundary of a cell. All of these can be attained from the mesh class from the commands displayed below

```
AmanziMesh::Entity_ID_List nodeids, edgeids;
AmanziMesh::Entity_ID node0, node1, node2, node3;
AmanziGeometry::Point v0, v1, v2, v3;
AmanziGeometry::Point n0, n1, n2, n3, vP;
double l0, l1, l2, l3;

vP = mesh_->cell_centroid(c);
mesh->cell_get_edges(c, &edgeids);

mesh_->edge_get_nodes(edgeids[0], &node0, &node1);
mesh_->edge_get_nodes(edgeids[1], &node1, &node2);
mesh_->edge_get_nodes(edgeids[2], &node2, &node3);

mesh_->node_get_coordinates(node0, &v0);
mesh_->node_get_coordinates(node1, &v1);
mesh_->node_get_coordinates(node2, &v2);
mesh_->node_get_coordinates(node3, &v3);

l0 = mesh_->edge_length(edgeids[0]);
l1 = mesh_->edge_length(edgeids[1]);
l2 = mesh_->edge_length(edgeids[2]);
l3 = mesh_->edge_length(edgeids[3]);

n0 = mesh_->face_normal(edgeids[0], false, c);
n1 = mesh_->face_normal(edgeids[1], false, c);
```

```
n2 = mesh_->face_normal(edgeids[2], false, c);
n3 = mesh_->face_normal(edgeids[3], false, c);
```

Notice that to attain normal vectors we use the command `face_normal` this is because in amanzi faces and edges are the same entity in two dimensions. Next, we will construct the matrices N and R as defined in (**??**). They will be instances of the class DenseMatrix in the WhetStone namespace. This will give us access to important member functions that perform standard operations in linear algebra like inverting or multiplying matrices. Their construction is:

```
WhetStone::DenseMatrix N(4,3);
WhetStone::DenseMatrix R(4,3);

N(0,0) = 1, N(0,1) = v0[0] - vP[0], N(0,2) = v0[1] - vP[1];
N(1,0) = 1, N(1,1) = v1[0] - vP[0], N(1,2) = v1[1] - vP[1];
N(2,0) = 1, N(2,1) = v2[0] - vP[0], N(2,2) = v2[1] - vP[1];
N(3,0) = 1, N(3,1) = v3[0] - vP[0], N(3,2) = v3[1] - vP[1];

R(0,0) = 0, R(0,1) = l3*n3[0]+l0*n0[0], R(0,2) = l3*n3[1]+l0*n0[1];
R(1,0) = 0, R(1,1) = l0*n0[0]+l1*n1[0], R(1,2) = l0*n0[1]+l1*n1[1];
R(2,0) = 0, R(2,1) = l1*n1[0]+l2*n2[0], R(2,2) = l1*n1[1]+l2*n2[1];
R(3,0) = 0, R(3,1) = l2*n2[0]+l3*n3[0], R(3,2) = l2*n2[1]+l3*n3[1];
```

Finally, all that we have left in order to construct the local stiffness matrices is to perform the operators in (**??**). To do this we will define a series of temporary variables that store the variables in between.

```
WhetStone::DenseMatrix TR(3,4);
WhetStone::DenseMatrix Mcell(4,4), Scell(4,4);
WhetStone::DenseMatrix temp1(3,3), temp2(4,3);

// formula: R(R^TN)^t R^T+lambda*(Id-N(N^TN)^{-1}N^T)/2
// lambda = tr(R(R^TN)^t R^T)

// the first summand.
temp1.Multiply(R, N, true);
temp1.InverseMoorePenrose();
temp2.Multiply(R, temp1, false);
TR.Transpose(R);
Mcell.Multiply(temp2, TR, false);

// the second summand.
double lambda = Mcell.Trace();
temp1.Multiply(N, N, true);
temp1.InverseSPD();
temp2.Multiply(N, temp1, false);
TR.Transpose(N);
Scell.Multiply(temp2, TR, false);

Scell *= -lambda / 2;
for(int i = 0; i < 4; i++) Scell(i, i) += lambda / 2;
local_op_->matrices[c] = Mcell + Scell;
```

The nested for loops that are performed towards the end of the above code are intended to substract the identity while simultaneously multiplying by lambda.

To finalize we need to define the global operator which requires us to fist specify a composite vector space that is consistent with the schema that we defined in the local systems. Thankfully schema has a routine that does this for us. Thus, initializing the global operator can be done in four lines as follows:

```
cvs_ = Teuchos::rcp(new CompositeVectorSpace(
    cvsFromSchema(p_schema, mesh, false)));

// constructor for a global operator requires a parameter list
Teuchos::ParameterList plist = Teuchos::ParameterList();

// create a global operator for the mass matrix
global_op_ = Teuchos::rcp(new Operator_Schema(cvs_, plist, p_schema));

// assign the corresponding container of local matrices
std::string my_name = "Diffusion:■Poisson";
local_op_ = Teuchos::rcp(new Op_Cell_Node(my_name, mesh_));
global_op_->OpPushBack(local_op_);
```

4.2 Creating a mesh

The class `MeshFactory` gives the necessary tools to create a mesh. Mesh factory requires some inputs: the MPI communicator and the preferences which provide the specific capability that is used. A default communicator is already defined in Amanzi namespace and the preference is usually set to the MSTK framework.

The mesh factory object can create meshes in several ways depending on the dimensionality (2D or 3D) and the types of cells. A complete routine to build a simple quadrilateral mesh in 2D will look like this:

```
auto comm = Amanzi::getDefaultComm();
MeshFactory factory(comm);
factory.set_preference(Preference({Framework::MSTK}));
// generate a square mesh covering [-1,1] x [-1,1] with 9 cells.
Teuchos::RCP<const Mesh> mesh = factory.create(-1.0,-1.0,1.0,1.0, 3,3);
```

4.3 Adding boundary conditions

The class `PDE_HelperDiscretization` has some built-in features to impose boundary conditions but in order to access them we need to define the object of class `BCs` which takes as creation arguments the mesh, where the degrees of freedom will be places and the type of degree of freedom. Moreover, we must also populate the two class variables, `bc_model` which defines what type of boundary condition we want to prescribe and `bc_value` which precise value of such boundary condition. For example,

```
// the BCs are placed on the nodes and are scalars
Teuchos::RCP<BCs> bcv = Teuchos::rcp(new BCs(
    mesh, AmanziMesh::NODE, WhetStone::DOF_Type::SCALAR));
```

```
std::vector<int>& bcv_model = bcv->bc_model();
std::vector<double>& bcv_value = bcv->bc_value();

Point xv(2);  // a point with two entries
// nnode_wghost is the number of nodes in the mesh including ghosts
for (int v = 0; v < nnodes_wghost; ++v) {
  mesh->node_get_coordinates(v, &xv);
  // This will identify which points lie in the boundary
  if (fabs(xv[0] + 1.0) < 1e-6 || fabs(xv[0] - 1.0) < 1e-6 ||
      fabs(xv[1] + 1.0) < 1e-6 || fabs(xv[1] - 1.0) < 1e-6) {
    bcv_model[v] = Operators::OPERATOR_BC_DIRICHLET;
    bcv_value[v] = 1.0;
  }
}
```

4.4 Assembly and imposing the boundary conditions

Amanzi imposes boundary conditions by placing 1 in the correct place in the global matrix and adding the value to the right hand side yielding the correct values in the final solution after the linear solve is performed. Thus, before imposing these conditions we must feed the right hand side to the object created by our PDE class. This is fairly simple since we already have created a composite vector space for the functions defined in our schema we can make use of this factory to initialize the right hand side and manually populate its entries as follows

```
CompositeVector source(cvs);
Epetra_MultiVector& src = *source.ViewComponent("node");
for (int v = 0; v < nnodes_owned; v++) {
  mesh->node_get_coordinates(v, &xv);
  src[0][v] = 1.0;
}
```

Next we instantiate our PDE class, feed the right hand side, apply the boundary conditions and assemble the system

```
auto op_poisson = Teuchos::rcp(new PDE_SecondOrderPoisson(mesh));
op_poisson->SetBCs(bcv, bcv);
op_poisson->UpdateMatrices(Teuchos::null, Teuchos::null);
op_poisson->ApplyBCs(true, true, true);

// global assembly
Teuchos::RCP<Operator> global_op = op_poisson->global_operator();
global_op->UpdateRHS(source, true);
global_op->SymbolicAssembleMatrix();
global_op->AssembleMatrix();
```

4.5 The linear solve

In order to apply a linear solver we must initialize a vector for the solution and initialize the preconditioner. The linear solve is templated to fit the different types of scenarios where linear solves are necessary.

```
const CompositeVectorSpace& cvs = *op_poisson->GetCVS();
CompositeVector solution(cvs);
solution.PutScalar(0.0);  // solution initialized with the value zero

// This file contains the specifications for the preconditioner
std::string xmlFileName = "test/operator_SecondOrderPoisson.xml";
Teuchos::ParameterXMLFileReader xmlreader(xmlFileName);
Teuchos::ParameterList plist = xmlreader.getParameters();
auto slist = plist.sublist("preconditioners").sublist("Hypre AMG");
global_op->InitializePreconditioner(slist);
global_op->UpdatePreconditioner();

auto olist = plist.sublist("solvers").sublist("PCG").sublist("pcg parameters");
AmanziSolvers::LinearOperatorPCG<Operator, CompositeVector,
    CompositeVectorSpace> pcg(global_op, global_op);

pcg.Init(olist);
CompositeVector& rhs = *global_op->rhs();
int ierr = pcg.ApplyInverse(rhs, solution);
```

The xml file used above contains the following information

```
<ParameterList name="solvers">
 <ParameterList name="PCG">
  <Parameter name="iterative method" type="string" value="pcg"/>
  <ParameterList name="pcg parameters">
   <Parameter name="maximum number of iterations" type="int" value="20"/>
   <Parameter name="error tolerance" type="double" value="1e-12"/>
  </ParameterList>
 </ParameterList>
</ParameterList>

<ParameterList name="preconditioners">
 <ParameterList name="Hypre AMG">
  <Parameter name="discretization method" type="string" value="generic mfd"/>
  <Parameter name="preconditioner type" type="string" value="boomer amg"/>
  <ParameterList name="boomer amg parameters">
   <Parameter name="cycle applications" type="int" value="2"/>
   <Parameter name="smoother sweeps" type="int" value="3"/>
   <Parameter name="strong threshold" type="double" value="0.5"/>
   <Parameter name="tolerance" type="double" value="0.0"/>
   <Parameter name="relaxation type" type="int" value="6"/>
   <Parameter name="verbosity" type="int" value="0"/>
  </ParameterList>
 </ParameterList>
</ParameterList>
```

5 Code development

5.1 Development cycle

Development of a new capability consists of several steps that are summarized below. Some steps can be skipped during a casual work cycle of code support, bug fixes, and minor improvements.

- Create a new github development branch.
- Create github ticket or multiple tickets that summarize and stage the development process.
- Implement numerical algorithms and add them to an Amanzi library.
- Write unit tests for the new code.
- Integrate new functionality into other algorithms.
- Write integrated unit tests as needed.
- If implemented algorithms take control parameters from an XML file, document these parameters.
- Test new capability and add a benchmark or verification test to the user guide.
- Create a pull request to inform team members about the new capability and to collect miscallenous feedback.
- Merge the development branch into the master branch.

5.2 Smart pointers

We suggest the following guidelines on smart pointers and argument passing.

```
unique_ptr<X> factory();       // creates an X
void sink(unique_ptr<X>&);     // consumes X, the caller cannot keep X
                               // or use it after call
void reseat(unique_ptr<X>&);   // change which X the caller points to

Teuchos::RCP<X> factory();  // creates an X
Teuchos::RCP<X> share();    // the caller will or might keep X
void share(const Teuchos::RCP<X>&);  // the callee will or might keep X
void reseat(Teuchos::RCP<X>&);       // change which X the caller points to

void optional(X*);  // if X is an optional argument, i.e. pointer can be null

void if_primitive(X);  // if none of the above apply, and X is a primitive
void if_not_primitive(X&);  // if none of the above apply, and X is not a primitive
X = primitive_return();        // if primitive
X& = non_primitive_return();   // if there is no ownership transfer implied,
                               // the caller just looks at (const) or modifies
                               // (nonconst) X but does not keep it.
```

Note that X can be replaced by const X with no loss of generality. Note that the key part of this is that passing by RCP should mean something about ownership, i.e. that the callee might keep X! This also suggests to NEVER use Teuchos::Ptr, instead using X*. Using Ptr requires that the object WAS stored in an RCP, and means an object stored in a unique_ptr must jump through hoops to be put in a non-owning Ptr before it can be passed.

6 Testing

Testing is a cornerstone of modern software development. In the form of Test-Driven Development, it is useful for providing feedback in the design process. In other forms, it is essential for preventing the project from descending into chaos, and controlling the cost of software maintenance. In this section we describe the various forms of testing used to certify that Amanzi works properly, in order of increasing scope.

6.1 Unit Testing

Each individual software component should have a defined set of assumptions under which it operates, and a set of behaviors and corresponding certifications on what it produces. These assumptions and behaviors are ideally articulated in the documentation of the component, but they should also be tested independently as part of the implementation process. A test of an individual component's assumptions and behaviors is called a *unit test*. A unit test provides a set of PASS/FAIL tests for each function, method, and attribute in a software component.

Some Amanzi's tests are integrated tests that fill a huge gap between short unit tests and long benchmark tests. At the moment they are also called unit tests.

6.2 Verification and Benchmark Testing

The various algorithms we use in Amanzi have to be tested on the basic subsurface problems that are relevant to our charter, and compared against other codes to weigh the costs and benefits of our choices against existing approaches.

A *verification test* consists of a simulation run with a given input describing a problem that has a known solution, a characterization of the quality of the solution, and a PASS or FAIL result based on the quality of that solution measured against some threshold.

A *benchmark test* is a simulation run with a given input whose output is compared to the output of one or more other codes. All codes must have inputs that describe the same "benchmark problem." The differences between the codes can be evaluated visually and/or with numerical metrics. Numerical metrics allow benchmark tests to have PASS/FAIL results, whereas a visual inspection test requires an expert for evaluation, so the former are preferred where practical.

6.3 Regression Testing

A *regression test* is a simulation-based PASS/FAIL test similar to a verification test, and is typically part of a large suite of tests that are run automatically and periodically to ensure that bugs and errors have not been introduced into Amanzi during code development. We provide a couple of tools for constructing PASS/FAIL tests that can be used to monitor bugs and regressions. In particular, we support two types of regression tests: *smoke tests* and *comparison tests*.

6.3.1 Smoke tests

A smoke test simply runs an Amanzi simulation with a given input, PASSes if the simulation runs to completion, and FAILs otherwise. A smoke test can be created (and added to Amanzi's regression test suite) by calling the following CMake command inside of a CMakeLists.txt file in a testing directory:

```
ADD_AMANZI_SMOKE_TEST(<test_name>
                      INPUT file.xml
                      [FILES file1;file2;...;fileN]
                      [PARALLEL]
                      [NPROCS procs1 ... ]
                      [MPI_EXEC_ARGS arg1 ... ])
```

Arguments:

- `test_name`: the name given to the comparison test
- `INPUT` (required): This (required) keyword defines an Amanzi XML input file that will be run.
- `FILES` (optional): A list of any additional files that the test needs in order to run in its directory/environment. These files will be copied from the source directory to the run directory.
- `PARALLEL` (optional): The presence of this keyword signifies that this is a parallel job. This is also implied by an NPROCS value ¿ 1
- `NPROCS` (optional): This keyword starts a list of the number of processors to run the test on, and defaults to 1.
- `MPI_EXEC_ARGS` (optional): This keyword denotes extra arguments to give to MPI. It is ignored for serial tests.

6.3.2 Comparison tests

A comparison test runs an Amanzi simulation with a given input, and then compares a field or an observation from that simulation to that in the specified reference file, PASSing if the L2 norm of the difference in the simulation and reference values falls below the given tolerance. One can add a comparison test to the Amanzi regression test suite by calling the following CMake command inside of a CMakeLists.txt file within a testing directory:

```
ADD_AMANZI_COMPARISON_TEST(<test_name>
                           INPUT file.xml
                           REFERENCE reference
                           [FILES file1;file2;...;fileN]
                           ABSOLUTE_TOLERANCE tolerance
                           RELATIVE_TOLERANCE tolerance
                           [FIELD field_name]
                           [OBSERVATION observation_name]
                           [PARALLEL]
                           [NPROCS procs1 ... ]
                           [MPI_EXEC_ARGS arg1 ... ])
```

Arguments:

- `test_name`: the name given to the comparison test
- `INPUT` (required): This (required) keyword defines an Amanzi XML input file that will be run.
- `REFERENCE` The name of the file containing reference data to which the simulation output will be compared.
- `TOLERANCE` (required): This specifies the maximum L2 error norm that can be measured for a successful testing outcome.
- `FILES` (optional): A list of any additional files that the test needs in order to run in its directory/environment. These files will be copied from the source directory to the run directory.
- `FIELD` (required if OBSERVATION not given): The name of the field in Amanzi that will be compared to its reference value for this test.
- `OBSERVATION` (required if FIELD not given): The name of the observation in the Amanzi input that will be compared to its reference value for this test.
- `PARALLEL` (optional): The presence of this keyword signifies that this is a parallel job. This is also implied by an NPROCS value ¿ 1
- `NPROCS` (optional): This keyword starts a list of the number of processors to run the test on, and defaults to 1.
- `MPI_EXEC_ARGS` (optional): This keyword denotes extra arguments to give to MPI. It is ignored for serial tests.

7 Documentation

7.1 User Guide

The description of each test in the user guide uses the structured test format and consists of a few sections.

1. Introduction describes purpose of the test.
2. Problem Specification describes the physical and mathematical model with the level of details sufficient to reproduce the results.
3. Results and Comparison summarizes numerical results in a form of plots and tables (e.g., drawdown curves). Comparison with analytic data, data produced by other codes, or data published elsewhere is strongly encouraged.
4. References.
5. About collects technical information for developers that include location of the files, names of the developers, names of critical files (XML input, Exodus mesh, analytic data, etc), names of output files.
6. Status describes status of the test and required future work.

To build the user guide a few python modules have to be installed inluding ipython and sphixcontrib extensions such as bibtex and tikz. Note that on OSX, the tikz extension is usually not available via MacPorts and has to be installed using pip.

7.2 Native Spec

This is a continuously evolving specification format used by the code developers. Its main purpose is to develop and test new capabilities without disruption of end-users. The documentation is in the form of a structured text, see `doc/input_spec/AmanziNativeSpecV8.rst`.