OAK RIDGE
NATIONAL LABORATORY
MANAGED BY UT-BATTELLE
FOR THE DEPARTMENT OF ENERGY

ORNL/TM-2010/34

Software Design Document for the AMP Nuclear Fuel Performance Code

February 2010

Prepared by
Bobby Philip, Kevin T. Clarno, and William K. Cochran
Oak Ridge National Laboratory

UT-BATTELLE
ORNL-27 (4-00)

ORNL/TM-2010/34

Nuclear Science and Technology Division

SOFTWARE DESIGN DOCUMENT FOR THE AMP NUCLEAR FUEL PERFORMANCE CODE

Bobby Philip, Kevin T. Clarno, and William K. Cochran

February 2010

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

Page

LIST OF FIGURES

SOFTWARE DESIGN DOCUMENT FOR THE AMP NUCLEAR FUEL PERFORMANCE CODE

Bobby Philip, Kevin T. Clarno, and William K. Cochran

Revision 0

1 INTRODUCTION

The purpose of this document is to describe the design of the AMP nuclear fuel performance code[1]. It provides an overview of the decomposition into separable components, an overview of what those components will do, and the strategic basis for the design. The primary components of a computational physics code include a user interface, physics packages, material properties, mathematics solvers, and computational infrastructure. Some capability from established off-the-shelf (OTS) packages will be leveraged in the development of AMP, but the primary physics components will be entirely new. The material properties required by these physics operators include many highly non-linear properties, which will be replicated from FRAPCON[2] and LIFE where applicable, as well as some computationally-intensive operations, such as gap conductance, which depends upon the plenum pressure. Because there is extensive capability in off-the-shelf leadership class computational solvers, AMP will leverage the Trilinos[3], PETSc[4], and SUNDIALS[5] packages. The computational infrastructure includes a build system, mesh database, and other building blocks of a computational physics package. The user interface will be developed through a collaborative effort with the Nuclear Energy Advanced Modeling and Simulation (NEAMS) Capability Transfer program element as much as possible and will be discussed in detail in a future document.

1.1 PURPOSE AND FEATURES

To meet the immediate need of the fuel performance community and provide a tool for clarifying the requirements of the 2015 NEAMS engineering-scale fuel performance code, the AMP software will be completed in August 2010 with a user-focused training session and final delivery to the Radiation Safety Information Computing Center (RSICC) to follow in September. AMP will be a new code developed through a close collaboration of the Oak Ridge, Idaho, Los Alamos, and Argonne national laboratories, and major leveraging of existing OTS codes, to provide an interim capability to (1) deliver a useful, new capability to the user community; (2) enhance our understanding of the software and user requirements; (3) demonstrate an understanding of the coupled physics simulation process with best-of-class software; and (4) gain experience developing software as a multi-institutional team with a single set of coding conventions, standards, and tools.

This effort will solidify the joint understanding of the physics that must be modeled, how they physics interrelate, and how the developers can streamline the process toward a true collaborative, multi-institutional software development environment. Much of the required capability exists in OTS codes that were enhanced and modularized in Fiscal Year (FY) 2009, but the multidimensional core of the fuel performance code (thermo-mechanical chemistry) will be developed from scratch in FY 2010 by leveraging the experience gained in FY 2009. This new code will tightly couple these core physics and leverage zero- and low-dimensional approximations for much of the associated physics. The AMP project will provide

1. a tightly coupled, three-dimensional thermochemical-mechanical solver that accounts for contact;
2. approximate models for the material properties, depletion, heat generation, plenum pressure, and convective heat transfer, which are similar to those found in FRAPCON and SCALE[6];
3. a simple user interface to set up, simulate, and understand the performance of Light-Water Reactor (LWR) oxides; and
4. a compiled version that executes in parallel on a cluster at Oak Ridge National Laboratory (ORNL).

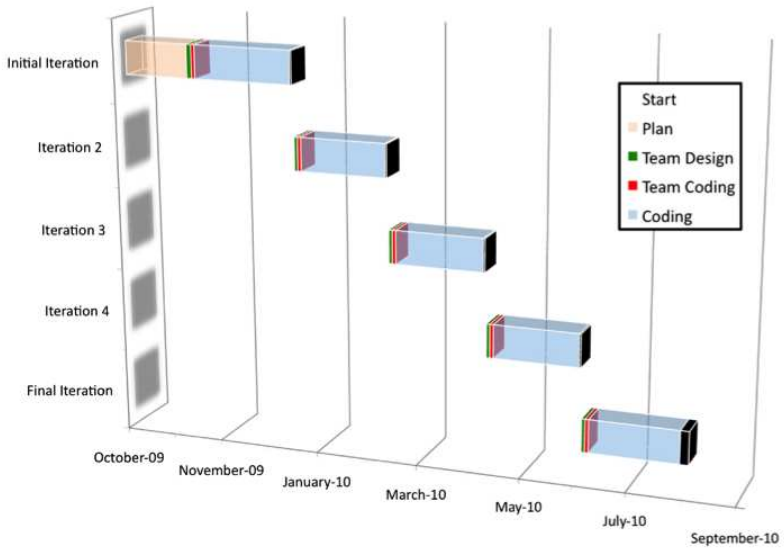

Figure 1: Iterative development plan for AMP.

With a final release in August 2010, AMP will be developed in five 2-month iterations (Fig. 1) that will each include an opportunity for a few expert users and associated NEAMS efforts (Enabling Computational Technologies, Capability Transfer, and Verification, Validation and Uncertainty Quantification) to review what has been accomplished and provide feedback.

AMP will be rapidly designed and developed without a focus on longevity or software quality engineering (especially within the OTS components). Because of the fundamental limitations that are present when working with a collection of OTS codes, which were not designed with a consistent approach toward quality, modularity, or coupling, the initial release of the 2015 code in August 2012 will replace AMP with software designed and built to simplify maintenance, enhance inherent quality, add additional physics, and incorporate lower-length-scale models. Therefore, there will be no additional releases of AMP after August 2010. The specific deliverables associated with this project are included in Appendix A.

1.2 SUMMARY

This document provides a detailed design of the AMP code and basis for the decisions that led to this particular design. This section provides an overview of the purpose and overarching requirements of AMP and summarizes the L2 and L3 deliverables for the project. Section 2 provides a brief overview of nuclear fuel simulation, the physics approximations that AMP will leverage, and the external physics packages that will be leveraged. Section 3 includes a discussion of the development process and resulting design considerations, as well as the software infrastructure and software development policies that will guide the development of AMP. The design considerations, discussed in Section 3, lead to a general system architecture, described in Section 4. The details of the software components are described in Section 5.

2 FUEL SIMULATION OVERVIEW

This section provides a brief overview of the physics that will be modeled in AMP, the physics that will be neglected, and the anticipated future use of the code. These are the highest-level requirements of the code that provide the basis for design considerations, which are discussed in detail in Section 3. For a more detailed description of the physics, see Olander[7]. A more detailed description of the software requirements will be provided in a supplementary document.

The AMP nuclear fuel performance code will compute the three-dimensional thermal, mechanical, and stoichiometric state of traditional nuclear fuel (UO_2) in an LWR. Nuclear fuel in an LWR is composed

of hundreds of individual ceramic UO_2 pellets stacked inside a protective metal cladding tube, which is surrounded by flowing water to cool the system. The nuclear heat produced in the materials is transported through the fuel, across a gap, through the cladding, and removed by the coolant. The materials respond mechanically to the thermal stresses and the stoichiometric state changes due to thermal gradients. Because the temperature distribution is dependent upon both the mechanical and stoichiometric state, the physics are always nonlinearly coupled. Nuclear irradiation changes the isotopic and elemental composition of the materials, which changes the material properties of the materials and, along with thermal gradients, imposes slowly varying stresses (densification, creep, and swelling).

Modeling these physical processes requires that AMP incorporate:

- a nuclear source term (simplified through preprocessing),
- elastic-plastic mechanics within solid bodies and mechanical stresses between solid bodies,
- thermal and oxygen diffusion within solid bodies and heat transfer between solid bodies,
- coolant flow and heat transfer (approximated as one-dimensional),
- nonlinear material properties, and
- mathematical solvers, all built upon
- a general, computational backplane.

However, there are several significant physics that will not be modeled, including, but not limited to, chemistry, mechanical fracture, multidimensional flow and neutronics, and grain-level physics. A primary purpose of AMP is to be used as an exploratory tool to understand the software requirements associated with incorporating these physics in the planning for a future, predictive nuclear fuel performance code. Therefore, AMP is designed to enable the incremental incorporation of additional physics for rapid prototyping. Each physics that may be modeled will have a different degree of coupling with the other physics and require resolution on different time scales.

Because AMP is being rapidly designed, developed, and delivered, it will leverage established OTS software where possible. The material properties will leverage the functional equations from the FRAPCON code, which has a strong legacy in modeling UO_2 fuel in a LWR. The mathematical solver will be built upon the Trilinos package, which has a large set of parallel solvers and preconditioners for linear and nonlinear systems of equations. The isotopic depletion will be modeled with the ORIGEN-S code, from the SCALE nuclear analysis code suite, which provides the most extensive data set available. The mesh database and finite-element library will leverage the LibMesh software[8], which has an extensive user base for computational physics simulation.

However, risk mitigation requires that AMP not be tightly bound to any given package, in the event that a package proves insufficient. Therefore, AMP is designed to provide a modular coupling to these external packages that will minimize the cost of exchanging a given package for something different.

3 DEVELOPMENT PROCESS AND DESIGN CONSIDERATIONS

AMP is being rapidly developed by multiple researchers at four national laboratories. To enable efficient development, we will rely on modern revision control software and testing to discover and correct bugs and ensure interoperability between external components as soon as possible. The coding standards will be minimally defined and loosely enforced because there are insufficient resources and time to provide adequate review of all coding. However, through the development of AMP, the collective set of researchers will have learned to work together and developed a common nomenclature in a collaborative software development experience that will enable us to define strict coding standards and practices for the development of a future, predictive fuel performance code.

3.1 Design Considerations

AMP is written to:

- provide users with a consistent, simple, and extensible interface for solving multi-physics problems
- provide developers with the ability to leverage multiple existing software frameworks through a consistent interface
- allow for loose as well as tightly coupled physics components
- allow incremental approaches to solving multi-physics problems
- allow rapid prototyping in parallel over multiple potentially interacting mesh data structures
- allow for time dependent, time independent, and differential algebraic equation (DAE) systems

Many software frameworks are built *on top* of existing frameworks to leverage their capabilities. This, over time can lead to over dependence on a particular software framework, which may or may not continue to be supported by its own developers. AMP takes a slightly different approach, being built to live *in between* existing software frameworks. This approach allows AMP to leverage existing frameworks while at the same time avoid over dependence on a particular package or capability.

3.2 INFRASTRUCTURE

The software and documentation will be maintained, with revision control, on a GFORGE site at ORNL[9]. This site is accessible to all developers on the team who have a license to the source code of the SCALE code system (for export control protection) and ORNL cyber access. Documents, presentations, and software development information regarding AMP will be collected and maintained on the GFORGE site. For more information on GFORGE, see the GFORGE website[10]. The AMP software will be maintained under revision control on the GFORGE site through the use of the Subversion software[11]. In addition, documentation that is developed in a nonproprietary format (such as HTML and LaTeX, as opposed to .doc or .ppt), will be maintained along with the software. An excellent resource for Subversion is the redbean website[12].

AMP will leverage the Nemesis build system, from the Denovo radiation transport code[13] in the SCALE nuclear analysis code system. The Nemesis build system is a portable build system and development environment configuration that leverages the Autoconf software[14] that creates a configuration script for a package from a template file which lists the operating system features that the package can use. Nemesis contains the configuration utilities for the external packages that will, or may, be used by AMP, including Trilinos, PETSc, SUNDIALS, and LibMesh. Nemesis contains a testing harness that simplifies the development of unit and regression testing.

3.3 POLICIES

There will be no rigorous review of software to ensure that it conforms to a given nomenclature, format, or quality. However, we will conform to traditional software development etiquette of "if you write the code, you own it, and are responsible for it", which provides sufficient motivation for each developer to rigorously test and verify the proper operation and integration of the code. This and other developer information will be maintained in the "docs" section of the GFORGE site. The repository is decomposed into packages and each package will have a test directory. The Nemesis build system provides an infrastructure to incorporate a unit test for every class and/or function in every package of AMP.

1. It is the responsibility of developers to ensure the software they commit to the repository compiles, links, and is bug-free.
2. Every class committed to the repository should have at least one associated unit test. Examples of unit tests can be found in every package, including those distributed with Nemesis.

3. Before committing code the "trunk" of the repository, every developer should perform a "make check" of the entire AMP to ensure that all unit tests compile and run.

4. After committing to the "trunk" of the repository, every developer should perform a clean checkout, configure, compile, and test of the code.

5. If a developer plans to make a major change to the repository, a "branch" of the main development "trunk" should be created.

6. If a "branch" to the repository is made, it is the developer working on the branches responsibility to merge changes from the trunk into the branch to stay in sync.

7. When a "branch" is going to be merged into the trunk, the final changes should be made at a time and in a manner so as to minimally impact other developers (late at night, weekends, or early mornings).

8. If a developer finds an apparent bug in a section of coding, the GFORGE site provides a tracker to report the error

9. The individual who discovers a bug should provide a small test code (unit test) that demonstrates the error in a clear fashion.

10. Fixing a bug ultimately lies with the person who added the code to the repository; however, anyone is welcome to fix an error.

There will be a nightly checkout, configure, compile, and regression test of AMP on at least two platforms. An e-mail will be distributed to each active developer to ensure that the developers are aware of the current state of the code and can address bugs as they arise. The regression test will be composed of the entire suite of unit tests.

4 SYSTEM ARCHITECTURE

The design considerations for AMP resulted in several software components. In particular, components exist for:

- Mesh and geometry
- Discretization
- Vectors and matrices
- Operators
- Solvers
- Time integrators

Each component is designed to provide a uniform consistent interface which interacts with other components, and developers of other components are only exposed to these interfaces. This is despite the fact that AMP is designed to sit *in between* existing software frameworks to leverage their strengths and investments without overdependence. The complexities of interfacing different software frameworks are kept behind the standard interfaces that AMP provides (Fig. 2).

- **Mesh and geometry:** The mesh and geometry interface (AMPMesh) allows AMP to potentially interact with multiple mesh or geometry packages. AMPMesh already allows us to interface with the LibMesh package and may be used to interface with the STKMesh package which is a part of Trilinos.
- **Discretization:** Due to the close coupling between mesh and discretization, AMPMesh (through LibMesh, currently handles the discretization also. This is may become be a separate interface, if the need arises for to interface different discretization packages with mesh packages.

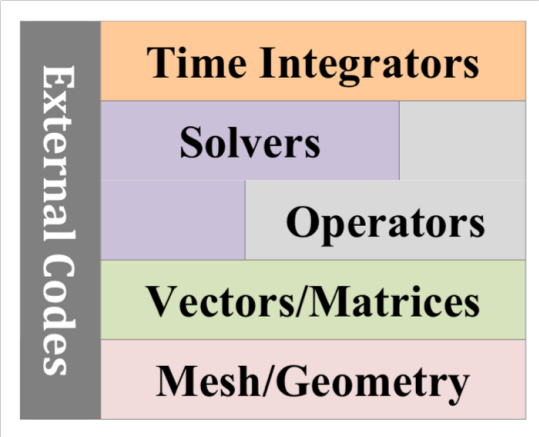

Figure 2: Structure of components in AMP.

- **Vectors and matrices:** AMP provides standard AMP::Vector and AMP::Matrix classes which serve two purposes. Firstly, they provide users with a standard interface to perform vector and matrix operators. At the same time, the classes hide the details of the interfacing various software packages that have their own definition of vectors and matrices. For example, Trilinos and PETSc both provide matrix operations and Trilinos, PETSc, and SUNDIALS provide and/or use vector operations. AMP Vector and Matrix act as the interfaces to these packages through the Vector and Matrix classes. The complexities associated with enabling these packages to interact is hidden from the user and the packages.

- **Operators:** Operators are the core of the AMP design and where all of the physics is contained. Operators encapsulate the details of the mapping operation $\mathcal{L} : X \rightarrow Y$ where X and Y are appropriately defined spaces. Operators may represent discretized PDE operators, boundary operators, an operation to extract material properties from material databases or tables, linear or nonlinear algebraic operations, or compositions of the above. The ability to compose operators and to extract information from compositions is intended to facilitate the incremental construction of multi-physics and/or multidomain simulations as well as rapid prototyping and experimentation to understand couplings in multi-physics simulations.

- **Solvers:** Solvers in AMP refer to the nonlinear and linear solvers that represent the action of an approximate inverse map of a given operator if that inverse operation has some well defined meaning. In this sense solvers can also be considered as operators. Whether solvers should be implemented as (approximate) inverse operators is a design choice that might need to be revisited. Currently, an inverse operator can be easily constructed by wrapping a solver in an inverse operator class. The solver interface allows the user to utilize a standard interface to solvers from Trilinos and PETSc (currently interfaces exist), native AMP solvers (these exist), and potentially other packages in future. Again, the design emphasis has been to provide a standard interface to hide the complexity of particular software packages from a user and to avoid overdependence on a particular software package.

- **Time integrators:** AMP time integrators provide a uniform interface to solving time-dependent systems which can include Differential Algebraic Equations (DAEs). This is necessary within the context of our target application because of coupling between time dependent thermal and quasi-static mechanical systems being simulated. The design allows for explicit, semi-implicit, and fully implicit simulations of coupled multi-physics problems. In the case of semi-implicit and fully implicit calculations, the solver interfaces in AMP are used, and in all cases, the operator interfaces are used to allow composable multi-physics simulations allowing users to experiment with coupling different physics together. The time integrator interface is used to provide an interface to the SUNDIALS suite

of time integrators and can be used in future to interface to the Rhythmos package of Trilinos as it matures.

The specific I/O entities for developing a fuel simulation input have not yet been determined and will be included in a future document.

5 DETAILED SYSTEM DESIGN

This section is intended to be a higher-level supplement to the highly-detailed documentation embedded within the source code, which are automatically generated (and included for reference in Appendix B) as HTML files by the Nemesis build system through the use of the Doxygen software.

5.1 BACKPLANE

5.1.1 Variables

Since the intent of AMP is to perform multi-physics simulation, AMP is designed to ease coupling of disparate physics. These physics are often described individually as operators, such as in the residual equations, $D_1 x = 0$ and $D_2 z = 0$, where $D_1 : X \to Y$ and $D_2 : Z \to W$ are operators. These operators are then combined through a coupling mechanism, C, to create a "global system" to be solved: $C(D_1, D_2) : X \times Z \to Y \times W$. This, in turn, is another operator.

These operators are implemented using a discretization process, such as finite differences or energy minimizing variational formulations. So, D_1, a continuous operator, would give rise to $\hat{D}_1$, a discrete operator. A multi-physics simulation may require applying any or all of the discrete operators on a variety of variables discretized on a variety of meshes. To facilitate this, AMP implements the concept of a "variable." A variable is a description of how an operator expects its input or output to be discretized. For instance, if D_1 is an operator arising from mechanics in three dimensions, then X may be R^3, the set of 3-vector valued functions, and any particular $x \in X$ might be computed at the nodes of a mesh.

A variable is also a context. Again, if D_1 is the mechanics operator mentioned above, then $X = Y = R^3$. Applying D_1 to a variable gives a variable in the same space: $D_1 x_1 = x_2$. The variables, x_1 and x_2, are said to have different contexts, often represented mathematically as different subscripts or different symbols.

To speed implementation of coupled physics, two or more variables can be composed into a single variable. If C is the coupling mechanism mentioned above, D_1 is a mechanics operator, and D_2 is a thermal operator ($Z = W = R$), then $C(D_1, D_2)u = 0$, a thermo-mechanical operator, can be applied to any $u \in X \times Z = R^3 \times R$, a composition of displacement and temperature.

Variables also provide a mechanism to order memory access. In the thermo-mechanical example above, the data in the variable, u, can be organized in several ways. For instance, for vector displacement at node, n, $x^n = \{x_1^n, x_2^n, x_3^n\}$ and scalar temperature T^n at node n, the data in memory could be organized as

$$u = \left\{ \begin{array}{c} x \\ T \end{array} \right\} = \{x_1^1 \; x_2^1 \; x_3^1 \; x_1^2 \; x_2^2 \; x_3^2 \; \ldots \; T^1 \; T^2 \; \ldots\}^T. \tag{1}$$

Alternatively, u can be completely interleaved, $u = \{x_1^1 \; x_2^1 \; x_3^1 \; T^1 \; x_1^2 \; x_2^2 \; x_3^2 \; T^2 \; \ldots\}^T$. Or, if the mechanics are highly anisotropic and the operator reflects this, then the interleaving could be different:

$$u = \left\{ \begin{array}{c} \left\{ \begin{array}{c} x_1 \\ x_2 \end{array} \right\} \\ x_3 \\ T \end{array} \right\} = \{x_1^1 \; x_2^1 \; x_1^2 \; x_2^2 \; \ldots \; x_3^1 \; x_3^2 \; \ldots \; T^1 \; T^2 \; \ldots\}^T.$$

Any of these organizations is constructed by composing variables in the appropriate fashion. When two variables x and T are composed, then the storage designated by x and T are used in the composition as in equation (1). In this way, simulation designers can have very precise control of memory access patterns using a concept with which they are already familiar.

As part of the implementation, vector values may be stored at nodes or on elements. For instance, a displacement computed or stored on a node would be a Nodal3VectorVariable. In this sense, the term

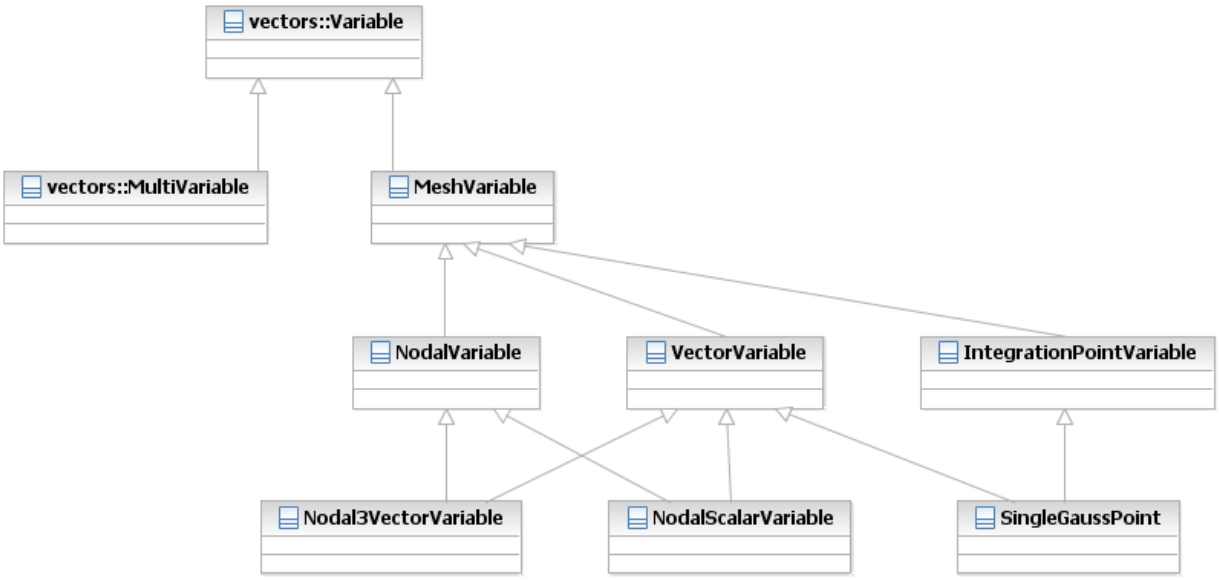

Figure 3: Inheritance model for variable classes.

"vector" refers to the range of a vector valued function or image of a map. In Fig. 3, the inheritance of the model is shown. VectorVariable is a template on variable type and length of the output of the discretized vector valued function. In future implementations, VectorVariable will not be a specialization of MeshVariable, but of Variable.

5.1.2 Vectors

Simulation descriptions often use the term vector in two similar ways. A variable as defined above may be referred to as a vector, being a member of a vector space, or a vector valued function, the result of which is a vector. In the former, a vector, u, may be in the space of once continuously differentiable functions, C^1, or in the latter, a vector may be the result of such a function. Within AMP, the discretization of the variable is a "vector"—an approximation of a vector valued function taken at points in the domain of the function. As such, vectors are constructed from a variable and a discretization. Currently, the discretization is assumed to be a mesh.

There are a multitude of libraries that implement various vector functions such as inner products, norms, BLAS axpy, and many others. The vector class provides numerous interfaces to perform these operations. AMP leverages existing libraries by exposing a virtual interface to the operator, solver, and simulation designer while implementing wrappers to the libraries in specializations of the interface. In this way, the inheritance model not only indicates which library is used but also provides both a run-time and compile-time selection of library routines for use in existing solver libraries. This is accomplished by providing helper classes named for the libraries represented in the implementation.

Also, by providing classes that construct library-specific data structures for proven solver packages, the vectors in AMP can immediately use popular software libraries for solution of linear and nonlinear systems. For instance, using a PETSc nonlinear solver with an Epetra vector is accomplished by using the PetscVector and EpetraVector interface.

Since the overarching objective is to provide a robust multi-physics simulation environment, vectors also provide a mechanism to combine vectors "loosely" or "tightly." Continuing the example from the previous section, a thermo-mechanical simulation may loosely couple temperature and displacement: separate operators are solved independently and joined through another operator. In this case, the vector allows the combination of the displacement vector and temperature vector into a single vector, even if displacement and temperature are different vectors used in solvers from different packages. This vector, called a MultiVector,

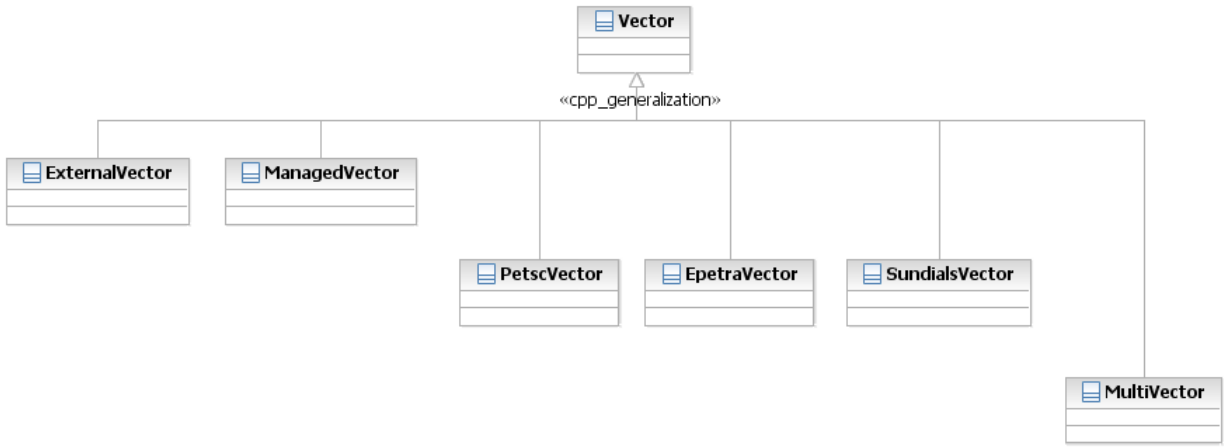

Figure 4: Inheritance model for vector classes.

provides all of the same functionality as its constituents. And, since it is a vector itself, it can be used by solvers and operators just as any other vector.

In Fig. 4, a truncated view of the inheritance model is given. ExternalVector and ManagedVector indicate how the vector is created. An ExternalVector is a package-specific implementation and storage wrapper that allows AMP to use vectors of other packages natively. ManagedVectors are vectors created by AMP, whose memory and communication lists are managed by AMP, and can be seamlessly used with any of the packages used by AMP without copying the contents of the vector.

PetscVector, EpetraVector, and SundialsVector are convenience classes that allow the inheritance model to both choose which engine should be used to perform vector operations, provide a createView mechanism that allows presentation of ManagedVectors to the various packages, and allows access to the underlying vector type for those packages.

Finally, MultiVector is a vector and has a collection of vectors. In this way, multiple different types of vectors can be combined to form a vector used in a coupled simulation where each constituent of the coupled operator uses different packages. Currently, there is no intelligent parallel decision making in this class. In the future, the MultiVector will use domain decomposition information to conclude certain variables only exist on certain subsets of processors allowing for parallelism in operator composition to be exploited.

It is important to note that users should never see the ExternalVector, ManagedVector, or MultiVector classes. These are used by factories such as meshes to generate the appropriate vector type given a variable. If a user uses entirely AMP operators and solvers, then only the base class Vector should be seen. If external packages are required, then the user can use the appropriate class and interface relative to the external package.

5.1.3 Matrices

Matrices are implemented in a similar fashion to vectors. The portion of the inheritance model of vectors related to library choice is mirrored in the matrix hierarchy. In this way, much of the linear algebra required for solution of operators is presented as a uniform interface independent of the library used to perform the mathematics. A user can concentrate on implementation of the physics and use proven OTS libraries interchangeably.

For instance, the PETSc Krylov Solver Package (KSP) can be used with PETSc matrix-vector operations or Trilinos matrix-vector operations, with no distinction made in the AMP operators, solvers, or time integrators. In this case, the interface presented by PETSc allows for various matrix-vector operations be provided by an outside library. Through the matrix and vector interfaces, a run-time decision is made to invoke the appropriate matrix-vector operation, be it the PETSc implementation, the Trilinos implementation, or other implementation.

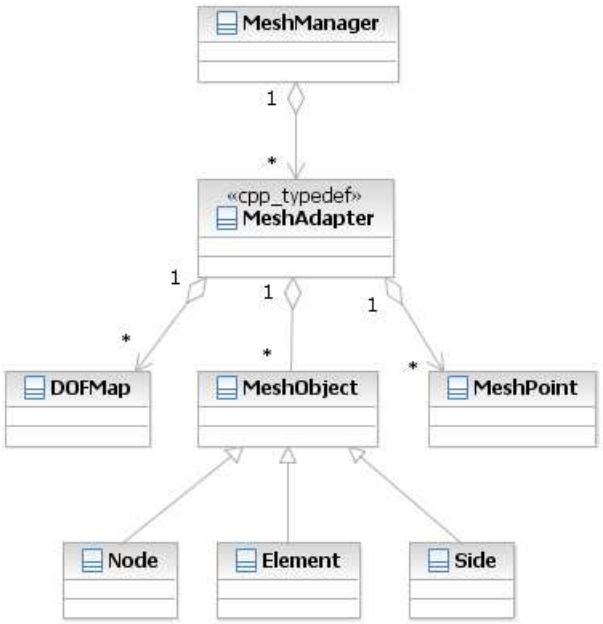

Figure 5: Inheritance model for mesh classes.

5.1.4 Mesh

A mesh is a locally supported discretization of a continuous domain. By discretization, we mean the values of a function on the continuous domain are sampled at a finite number of locations in the domain. By locally supported, we mean a low-order piecewise interpolant through nearby points can be consistent with the continuous function throughout a subset of the domain defined by the points. The points are referred to as nodes, and the subset of the domain as edges, faces, cells or elements, depending on context.

There are several different methods for storing meshes, the most common for scientific applications being "connectivity lists." A connectivity list is two arrays, one that provides locations of nodes and another that provides the nodes that make up cells. Packages such as libMesh rely on connectivity lists to store and manipulate meshes. These packages provide abstractions to intuitive mesh types such as sides, faces, cells, and nodes.

Other packages such as the SIERRA ToolKit rely on two abstract concepts: a mesh object and a relationship. A mesh is defined as a collection of these objects and relationships. In these packages, a mesh object has data, such as displacement, temperature, etc., and dimensionality, a tag that indicates where in the hierarchy of mesh objects this object is placed. A relationship is just that, a member of the set of possible relations of mesh objects. In order to provide support for computational science, the set of possible relations is well-ordered.

These two approaches both provide the basic concepts of cell and node. These approaches also lend themselves to use of the iterator idiom popularized by C++. While random access of nodes and elements may require $O(\log n)$ computation, iterative access is accomplished in $O(1)$. The AMP mesh adapter relies on this idiom to abstract away from the developer the underlying storage mechanism.

Borrowing from the SIERRA ToolKit, the AMP mesh adapter attempts to provide iterators to relatively homogeneous subsets of the mesh. Inasmuch as a mesh can be used to create operators and vectors uniformly, the mesh adapter presents elements to the operator that can be treated in exactly the same way. Since this capability is not present in libMesh, the concept of a mesh manager is used. The mesh manager is a collection of mesh that are homogenous with respect to element and material type.

Fig. 5 shows the inheritance model for the entire mesh utility. The mesh manager class is used on the entire continuum problem. If an operator needs to be composed from several mesh types (be it cells or materials), then the programmer can use the MeshManager class to obtain appropriate adapters for the

operator. Given a variable, a mesh will create a vector that can be used with the operator. On the other hand, if the operator represents a single type of physics, the programmer need only use the adapter class.

5.2 OPERATORS

As mentioned earlier, operators form the core of AMP (Fig. 6). Individual operators are implemented for particular single or multi-physics components that can be combined to form composite multi-physics operators. These individual or composite operators are required to provide a minimal interface that solvers expect in running simulations. The operator interface is meant to be as extensible as desired by the physics developers to meet individual requirements. AMP only requires the user to conform to the minimal interface that the various other components of AMP such as solvers and time integrators expect.

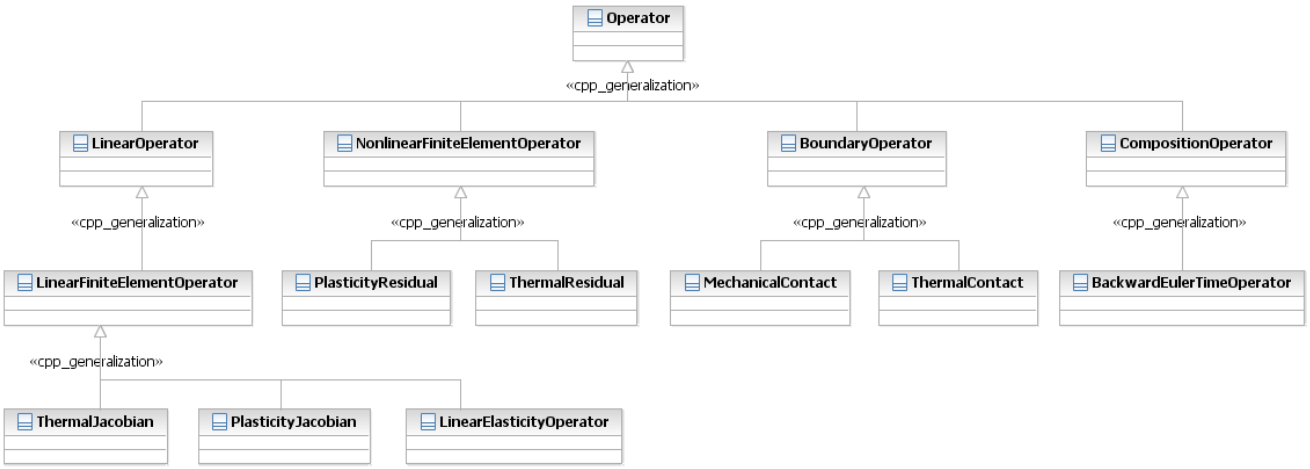

Figure 6: Design of the operators component.

- Construction: The constructor for an operator should use an OperatorParameter class argument derived from DiscreteOperatorParameters. By using a standard parameter interface and changing the members of the OperatorParameter class as requirements evolve over time, we minimize the disruption of initialization in large codes that can otherwise be encountered over time.
- Reset operation: Each operator provides a reset member function that takes an OperatorParameter object for re-initalization of the operator as parameters on which the operator may depend change.
- Apply operation: Each operator is required to provide an apply member function of the form

  $$\text{Operator::apply}(f,\ u,\ r,\ a,\ b).$$

  f, u, r are vectors, and a and b are scalars. The apply operation calculates $r = b * f + a * A(u)$. The default values are $a = -1.0$ and $b = 1.0$ which correspond to calculating the residual. However, this interface can also be used to calculate $A(u)$ by setting $a = 1.0$ and $b = 0.0$. Such an operator is required, for example, in Krylov solvers or the Full Approximation Scheme (FAS) nonlinear solver.
- getJacobianParameters operation: This interface is optional and needs to be implemented only if the user wishes to use a solver that requires a Jacobian, an approximate Jacobian, or components of a Jacobian. The getJacobianParameters member function returns an OperatorParameter object that can be used to construct or reset a Jacobian or approximate Jacobian operator. This interface is designed on purpose to return the parameters necessary to construct a Jacobian or approximate Jacobian rather than the actual operators themselves as different solvers and preconditioners could potentially request different parts or components of the Jacobian of the operator.

As mentioned previously, operators are primarily used to represent maps of the form $\mathcal{L} : X \to Y$ where X and Y are appropriately defined spaces. Below we enumerate a few of the common operator categories that are being designed and implemented.

- **Nonlinear and Linear PDE Discretization Operators:** These operators derived from AMP::DiscreteOperator form the discretizations of partial differential equations (PDEs) on a mesh which is built and accessed through AMPMesh.
- **Boundary Operators:** These are used to represent operators that live on the boundaries of domains. These can either impose boundary conditions or act as coupling operators, for example, in thermal or mechanical contact at the pellet-clad interface.
- **Materials Operators:** Materials operators will in general only implement the construction, apply, and reset interfaces. We choose to represent interfaces to extract material properties as operators also (through the apply operation) because the apply operation to obtain a material property may itself internally be as simple as an algebraic function to a full nonlinear solve or may involve lower-length-scale calculations.
- **Composite Operators:** Operator composition is a natural way to tackle the complex multi-physics couplings encountered in the problems we are tackling (Fig. 7). This allows users to build multi-physics components as individual components rather than as monolithic pieces that need to be constantly rewritten to account for additional physics, though for strongly coupled physics components monolithic components may be the optimal representation. Composition allows us to tackle either form or to use both in conjunction to form extremely complex models.

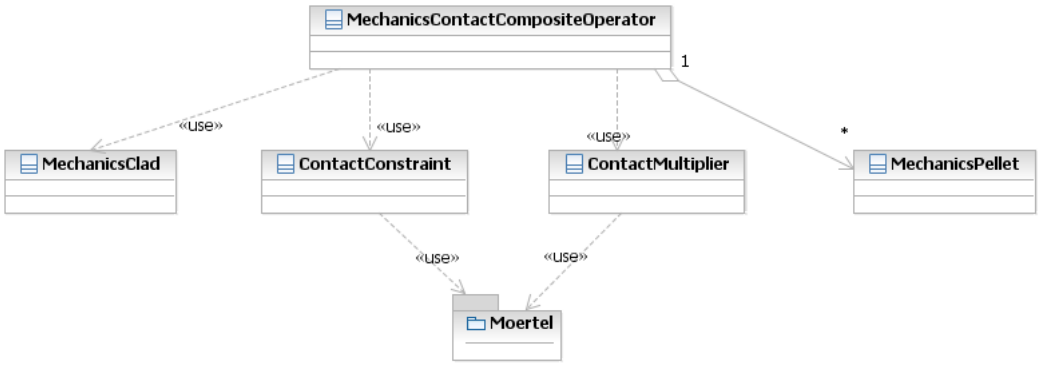

Figure 7: Example of a composite operator.

5.3 SOLVERS

AMP provides a uniform interface to nonlinear and linear solvers and preconditioners (Fig. 8). AMP currently interfaces to the Trilinos and PETSc solver frameworks, and the design of the vector and operator classes allows us to combine solver components from both packages to solve multi-physics problems and potentially from other packages in future. In addition, the SolverStrategy interface allows developers to implement solvers that can then be immediately tested with the multi-physics operators. Fig. 9 demonstrates the use of a Jacobian-Free Newton Krylov (JFNK) method.

- SolverStrategy::SolverStrategy(SolverStrategyParameters parameters): Each solver constructor takes a SolverParameter argument derived from a SolverStrategyParameter class. The SolverParameter can contain a pointer to an operator that can be used to initialize the solver.
- SolverStrategy::reset(SolverStrategyParameters parameters): meant to reset solver parameters.

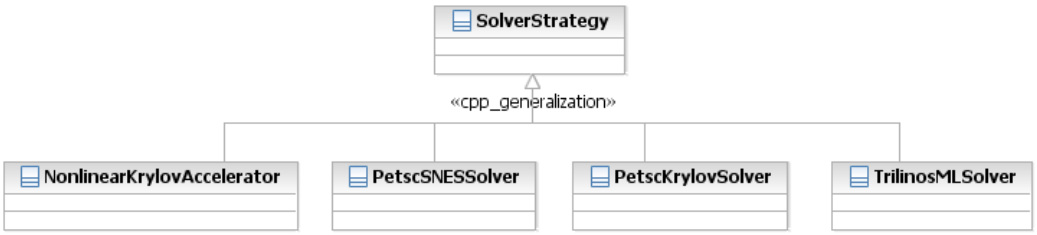

Figure 8: Design of the solvers component.

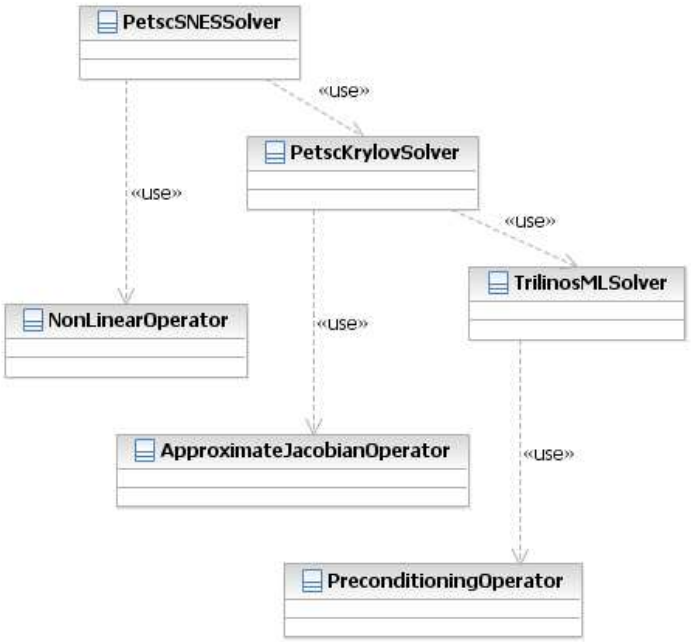

Figure 9: Example of a JFNK solver strategy.

- SolverStrategy::registerOperator(DiscreteOperator op): register an operator, A with the solver.
- SolverStrategy::resetOperator(DiscreteOperator op): reset the operator which is registered with the solver. This could be a quite sophisticated routine in the case of a nonlinear solver if the nonlinear solver internally uses linear solvers and preconditioners (such as in the case of preconditioned Newton-Krylov methods) which have associated operators that need to be reset based on the nonlinear solver reset.
- SolverStrategy::solve(f, u): solves the problem $A(u) = f$ where u and f are vectors.

Concrete derivations of the SolverStrategy class include:

- TrilinosMLSolver: An interface to the ML multilevel solver in Trilinos.
- PETScSNESSolver: An interface to the inexact Newton methods in PETSc.
- PETScKrylovSolver: An interface to the Krylov solver methods in PETSc.

The suite of solvers available in AMP includes a NonlinearKrylovSolver which implements the Accelerated Inexact Newton method (Fig. 10) of Miller and Carlsson. In the future, an interface will be added to the

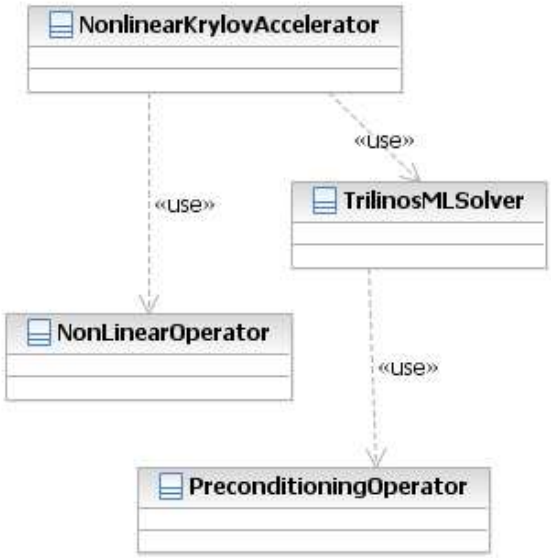

Figure 10: Example of an accelerated inexact newton solver strategy.

Trilinos NOX solver as well.

The uniform interface to the solvers, operators, and backplane components (vectors, matrices, and mesh), allow us to combine, for example, the NonlinearKrylovSolver with a PETScKrylovSolver preconditioned by a TrilinosMLSolver exploiting potential strengths of the various solver packages.

5.4 TIME INTEGRATORS

AMP has been designed to allow for explicit, semi-implicit, and fully implicit time integration (Fig. 11). We note that semi-implicit time integrators depend on the availability of both explicit and implicit time integrators. This explains why the design has to include explicit time integrators. A second reason for explicit time integrators is in the context of debugging where they are extremely useful as a reference calculation as well as a tool to ensure the correctness of nonlinear multi-physics function evaluations. The base class for time integrators is the TimeIntegrator class. It defines the minimal interface required of all time integrators. This interface consists of:

- registerOperator: registers a DiscreteOperator with a TimeIntegrator
- reset : reset the TimIntegrator, for example after a regridding operation
- advanceSolution: attempt to advance the solution
- checkNewSolution: check if the attempt to advance produced a valid solution
- updateSolution: if the criteria to advance to the next time step have been satisfied update internal state
- getNextDt: provides a mechanism for time step control

There are multiple means of configuring and using these time integrators to be able to explore various configurations for each of the physics that will be coupled.

- **Explicit time integrators:** in AMP are currently designed to solve problems of the form:

$$u_t = F(u), \;\; u(t=0) = u0;$$

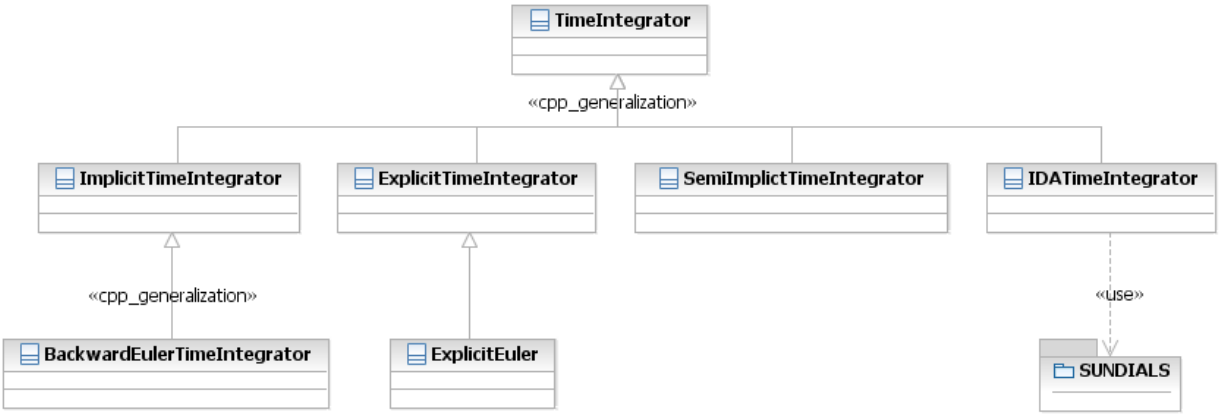

Figure 11: Design of the time integrators component.

A DiscreteOperator implements the operation on the right hand side denoted by $F(u)$ through the DiscreteOperator::apply() interface. The TimeIntegrator registers this operator through the registerOperator interface or at construction time. The explicit time integrators provide explicit time discretizations for the term u_t. Since all that is required of the user is to supply the operator (Fig. 12) which implements F(u) this provides a powerful mechanism for users to experiment with various explicit time integrators.

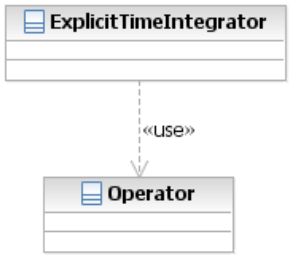

Figure 12: Example of an explicit time integration scheme.

- **Implicit time integrators:** in AMP are currently designed to solve problems of the form:

$$(F_1(u))_t = F_2(u); \quad u(t=0) = u0;$$

where $u0$ is an initial condition vector. The requirement on the user is to provide the DiscreteOperators that implement F_1 and F_2. The user can optionally provide the solvers that the implicit time integrator will need to solve a system of equations at each time step. All the implicit time integrators are derived from an ImplicitTimeIntegrator class that provides default implementations for the TimeIntegrator interfaces. Internally an ImplicitTimeIntegrator constructs a TimeOperator object that implements the discrete form of $(F_1(u))_t - F_2(u)$ specific to the implicit scheme. For example, for backward Euler the internally constructed TimeOperator would implement the operator

$$\frac{(F_1(u))}{\Delta t} - F_2(u)$$

where Δt is the current time step. Every ImplicitTimeIntegrator internally maintains a pointer to a SolverStrategy object which solves the implicit equations encapsulated in the associated TimeOperator (Fig. 13). We note that the operators corresponding to F_1 and F_2 may be formed by composition to enable complex multi-physics simulations.

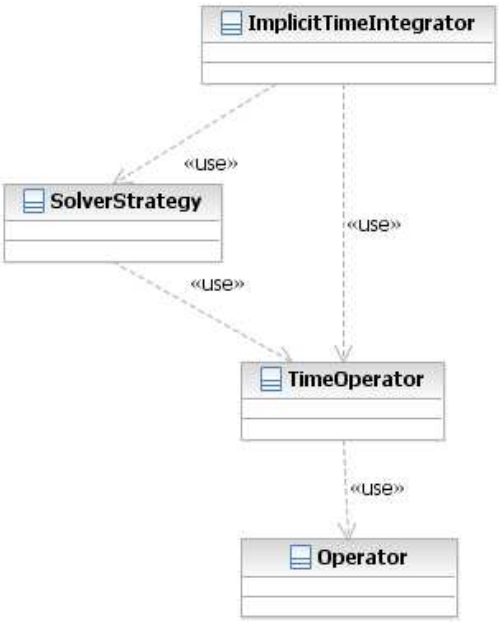

Figure 13: Example of an implicit time integration scheme.

- **Semi-implicit time integrators:** in AMP are currently designed to solve problems of the form:

$$(F_1(u))_t = F_2(u); \;\; u(t=0) = u0;$$

where $u0$ is an initial condition vector. The requirement on the user is to provide the DiscreteOperators that implement F_1 and F_2. It is assumed that the vector, u, represents more than one type of physics. It is in such contexts that semi-implicit integrators make sense. Semi-implicit time integrators will be implemented as controllers that drive a combination of implicit and explicit time integrators in an operator split manner (Fig. 14).

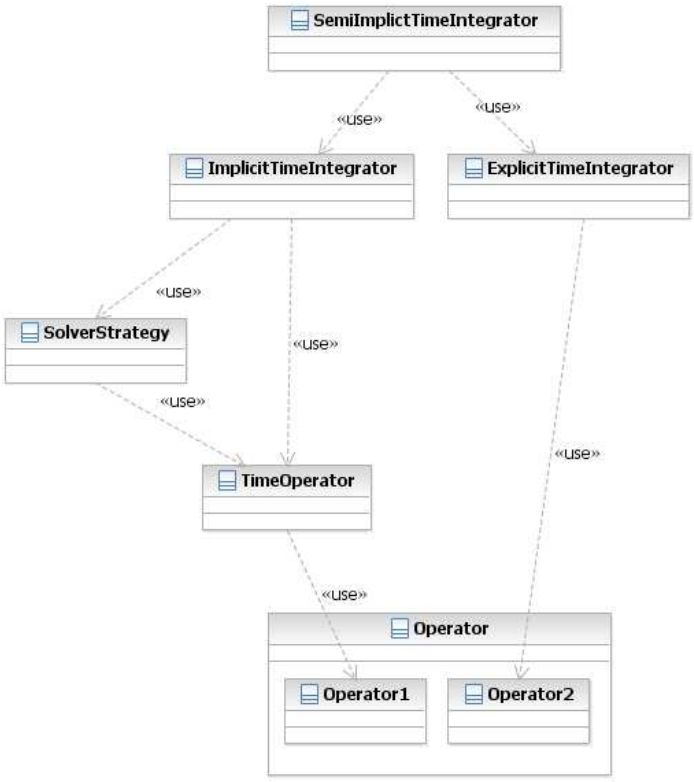

Figure 14: Example of a semi-implicit time integration scheme.

References

[1] J. Turner, K. Clarno, and G. Hansen, Roadmap to an Engineering-Scale Nuclear Fuel Performance Code, Technical Report ORNL/TM-2009/233, Oak Ridge National Laboratory, 2009.

[2] `http://www.pnl.gov/frapcon3/`.

[3] `http://trilinos.sandia.gov/`.

[4] `http://www.mcs.anl.gov/petsc/petsc-as/`.

[5] `https://computation.llnl.gov/casc/sundials/main.html`.

[6] `http://www.ornl.gov/sci/scale/`.

[7] D. Olander, *Fundamental Aspects of Nuclear Reactor Fuel Elements*, University of Michigan Library, 1976.

[8] `http://libmesh.sourceforge.net/`.

[9] `http://nstdsrv.ornl.gov/gf/project/amp/`.

[10] `http://gforge.org/gf/`.

[11] `http://subversion.tigris.org/`.

[12] `http://svnbook.red-bean.com/`.

[13] T. Evans and K. Clarno, C++ Coding Standards for AMP, Technical Report ORNL/TM-2009/240, Oak Ridge National Laboratory, 2009.

[14] `http://www.gnu.org/software/autoconf/`.

Appendices

A DELIVERABLES

1. AMP Planning
   - AMP Software Design Document that describes the software which will be developed, including what physics will be modeled and how the physics will be solved. An ORNL L3 milestone for January 30, 2010.
   - AMP Demonstration Specification Document that describes the demonstration problem which will be solved using the AMP code. An Idaho National Laboratory (INL) L3 milestone for January 30, 2010.
2. AMP User Interface
   - AMP User Interface Software that includes an input specification and reader, mesh generation, output collection, and visualization. An ORNL L3 milestone for April 30, 2010.
   - AMP User Interface Document that describes the AMP user interface. An INL L3 milestone for April 30, 2010.
3. The AMP Code Components
   - Development and integration of components for the TimeIntegrationSolver, ComputationalEngine, Mechanics, Infrastructure, and Neutronics into AMP. An ORNL L3 milestone for August 1, 2010.
   - Development and integration of components for Thermal / Species Diffusion, MaterialProperties, and Damage mechanics into AMP. A Los Alamos National Laboratory (LANL) L3 milestone for August 1, 2010.
   - Development and integration of components for MaterialEquations, Thermal/Mechanical Contact, and PlenumPressure/Volume into AMP. An INL L3 milestone for August 1, 2010.
4. The AMP Code
   - ORNL, INL, and LANL will deliver a new 3D coupled thermal-mechanical-chemical code with links to existing modularized zero-dimensional solvers (depletion, species formation, plenum pressure, and material properties), as well as a simplified flow and power module. The export-controlled AMP code will be distributed through the Radiation Safety Information Computational Center (RSICC). This is an L2 milestone for each of the three laboratories to be delivered on September 30, 2010.
   - In addition to the software, each laboratory will complete the additional tasks to satisfy the L2 milestone.
     - ORNL will perform the set of demonstration problems, specified in the Demonstration Specification.
     - LANL will deliver a document that describes the demonstration problem, results, and conclusions.
     - INL will host an AMP training course for the nuclear community.

B DOXYGEN-GENERATED DOCUMENTATION

The Doxygen-generated documentation from the existing AMP source code is included for reference. The documentation is provided for the following packages, respectively:

- vectors,
- matrices,
- ampmesh,
- materials,
- operators,
- solvers, and
- time integrators.

The page numbers and section labels were generated independently of, and do not correspond with, this document.

amp

amp-0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:28:06 2010

Contents

1 The AMP Documentation System

Authors

Kevin Clarno (ORNL). Glen Hansen (INL). Marius Stan (LANL). Bobby Philip (ORNL). Srdjan Simunovic (ORNL). Gary Dilts (LANL). Abdellatif Yacout (ANL). Jay Billings (ORNL). Rahul Sampath (ORNL). Srikanth Allu (ORNL). Pallab Barai (ORNL). Phani Nukala (ORNL). Jung Ho Lee (ORNL).

Executive Summary

AMP: Adfero Minuo Physiologus (latin: to contribute diminishing physics), Advanced Modeling of Phuel, or Another Multi-Physics code

Components

AMP consists of the following components. Click on the links to find documentation about each component.

- NewPackage
- ampmesh
- comm
- harness
- materials
- matrices
- operators
- solvers
- time_integrators
- utils
- vectors

vectors

amp-0_0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:27:56 2010

Contents

1 Overview of the vectors package

Version:

amp-0_0_0

Vectors package in amp

2 Class Documentation

2.1 AMP::Vector Class Reference

Class Vector is a base class for the vector types in AMP.

`#include <Vector.h>`

Inherited by AMP::DualVector, AMP::EpetraVector `[virtual]`, AMP::ManagedVector `[virtual]`, AMP::MultiVector, AMP::NativeVector `[virtual]`, and AMP::PetscVector `[virtual]`.

Public Types

- enum **ScatterType** { **BROADCAST**, **GATHER_SCATTER** }
- enum **UpdateState** { **NOT_UPDATING**, **ADDING**, **SETTING** }
- typedef boost::shared_ptr< Vector > **shared_ptr**

Public Member Functions

- Vector (VectorParameters::shared_ptr parameters)

Constructor for Vector class is used to construct each unique vector within an application.

- virtual ~Vector ()

  Virtual destructor for Vector class.

- virtual boost::shared_ptr< ParameterBase > **getParameters** ()=0
- void setVariable (const Variable::shared_ptr name)

  Set string identifier for this vector object.

- void setOutputStream (std::ostream &s)

  Set output stream for vector object.

- std::ostream & getOutputStream ()

  Return reference to the output stream used by this vector object.

- const Variable::shared_ptr getVariable () const

  Return Variable identifier for this vector object.

- virtual shared_ptr cloneVector (const Variable::shared_ptr name) const =0

  Clone this vector object and return a pointer to the vector copy (i.e., a new vector).

- shared_ptr **cloneVector** () const
- shared_ptr **cloneVector** (const char *name)
- virtual shared_ptr subsetVectorForVariable (const Variable::shared_ptr &name)

  Select the portion of the vector that corresponds to this variable.

- virtual void freeVectorComponents ()

  Destroy the storage corresponding to the vector components and free the associated patch data entries from the variable database (which will also clear the indices from the patch descriptor).

- virtual void allocateVectorData (const double timestamp=0.0)

  Allocate data storage for all components of this vector object.

- virtual void deallocateVectorData ()

  Deallocate data storage for all components of this vector object.

- template<typename RETURN_TYPE>
  RETURN_TYPE * getRawDataBlock ()

  Access a block of data of type RETURN_TYPE used to store the values of the vector.

- template<typename RETURN_TYPE>
  const RETURN_TYPE * **getRawDataBlock** () const
- virtual void copyVector (const Vector &src_vec)=0

  Copy data from source vector components to components of this vector.

- void **copyVector** (const shared_ptr &src_vec)
- virtual void swapVectors (Vector &other)=0

  Swap data components (i.e.

- void **swapVectors** (shared_ptr &other)
- virtual void aliasVector (Vector &other, size_t offset=0)=0

  Alias this vector with the argument vector or offset into the argument vector.

- void **aliasVector** (shared_ptr &other, size_t offset=0)

Vector arithmetic functions

- virtual void setToScalar (double alpha)=0

  Set all components of this vector to given scalar value.

- virtual void scale (double alpha, const Vector &x)=0

  Set this vector to src vector multiplied by given scalar.

- void scale (double alpha, const shared_ptr &x)

  Set all components of this vector to given scalar value.

- virtual void scale (double alpha)=0

  Multiply this vector by given scalar.

- virtual void addScalar (const Vector &x, double alpha)

  Set this vector to sum of given vector and scalar.

- void addScalar (const shared_ptr &x, double alpha)

  Set all components of this vector to given scalar value.

- virtual void add (const Vector &x, const Vector &y)=0

  Set this vector to sum of two given vectors.

- void add (const shared_ptr &x, const shared_ptr &y)

  Set all components of this vector to given scalar value.

- virtual void subtract (const Vector &x, const Vector &y)=0

  Set this vector to difference of two given vectors (i.e., x - y).

- void subtract (const shared_ptr &x, const shared_ptr &y)

  Set all components of this vector to given scalar value.

- virtual void multiply (const Vector &x, const Vector &y)=0

  Set each entry in this vector to product of corresponding entries in input vectors.

- void multiply (const shared_ptr &x, const shared_ptr &y)

  Set all components of this vector to given scalar value.

- virtual void divide (const Vector &x, const Vector &y)=0

  Set each entry in this vector to ratio of corresponding entries in input vectors (i.e., this = x .

- void divide (const shared_ptr &x, const shared_ptr &y)

  Set all components of this vector to given scalar value.

- virtual void reciprocal (const Vector &x)=0

  Set each entry of this vector to reciprocal of corresponding entry in input vector.

- void reciprocal (const shared_ptr &x)

  Set all components of this vector to given scalar value.

- virtual double minQuotient (const Vector &x, const Vector &y)

  Set all components of this vector to given scalar value.

- double minQuotient (const shared_ptr &x, const shared_ptr &y)

  Set all components of this vector to given scalar value.

- virtual double wrmsNorm (const Vector &x, const Vector &y)

  Set all components of this vector to given scalar value.

- double wrmsNorm (const shared_ptr &x, const shared_ptr &y)

  Set all components of this vector to given scalar value.

- virtual void linearSum (double alpha, const Vector &x, double beta, const Vector &y)=0

  Set this vector to the linear sum $\alpha x + @betay$, where $\alpha, @beta$ are scalars and x, y are vectors.

- void linearSum (double alpha, const shared_ptr &x, double beta, const shared_ptr &y)

  Set all components of this vector to given scalar value.

- virtual void axpy (double alpha, const Vector &x, const Vector &y)=0

  Set this vector to the sum $\alpha x + y$, where α is a scalar and x, y are vectors.

- void axpy (double alpha, const shared_ptr &x, const shared_ptr &y)
  Set all components of this vector to given scalar value.
- virtual void axpby (double alpha, double beta, const Vector &x)=0
  this = alpha∗x + beta∗this
- void axpby (double alpha, double beta, const shared_ptr &x)
  Set all components of this vector to given scalar value.
- virtual void abs (const Vector &x)=0
  Set each entry of this vector to absolute values of corresponding entry in input vector.
- void abs (const shared_ptr &x)
  Set all components of this vector to given scalar value.
- virtual double min (void) const =0
  Return the minimum data entry in this vector.
- virtual double max (void) const =0
  Return the maximum entry of this vector.
- virtual void setRandomValues (void)=0
  Set data in this vector to random values.
- virtual void setValuesByLocalID (int num, int ∗indices, double ∗vals)=0
  Set all components of this vector to given scalar value.
- virtual void setValueByLocalID (int i, double val)
  Set all components of this vector to given scalar value.
- virtual void setLocalValuesByGlobalID (int num, int ∗indices, double ∗vals)=0
  Set all components of this vector to given scalar value.
- virtual void setLocalValueByGlobalID (int i, double val)
  Set all components of this vector to given scalar value.
- virtual void setValuesByGlobalID (int num, int ∗indices, double ∗vals)
  Set all components of this vector to given scalar value.
- virtual void setValueByGlobalID (int i, double val)
  Set all components of this vector to given scalar value.
- virtual void addValuesByLocalID (int num, int ∗indices, double ∗vals)=0

Set all components of this vector to given scalar value.

- virtual void addValueByLocalID (int i, double val)
  Set all components of this vector to given scalar value.
- virtual void addLocalValuesByGlobalID (int num, int *indices, double *vals)=0
  Set all components of this vector to given scalar value.
- virtual void addLocalValueByGlobalID (int i, double val)
  Set all components of this vector to given scalar value.
- virtual void addValuesByGlobalID (int num, int *indices, double *vals)
  Set all components of this vector to given scalar value.
- virtual void addValueByGlobalID (int i, double val)
  Set all components of this vector to given scalar value.
- virtual void getValuesByGlobalID (int numVals, int *ndx, double *vals) const
  Set all components of this vector to given scalar value.
- virtual void getLocalValuesByGlobalID (int numVals, int *ndx, double *vals) const =0
  Set all components of this vector to given scalar value.
- virtual double getValueByGlobalID (int i) const
  Set all components of this vector to given scalar value.
- virtual void getValuesByLocalID (int numVals, int *ndx, double *vals) const
  Set all components of this vector to given scalar value.
- virtual double getValueByLocalID (int ndx) const
  Set all components of this vector to given scalar value.
- virtual void makeConsistent (ScatterType t=GATHER_SCATTER)
  Set all components of this vector to given scalar value.
- virtual void assemble ()=0
  Set all components of this vector to given scalar value.
- virtual double L1Norm (void) const =0
  Return discrete L_1 -norm of this vector using the control volume to weight the contribution of each data entry to the sum.

- virtual double L2Norm (void) const =0
  Return discrete L_2 -norm of this vector using the control volume to weight the contribution of each data entry to the sum.
- virtual double maxNorm (void) const =0
  Return the max *-norm of this vector.*
- virtual double dot (const Vector &x) const =0
  Return the dot product of this vector with the argument vector.
- double dot (const shared_ptr &x)
  Set all components of this vector to given scalar value.
- virtual unsigned int getLocalSize () const =0
  Set all components of this vector to given scalar value.
- virtual unsigned int getGlobalSize () const =0
  Set all components of this vector to given scalar value.
- virtual unsigned int getGhostSize () const
  Set all components of this vector to given scalar value.
- virtual void setCommunicationList (CommunicationList::shared_ptr comm)
  Set all components of this vector to given scalar value.

Protected Member Functions

- virtual void * **getRawDataBlockAsVoid** ()=0
- virtual const void * **getRawDataBlockAsVoid** () const =0
- virtual void **addCommunicationListToParameters** (CommunicationList::shared_ptr)
- void **aliasGhostBuffer** (shared_ptr in)

Protected Attributes

- CommunicationList::shared_ptr **d_CommList**
- UpdateState **d_UpdateState**
- Variable::shared_ptr **d_pVariable**

2.1.1 Detailed Description

Class Vector is a base class for the vector types in AMP.

Specifically, this class provides a set of common vector operations to manipulate all of the data components as a whole. The most obvious use of this class is in AMP applications that use solver libraries, such as PETSc or Trilinos. Specific vector objects that can be used with these packages are defined elsewhere in AMP. However, all these vactor interfaces are built using this vector class.

Before the vector operations can be used, the storage for each of its components must be allocated. Storage allocation is only possible through a vector object after all component variables are added to the vector (using the addComponent() function). Then, the allocateVectorData() function will allocate storage for all components when called. Alternatively, patch data objects (corresponding to the variables and vector patch data indices) may be explicitly created elsewhere. However, depending on the circumstance, this second alternative may be more confusing and require more bookkeeping on the user's part. See the documentation accompanying the addComponent() function for more information.

Definition at line 64 of file Vector.h.

2.1.2 Constructor & Destructor Documentation

2.1.2.1 AMP::Vector::Vector (VectorParameters::shared_ptr *parameters*)

Constructor for Vector class is used to construct each unique vector within an application.

That is, each vector that is used to represent a unique set of variable quantities is considered unique. This constructor is used to create a solution vector for an application or solver algorithm. The cloneVector() function is provided to generate copies of a given vector. For example, the clone function may be used by a solver to generate copies of the vector as needed; e.g., in a Krylov subspace method like GMRES.

Before the vector may be used, data components must be added to it using the addd-Component() function. Also, this constructor does not allocate storage for vector data. This is usually done after all components are added. The allocateVectorData() function is used for this purpose. Otherwise, existing patch data quantities can be added as vector components. In any case, storage for all components must be allocated before the vector can be used.

It is important to note that a non-recoverable assertion will result if the specified levels do not exist in the hierarchy before a vector object is used, or if the hierarchy pointer itself is null. The range levels can be reset at any time (e.g., if the level configuration changes by re-meshing), by calling the resetLevels() member function.

Although an empty string may be passed as the vector name, it is recommended that a descriptive name be used to facilitate debugging and error reporting.

By default the vector component information and data will be sent to the "plog" output stream when the print() function is called. This stream can be changed at any time via the setOutputStream() function.

Definition at line 45 of file Vector.cc.

2.1.2.2 AMP::Vector::~Vector () [virtual]

Virtual destructor for Vector class.

The destructor destroys all vector component information. However, the destructor does not deallocate the vector component storage, nor does it return the vector patch data indices to the patch descriptor free list. The freeVectorComponents() function is provided for this task. The reason for this is that an application may create a vector based on some pre-existing patch data objects that must live beyond the destruction of the vector object.

Definition at line 69 of file Vector.cc.

2.1.3 Member Function Documentation

2.1.3.1 void AMP::Vector::setOutputStream (std::ostream & *s*)

Set output stream for vector object.

When the print() function is called, all vector data will be sent to the given output stream.

2.1.3.2 std::ostream& AMP::Vector::getOutputStream ()

Return reference to the output stream used by this vector object.

This function is primarily used by classes which define interfaces between this vector class and vector kernels defined by other packages. Specifically, AMP vectors and package-specific wrappers for those vectors may all access the same output stream.

2.1.3.3 virtual shared_ptr AMP::Vector::cloneVector (const Variable::shared_ptr *name*) const [pure virtual]

Clone this vector object and return a pointer to the vector copy (i.e., a new vector).

If an empty string is passed in, the name of this vector object is used for the new vector.

Referenced by addScalar(), minQuotient(), and wrmsNorm().

2.1.3.4 void AMP::Vector::allocateVectorData (const double *timestamp* = 0.0) [virtual]

Allocate data storage for all components of this vector object.

If no memory arena is specified, then the standard memory arena will be used.

Definition at line 142 of file Vector.cc.

2.1.3.5 void AMP::Vector::deallocateVectorData () [virtual]

Deallocate data storage for all components of this vector object.

Note that this routine will not free the associated data indices in the patch descriptor. See freeVectorComponents() function.

Definition at line 147 of file Vector.cc.

Referenced by freeVectorComponents().

2.1.3.6 virtual void AMP::Vector::swapVectors (Vector & *other*) [pure virtual]

Swap data components (i.e.

storage) between this vector object and argument vector.

2.1.3.7 virtual void AMP::Vector::multiply (const Vector & *x*, const Vector & *y*) [pure virtual]

Set each entry in this vector to product of corresponding entries in input vectors.

(i.e., this = x .∗ y)

Referenced by multiply().

2.1.3.8 virtual void AMP::Vector::divide (const Vector & *x*, const Vector & *y*) [pure virtual]

Set each entry in this vector to ratio of corresponding entries in input vectors (i.e., this = x .

/ y). No check for division by zero.

Referenced by divide().

2.1.3.9 virtual void AMP::Vector::reciprocal (const Vector & *x*) [pure virtual]

Set each entry of this vector to reciprocal of corresponding entry in input vector.

No check is made for division by zero.

Referenced by reciprocal().

2.1.3.10 virtual double AMP::Vector::min (void) const [pure virtual]

Return the minimum data entry in this vector.

Note that this routine returns a global min over all vector components and makes no adjustment for coarser level vector data that may be masked out by the existence of underlying fine values. In particular, the control volumes are not used in this operation. This may change based on user needs.

2.1.3.11 virtual double AMP::Vector::max (void) const [pure virtual]

Return the maximum entry of this vector.

Note that this routine returns a global max over all vector components and makes no adjustment for coarser level vector data that may be masked out by the existence of underlying fine values. In particular, the control volumes are not used in this operation. This may change based on user needs.

2.1.3.12 virtual double AMP::Vector::L1Norm (void) const [pure virtual]

Return discrete L_1 -norm of this vector using the control volume to weight the contribution of each data entry to the sum.

That is, the return value is the sum $\sum_i (\|data_i\| cvol_i)$. If the control volume is not defined for a component, the contribution is $\sum_i (\|data_i\|)$ for that data component. Thus, to have a consistent norm calculation all components must have control volumes, or no control volumes should be used at all.

2.1.3.13 virtual double AMP::Vector::L2Norm (void) const [pure virtual]

Return discrete L_2 -norm of this vector using the control volume to weight the contribution of each data entry to the sum.

That is, the return value is the sum $\sqrt{\sum_i ((data_i)^2 cvol_i)}$. If the control volume is not defined for a component, the contribution is $\sqrt{\sum_i ((data_i)^2)}$ for that data component. Thus, to have a consistent norm calculation all components must have control volumes, or no control volumes should be used at all.

2.1.3.14 virtual double AMP::Vector::maxNorm (void) const [pure virtual]

Return the $\max$ -norm of this vector.

If control volumes are defined for all components, the return value is the max norm over all data values where the control volumes are non-zero. If the control volume is not defined for a component, its contribution to the norm will take a max over all of

its data values. Thus, to have a consistent norm calculation all components must have control volumes, or no control volumes should be used at all.

2.1.3.15 virtual double AMP::Vector::dot (const Vector & *x*) const [pure virtual]

Return the dot product of this vector with the argument vector.

If control volumes are defined for all components, the return value is a weighted sum involving all data values where the control volumes are non-zero. If the control volume is not defined for a component, its contribution to the sum will involve all of its data values. Thus, to have a consistent dot product calculation all components must have control volumes, or no control volumes should be used at all.

Referenced by dot().

The documentation for this class was generated from the following files:

- Vector.h
- Vector.cc

Index

matrices

amp-0_0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:28:00 2010

Contents

1 Overview of the matrices package

Version:

amp-0_0_0

Matrices package in amp

2 Class Documentation

2.1 AMP::ManagedPetscMatrix Class Reference

A wrapper around PETSc's Mat object.

`#include <ManagedPetscMatrix.h>`

Inherits AMP::PetscMatrix, and AMP::ManagedEpetraMatrix.

Public Member Functions

- **ManagedPetscMatrix** (ParametersPtr params)
- void **copyFromMat** (Mat)

- virtual Vector::shared_ptr **getRightVector** ()
- virtual Vector::shared_ptr **getLeftVector** ()
- shared_ptr **cloneMatrix** () const

Static Public Member Functions

- static Matrix::shared_ptr **duplicateMat** (Mat)

Protected Member Functions

- **ManagedPetscMatrix** (const ManagedPetscMatrix &rhs)
- void **initPetscMat** ()

2.1.1 Detailed Description

A wrapper around PETSc's Mat object.

Definition at line 20 of file ManagedPetscMatrix.h.

The documentation for this class was generated from the following files:

- ManagedPetscMatrix.h
- ManagedPetscMatrix.cc

2.2 AMP::Matrix Class Reference

A pure virtual base class to store Matrices.

`#include <Matrix.h>`

Inherited by AMP::EpetraMatrix `[virtual]`, AMP::ManagedMatrix `[virtual]`, AMP::NativePetscMatrix, and AMP::PetscMatrix `[virtual]`.

Public Types

- typedef boost::shared_ptr< Matrix > **shared_ptr**
- typedef MatrixParameters::shared_ptr **ParametersPtr**

Public Member Functions

- **Matrix** (ParametersPtr params)
- virtual void mult (const Vector &in, Vector &out)=0

  This function must be implemented by the child classes.

- void **mult** (const boost::shared_ptr< Vector > &in, boost::shared_ptr< Vector > &out)
- virtual shared_ptr **cloneMatrix** () const =0
- virtual void **scale** (double alpha)=0
- virtual void **axpy** (double alpha, const Matrix &x)=0
- void **axpy** (double alpha, const Matrix::shared_ptr &x)
- virtual void **addValuesByGlobalID** (int num_rows, int num_cols, int *rows, int *cols, double *values)=0
- virtual void **setValuesByGlobalID** (int num_rows, int num_cols, int *rows, int *cols, double *values)=0
- virtual void **addValueByGlobalID** (int row, int col, double value)
- virtual void **setValueByGlobalID** (int row, int col, double value)
- virtual void **setScalar** (double)=0
- void **zero** ()
- virtual void **getRowByGlobalID** (int row, std::vector< unsigned int > &cols, std::vector< double > &values) const =0
- virtual void **setDiagonal** (const Vector &in)=0
- void **setDiagonal** (const Vector::shared_ptr &in)
- virtual void **makeConsistent** ()=0
- virtual Vector::shared_ptr **extractDiagonal** (Vector::shared_ptr buf=Vector::shared_ptr())=0
- virtual Vector::shared_ptr **getRightVector** ()=0
- virtual Vector::shared_ptr **getLeftVector** ()=0
- virtual double **L1Norm** ()=0

Protected Member Functions

- **Matrix** (const Matrix &)

2.2.1 Detailed Description

A pure virtual base class to store Matrices.

Definition at line 22 of file Matrix.h.

2.2.2 Member Function Documentation

2.2.2.1 virtual void AMP::Matrix::mult (const Vector & *in*, Vector & *out*) [pure virtual]

This function must be implemented by the child classes.

Parameters:

in input vector

out output vector. The result of multiplying this matrix with in.

Implemented in AMP::NativePetscMatrix.

The documentation for this class was generated from the following files:

- Matrix.h
- Matrix.cc

2.3 AMP::NativePetscMatrix Class Reference

A wrapper around PETSc's Mat object.

`#include <NativePetscMatrix.h>`

Inherits AMP::Matrix.

Public Member Functions

- **NativePetscMatrix** (Mat m, bool internally_created=false)
- void **copyFromMat** (Mat)
- virtual void mult (const Vector &in, Vector &out)

  This function must be implemented by the child classes.

- virtual shared_ptr **cloneMatrix** () const
- Mat **getMat** ()
- virtual Vector::shared_ptr **getRightVector** ()
- virtual Vector::shared_ptr **getLeftVector** ()
- virtual void **scale** (double alpha)
- virtual void **axpy** (double alpha, const Matrix &x)
- virtual void **addValuesByGlobalID** (int num_rows, int num_cols, int *rows, int *cols, double *values)
- virtual void **setValuesByGlobalID** (int num_rows, int num_cols, int *rows, int *cols, double *values)
- virtual void **getRowByGlobalID** (int row, std::vector< unsigned int > &cols, std::vector< double > &values) const
- virtual void **setScalar** (double)
- virtual void **setDiagonal** (const Vector &in)
- virtual void **makeConsistent** ()
- virtual Vector::shared_ptr **extractDiagonal** (Vector::shared_ptr p=Vector::shared_ptr())
- virtual double **L1Norm** ()

Static Public Member Functions

- static Matrix::shared_ptr **duplicateMat** (Mat)

Protected Attributes

- Mat **d_Mat**

2.3.1 Detailed Description

A wrapper around PETSc's Mat object.

Definition at line 18 of file NativePetscMatrix.h.

2.3.2 Member Function Documentation

2.3.2.1 void AMP::NativePetscMatrix::mult (const Vector & *in*, Vector & *out*) [virtual]

This function must be implemented by the child classes.

Parameters:

in input vector

out output vector. The result of multiplying this matrix with in.

Implements AMP::Matrix.

Definition at line 73 of file NativePetscMatrix.cc.

The documentation for this class was generated from the following files:

- NativePetscMatrix.h
- NativePetscMatrix.cc

2.4 AMP::PetscMatrix Class Reference

A wrapper around PETSc's Mat object.

`#include <PetscMatrix.h>`

Inherits AMP::Matrix.

Inherited by AMP::ManagedPetscMatrix.

Public Member Functions

- **PetscMatrix** (ParametersPtr params)
- virtual Mat & **getMat** ()
- virtual Mat **getMat** () const

Static Public Member Functions

- static shared_ptr **createView** (shared_ptr)

Protected Member Functions

- **PetscMatrix** (const PetscMatrix &rhs)

Protected Attributes

- bool **d_MatCreatedInternally**
- Mat **d_Mat**

2.4.1 Detailed Description

A wrapper around PETSc's Mat object.

Definition at line 17 of file PetscMatrix.h.

The documentation for this class was generated from the following files:

- PetscMatrix.h
- PetscMatrix.cc

Index

ampmesh

amp-0_0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:28:02 2010

Contents

1 Overview of the ampmesh package

Version:

amp-0_0_0

Ampmesh package in amp

2 Class Documentation

2.1 AMP::BCDirichlet Class Reference

A class to store a Dirichlet boundary condition.

`#include <BCDirichlet.h>`

Public Attributes

- unsigned int **d_GlobalDOFid**
- double **d_Value**

2.1.1 Detailed Description

A class to store a Dirichlet boundary condition.

Definition at line 10 of file BCDirichlet.h.

The documentation for this class was generated from the following file:

- BCDirichlet.h

2.2 AMP::BCPointForce Class Reference

A class to store an applied point force.

`#include <BCPointForce.h>`

Public Attributes

- unsigned int **d_GlobalDOFid**
- double **d_Value**

2.2.1 Detailed Description

A class to store an applied point force.

Definition at line 10 of file BCPointForce.h.

The documentation for this class was generated from the following file:

- BCPointForce.h

2.3 AMP::MeshUtils Class Reference

A class for managing the mesh related operations required by the Operators.

`#include <MeshUtils.h>`

Public Member Functions

- **MeshUtils** (const boost::shared_ptr< MeshAdapter > &mesh, int numSystemVariables, MPI_Comm comm)
- **MeshUtils** (const boost::shared_ptr< Mesh > &mesh, int numSystemVariables, MPI_Comm comm)
- void **loadPointForces** (char *fname)
- void **loadDirichletValues** (char *fname)
- void **scalePointForces** (double scale)
- void **scaleDirichletValues** (double scale)
- boost::shared_ptr< Mesh > **getMesh** ()

- MeshAdapter::shared_ptr **getMeshAdapter** ()
- unsigned int **getNumPointForces** ()
- BCPointForce **getPointForce** (unsigned int j)
- unsigned int **getNumDirichlet** ()
- BCDirichlet **getDirichlet** (unsigned int j)
- unsigned int **getLocalId** (unsigned int globalId)
- template<typename T>
  void createGhostedVector (const std::vector< T > &inVec, std::vector< T > &ghostedVec, MPI_Datatype datatype)

  Scatters values from the non-ghosted vector into the ghosted vector.

- bool **isDirichlet** (unsigned int globalId)
- const EquationSystems & **get_equation_systems** () const

Protected Types

- enum **DOFtype** { **DIRICHLET**, **FREE** }

Protected Member Functions

- void **init** (int numSystemVariables)

Protected Attributes

- std::vector< DOFtype > **d_DOFtypeList**
- std::vector< unsigned int > **d_PartitionInfo**
- std::vector< unsigned int > **d_DOFlist**
- std::vector< int > **d_GhostSideSizes**
- std::vector< int > **d_OwnerSideSizes**
- std::vector< int > **d_GhostSideDisps**
- std::vector< int > **d_OwnerSideDisps**
- std::vector< unsigned int > **d_OwnerSideLocalIds**
- MPI_Comm **d_Comm**
- boost::shared_ptr< MeshAdapter > **d_MeshPtr**
- EquationSystems **d_Equation_systems**
- std::vector< BCPointForce > **d_PointForces**
- std::vector< BCDirichlet > **d_DirichletValues**

2.3.1 Detailed Description

A class for managing the mesh related operations required by the Operators.

Definition at line 28 of file MeshUtils.h.

2.3.2 Member Function Documentation

2.3.2.1 template<typename T> void AMP::MeshUtils::createGhostedVector (const std::vector< T > & *inVec*, std::vector< T > & *ghostedVec*, MPI_Datatype *datatype*) [inline]

Scatters values from the non-ghosted vector into the ghosted vector.

Parameters:

inVec non-ghosted vector of type T

ghostedVec ghosted vector of type T

datatype MPI_Datatype for T

Definition at line 214 of file MeshUtils.cc.

The documentation for this class was generated from the following files:

- MeshUtils.h
- MeshUtils.cc

Index

materials

amp-0_0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:28:01 2010

Contents

1 Overview of the vectors package

Version:

amp-0_0_0

Vectors package in amp

2 Class Documentation

2.1 AMP::materials::DefaultTraits Class Reference

Provide complete list of default undefined properties.

`#include <Material.h>`

Public Types

- typedef UndefinedTC **ThermalConductivity_t**
- typedef UndefinedPR **PoissonRatio_t**
- typedef UndefinedD **Density_t**

2.1.1 Detailed Description

Provide complete list of default undefined properties.

This class should be the base class for any material Traits class used as a template argument to MaterialBase.

Definition at line 32 of file Material.h.

The documentation for this class was generated from the following file:

- Material.h

2.2 AMP::materials::Material Class Reference

Root material abstract base class.

`#include <Material.h>`

Inherited by AMP::materials::MaterialBase< Traits >, and AMP::materials::MaterialBase< AMP::materials::DefaultTraits >.

Public Member Functions

- virtual void **thermalconductivity** (double *r, const double *T, const double *U, const unsigned int n)=0
- virtual void **poissonratio** (double *r, const unsigned int n)=0
- virtual void **density** (double *r, const double *T, const unsigned int n)=0

2.2.1 Detailed Description

Root material abstract base class.

Definition at line 41 of file Material.h.

The documentation for this class was generated from the following file:

- Material.h

2.3 AMP::materials::MaterialBase< Traits > Class Template Reference

Provides the complete list of material properties that can be supplied by a material class.

`#include <Material.h>`

Inherits AMP::materials::Material.

Inherited by AMP::materials::UndefinedMaterial.

Public Member Functions

- double **thermalconductivityScalar** (const double T, const double U)
- double **poissonratioScalar** ()
- double **densityScalar** (double T)
- virtual void **thermalconductivity** (double *r, const double *T, const double *U, const unsigned int n)
- virtual void **poissonratio** (double *r, const unsigned int n)
- virtual void **density** (double *r, const double *T, const unsigned int n)

2.3.1 Detailed Description

template<class Traits> class AMP::materials::MaterialBase< Traits >

Provides the complete list of material properties that can be supplied by a material class.

Each is expected to be derived from the property classes in Property.h.

Parameters:

Traits should be a subclass of DefaultTraits.

Definition at line 56 of file Material.h.

The documentation for this class was generated from the following file:

- Material.h

2.4 Prop0< type > Class Template Reference

A helper class to minimize typing.

`#include <Helpers.h>`

Public Member Functions

- double **eval** ()
- virtual void **evalv** (double *r, const unsigned int n)

2.4.1 Detailed Description

template<PropertyType type> class Prop0< type >

A helper class to minimize typing.

Definition at line 22 of file Helpers.h.

The documentation for this class was generated from the following file:

- Helpers.h

2.5 Prop1< type > Class Template Reference

A helper class to minimize typing.

`#include <Helpers.h>`

Public Member Functions

- double **eval** (const double)
- virtual void **evalv** (double *r, const double *a0, const unsigned int n)

2.5.1 Detailed Description

template<PropertyType type> class Prop1< type >

A helper class to minimize typing.

Definition at line 40 of file Helpers.h.

The documentation for this class was generated from the following file:

- Helpers.h

2.6 Prop2< type > Class Template Reference

A helper class to minimize typing.

`#include <Helpers.h>`

Public Member Functions

- double **eval** (const double, const double)
- virtual void **evalv** (double *r, const double *a0, const double *a1, const unsigned int n)

2.6.1 Detailed Description

template<PropertyType type> class Prop2< type >

A helper class to minimize typing.

Definition at line 58 of file Helpers.h.

The documentation for this class was generated from the following file:

- Helpers.h

2.7 Prop2Param< type > Class Template Reference

A helper class to minimize typing.

`#include <Helpers.h>`

Public Member Functions

- **Prop2Param** (double *param, unsigned int nparam)
- double **eval** (const double, const double)
- virtual void **evalv** (double *r, const double *a0, const double *a1, const unsigned int n)

2.7.1 Detailed Description

template<PropertyType type> class Prop2Param< type >

A helper class to minimize typing.

Definition at line 76 of file Helpers.h.

The documentation for this class was generated from the following file:

- Helpers.h

2.8 AMP::materials::Property0D< Number > Class Template Reference

Provides a base class for obtaining constants from a property.

`#include <Property.h>`

Inherits AMP::materials::PropertyBase< Number >.

Public Member Functions

- **Property0D** (const PropertySpec< Number > spec)
- virtual void evalv (Number *result, const unsigned int n)

  Each subclass will have a virtual evaluator for an array of input values.

- template<>
  void **evalv** (double *r, unsigned int n)
- template<>
  void **evalv** (float *r, unsigned int n)

2.8.1 Detailed Description

template<typename Number> class AMP::materials::Property0D< Number >

Provides a base class for obtaining constants from a property.

Definition at line 100 of file Property.h.

2.8.2 Member Function Documentation

2.8.2.1 template<typename Number> virtual void AMP::materials::Property0D< Number >::evalv (Number * *result*, const unsigned int *n*) [virtual]

Each subclass will have a virtual evaluator for an array of input values.

The virtual function resolution mechanism overhead will be amortized over the vector evaluations. It is best to use a large value for n.

Parameters:

result array for results of evaluation of property

n number of evaluations to perform

The documentation for this class was generated from the following file:

- Property.h

2.9 AMP::materials::Property1D< Number > Class Template Reference

Provides a base class for functions of one variable.

`#include <Property.h>`

Inherits AMP::materials::PropertyBase< Number >.

Public Member Functions

- **Property1D** (const PropertySpec< Number > spec)
- virtual void evalv (Number *result, const Number *x, const unsigned int n)

  Each subclass will have a virtual evaluator for an array of input values.

- template<>
  void **evalv** (double *r, const double *x, unsigned int n)
- template<>
  void **evalv** (float *r, const float *x, unsigned int n)

2.9.1 Detailed Description

template<typename Number> class AMP::materials::Property1D< Number >

Provides a base class for functions of one variable.

Definition at line 127 of file Property.h.

2.9.2 Member Function Documentation

2.9.2.1 template<typename Number> virtual void AMP::materials::Property1D< Number >::evalv (Number * *result*, const Number * *x*, const unsigned int *n*) [virtual]

Each subclass will have a virtual evaluator for an array of input values.

The virtual function resolution mechanism overhead will be amortized over the vector evaluations. It is best to use a large value for n.

Parameters:

result array for results of evaluation of property

x input arguments to evaluator

n number of evaluations to perform

The documentation for this class was generated from the following file:

- Property.h

2.10 AMP::materials::Property2D< Number > Class Template Reference

Provides a base class for functions of two variables.

`#include <Property.h>`

Inherits AMP::materials::PropertyBase< Number >.

Public Member Functions

- **Property2D** (const PropertySpec< Number > spec)
- virtual void evalv (Number *result, const Number *x, const Number *y, const unsigned int n)

  Each subclass will have a virtual evaluator for an array of input values.

- template<>
  void **evalv** (double *r, const double *x, const double *y, unsigned int n)
- template<>
  void **evalv** (float *r, const float *x, const float *y, unsigned int n)

2.10.1 Detailed Description

template<typename Number> class AMP::materials::Property2D< Number >

Provides a base class for functions of two variables.

Definition at line 155 of file Property.h.

2.10.2 Member Function Documentation

2.10.2.1 template<typename Number> virtual void AMP::materials::Property2D< Number >::evalv (Number * *result*, const Number * *x*, const Number * *y*, const unsigned int *n*) [virtual]

Each subclass will have a virtual evaluator for an array of input values.

The virtual function resolution mechanism overhead will be amortized over the vector evaluations. It is best to use a large value for n.

Parameters:

result array for results of evaluation of property

first argument to evaluator

second argument to evaluator

n number of evaluations to perform

The documentation for this class was generated from the following file:

- Property.h

2.11 AMP::materials::PropertyBase< Number > Class Template Reference

Base class for all material properties.

`#include <Property.h>`

Inherited by AMP::materials::Property0D< Number >, AMP::materials::Property1D< Number >, and AMP::materials::Property2D< Number >.

Public Member Functions

- PropertyType **get_type** ()
- std::string **get_name** ()

- std::string **get_source** ()
- std::valarray< Number > **get_parameters** ()
- **PropertyBase** (const PropertySpec< Number > &spec)

2.11.1 Detailed Description

template<typename Number> class AMP::materials::PropertyBase< Number >

Base class for all material properties.

Definition at line 73 of file Property.h.

The documentation for this class was generated from the following file:

- Property.h

2.12 AMP::materials::PropertySpec< Number > Class Template Reference

Property Specification.

`#include <Property.h>`

Public Member Functions

- **PropertySpec** (const PropertyType type=None, const std::string name=std::string("none"), const std::string source=std::string("unknown"), const Number *params=NULL, const unsigned int nparams=0)

Public Attributes

- const PropertyType **d_type**
- const std::string **d_name**
- const std::string **d_source**
- const std::valarray< Number > **d_params**
- const unsigned int **d_nparams**

2.12.1 Detailed Description

template<typename Number> class AMP::materials::PropertySpec< Number >

Property Specification.

Collects a variety of initialization information in one place for use in Property constuctors.

Definition at line 43 of file Property.h.

The documentation for this class was generated from the following file:

- Property.h

2.13 AMP::materials::Undefined Class Reference

Default classes to tell the client programmer they made a boo-boo.

`#include <Material.h>`

Inherited by AMP::materials::UndefinedD, AMP::materials::UndefinedPR, and AMP::materials::UndefinedTC.

Public Member Functions

- double **eval** ()
- double **eval** (double a)
- double **eval** (double a, double b)
- void **evalv** (double *r, const unsigned int n)
- void **evalv** (double *r, const double *a0, const unsigned int n)
- void **evalv** (double *r, const double *a0, const double *a1, const unsigned int n)

2.13.1 Detailed Description

Default classes to tell the client programmer they made a boo-boo.

Definition at line 16 of file Material.h.

The documentation for this class was generated from the following file:

- Material.h

2.14 AMP::materials::UndefinedMaterial Class Reference

Undefined material.

`#include <Material.h>`

Inherits AMP::materials::MaterialBase< AMP::materials::DefaultTraits >.

2.14.1 Detailed Description

Undefined material.

Definition at line 78 of file Material.h.

The documentation for this class was generated from the following file:

- Material.h

Index

operators

amp-0_0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:28:03 2010

Contents

1 Class Documentation

1.1 AMP::ColumnOperator Class Reference

A class for representing a composite (nonlinear) operator, F=(F1, F2, F3, .

`#include <ColumnOperator.h>`

Inherits AMP::Operator.

Public Member Functions

- **ColumnOperator** (const boost::shared_ptr< OperatorParameters > &params)
- virtual void apply (const Vector::shared_ptr &f, const Vector::shared_ptr &u, Vector::shared_ptr &r, const double a=-1.0, const double b=1.0)

  A default implementation has been provided.

- boost::shared_ptr< OperatorParameters > getJacobianParameters (const Vector::shared_ptr &u)

  A function for computing the information necessary to construct the jacobian.

- void reset (const boost::shared_ptr< OperatorParameters > &params)

  This function is useful for re-initializing an operator.

- virtual void **appendRow** (boost::shared_ptr< Operator > op)

Protected Attributes

- std::vector< boost::shared_ptr< Operator > > **d_Operators**

1.1.1 Detailed Description

A class for representing a composite (nonlinear) operator, F=(F1, F2, F3, .

., Fk), where each of F1,.., Fk are (nonlinear) operators. The user is expected to have created and initialized the operators F1,.., Fk

Definition at line 18 of file ColumnOperator.h.

1.1.2 Member Function Documentation

1.1.2.1 void AMP::ColumnOperator::apply (const Vector::shared_ptr & *f*, const Vector::shared_ptr & *u*, Vector::shared_ptr & *r*, const double *a* = `-1.0`, const double *b* = `1.0`) `[virtual]`

A default implementation has been provided.

This can be overriden by the child classes if necessary.

Parameters:

f,: rhs vector for A(u)=f, this may be a null pointer if f=0.

u,: input vector for u

r,: output vector containing the result: r = b∗f+a∗A(u)

Implements AMP::Operator.

Definition at line 9 of file ColumnOperator.cc.

1.1.2.2 boost::shared_ptr< OperatorParameters > AMP::ColumnOperator::getJacobianParameters (const Vector::shared_ptr & *u*)

A function for computing the information necessary to construct the jacobian.

Parameters:

u The solution vector that is used to construct the jacobian

Returns:

The parameters required to construct the jacobian.

Definition at line 21 of file ColumnOperator.cc.

References AMP::ColumnOperatorParameters::d_OperatorParameters.

1.1.2.3 void AMP::ColumnOperator::reset (const boost::shared_ptr< OperatorParameters > & *params*) [virtual]

This function is useful for re-initializing an operator.

Parameters:

params parameter object containing parameters to change

Reimplemented from AMP::Operator.

Definition at line 41 of file ColumnOperator.cc.

The documentation for this class was generated from the following files:

- ColumnOperator.h
- ColumnOperator.cc

1.2 AMP::ColumnOperatorParameters Class Reference

A class that encapsulates the parameters required to construct the composite Operator operator.

`#include <ColumnOperatorParameters.h>`

Inherits AMP::OperatorParameters.

Public Member Functions

- **ColumnOperatorParameters** (const boost::shared_ptr< AMP::Database > &db)

Public Attributes

- std::vector< boost::shared_ptr< OperatorParameters > > **d_OperatorParameters**

1.2.1 Detailed Description

A class that encapsulates the parameters required to construct the composite Operator operator.

See also:

ColumnOperator

Definition at line 19 of file ColumnOperatorParameters.h.

The documentation for this class was generated from the following file:

- ColumnOperatorParameters.h

1.3 AMP::LinearElasticityOperator Class Reference

A class representing the linear elasticity operator with homogeneous and isotropic material properties.

`#include <LinearElasticityOperator.h>`

Inherits AMP::LinearOperator.

Public Member Functions

- **LinearElasticityOperator** (const boost::shared_ptr< LinearElasticityParameters > &params)
- void reset (const boost::shared_ptr< OperatorParameters > &params)

  A function to reinitialize the object.

1.3.1 Detailed Description

A class representing the linear elasticity operator with homogeneous and isotropic material properties.

Definition at line 18 of file LinearElasticityOperator.h.

1.3.2 Member Function Documentation

1.3.2.1 void AMP::LinearElasticityOperator::reset (const boost::shared_ptr< OperatorParameters > & *params*) [virtual]

A function to reinitialize the object.

end for qp

Reimplemented from AMP::Operator.

Definition at line 40 of file LinearElasticityOperator.cc.

References AMP::OperatorParameters::d_db, and AMP::LinearElasticityParameters::meshPtr.

The documentation for this class was generated from the following files:

- LinearElasticityOperator.h
- LinearElasticityOperator.cc

1.4 AMP::LinearElasticityParameters Class Reference

A class that encapsulates the parameters that are required for constructing the linear elasticity operator.

`#include <LinearElasticityParameters.h>`

Inherits AMP::OperatorParameters.

Public Member Functions

- **LinearElasticityParameters** (const boost::shared_ptr< AMP::Database > &db)

Public Attributes

- Mesh * **meshPtr**

1.4.1 Detailed Description

A class that encapsulates the parameters that are required for constructing the linear elasticity operator.

Definition at line 18 of file LinearElasticityParameters.h.

The documentation for this class was generated from the following file:

- LinearElasticityParameters.h

1.5 AMP::LinearFEOperator Class Reference

A class for representing a linear finite element operator.

`#include <LinearFEOperator.h>`

Inherits AMP::LinearOperator.

Inherited by AMP::DiffusionLinearFEOperator, and AMP::MechanicsLinearFEOperator.

Public Member Functions

- **LinearFEOperator** (const boost::shared_ptr< AssemblyParameters > &params)
- virtual void **preAssembly** ()
- virtual void **postAssembly** ()
- virtual void **preElementOperation** (const MeshManager::Adapter::Element &elem, const DOFMap::shared_ptr &input_dof_map, const DOFMap::shared_ptr &output_dof_map)
- virtual void **postElementOperation** ()
- void reset (const boost::shared_ptr< OperatorParameters > &params)

  This function is useful for re-initializing an operator.

Protected Attributes

- boost::shared_ptr< ElementOperation > **d_elemOp**

1.5.1 Detailed Description

A class for representing a linear finite element operator.

Definition at line 21 of file LinearFEOperator.h.

1.5.2 Member Function Documentation

1.5.2.1 void AMP::LinearFEOperator::reset (const boost::shared_ptr< OperatorParameters > & *params*) [virtual]

This function is useful for re-initializing an operator.

Parameters:

params parameter object containing parameters to change

Reimplemented from AMP::Operator.

Definition at line 7 of file LinearFEOperator.cc.

References AMP::Operator::getInputVariable(), and AMP::Operator::getOutputVariable().

The documentation for this class was generated from the following files:

- LinearFEOperator.h
- LinearFEOperator.cc

1.6 AMP::LinearOperator Class Reference

An abstract base class for representing a linear operator.

`#include <LinearOperator.h>`

Inherits AMP::Operator.

Inherited by AMP::LinearElasticityOperator, AMP::LinearFEOperator, AMP::MassMatrix, AMP::SmallStrainPlasticJacobian, and AMP::ThermalJacobian.

Public Member Functions

- **LinearOperator** (const boost::shared_ptr< OperatorParameters > &params)
- virtual void apply (const Vector::shared_ptr &f, const Vector::shared_ptr &u, Vector::shared_ptr &r, const double a=-1.0, const double b=1.0)

  A default implementation has been provided for Matrix-based implementations.

- const boost::shared_ptr< Matrix > & getMatrix ()

  Returns the matrix representation of this linear operator.

- void **setMatrix** (const boost::shared_ptr< Matrix > &in_mat)

Protected Attributes

- boost::shared_ptr< Matrix > **d_matrix**

1.6.1 Detailed Description

An abstract base class for representing a linear operator.

Definition at line 16 of file LinearOperator.h.

1.6.2 Member Function Documentation

1.6.2.1 void AMP::LinearOperator::apply (const Vector::shared_ptr & *f*, const Vector::shared_ptr & *u*, Vector::shared_ptr & *r*, const double *a* = `-1.0`, const double *b* = `1.0`) `[virtual]`

A default implementation has been provided for Matrix-based implementations.

This can be overriden by the child classes if necessary.

Parameters:

f,: rhs vector for A(u)=f, this may be a null pointer if f=0.

u,: input vector for u

r,: output vector containing the result: r = b∗f+a∗A(u)

Implements AMP::Operator.

Definition at line 8 of file LinearOperator.cc.

References AMP::Operator::getInputVariable(), and AMP::Operator::getOutputVariable().

The documentation for this class was generated from the following files:

- LinearOperator.h
- LinearOperator.cc

1.7 AMP::MassMatrix Class Reference

Class MassMatrix is an abstract base class for representing a operator which may be linear or nonlinear.

`#include <MassMatrix.h>`

Inherits AMP::LinearOperator.

Public Member Functions

- **MassMatrix** (boost::shared_ptr< MassMatrixParameters > params)
- void reset (const boost::shared_ptr< OperatorParameters > &params)

  This function is useful for re-initializing an operator.

1.7.1 Detailed Description

Class MassMatrix is an abstract base class for representing a operator which may be linear or nonlinear.

Concrete implementations must include an implementation of the apply() function. The constructor for the class takes a pointer to a OperatorParameters object.

Definition at line 19 of file MassMatrix.h.

1.7.2 Member Function Documentation

1.7.2.1 void AMP::MassMatrix::reset (const boost::shared_ptr< OperatorParameters > & *params*) [virtual]

This function is useful for re-initializing an operator.

Parameters:

params parameter object containing parameters to change

Reimplemented from AMP::Operator.

Definition at line 37 of file MassMatrix.cc.

The documentation for this class was generated from the following files:

- MassMatrix.h
- MassMatrix.cc

1.8 AMP::NeutronicsSource Class Reference

A class for representing the neutronics source operator.

`#include <NeutronicsSource.h>`

Inherits AMP::Operator.

Public Member Functions

- **NeutronicsSource** (boost::shared_ptr< NeutronicsSourceParameters > parameters)
- virtual ~NeutronicsSource ()

  Empty destructor for NeutronicsSource.

- virtual void initialize (boost::shared_ptr< NeutronicsSourceParameters > parameters)

  Initialize state of NeutronicsSource package.

- void printClassData (std::ostream &os) const

  Print out all members of integrator instance to given output stream.

- void putToDatabase (boost::shared_ptr< AMP::Database > db)

  Write out state of object to given database.

- void apply (const boost::shared_ptr< Vector > &f, const boost::shared_ptr< Vector > &u, boost::shared_ptr< Vector > &r, const double a=1.0, const double b=0.0)

  The function that computes the residual.

- void reset (const boost::shared_ptr< OperatorParameters > &parameters)

  A function to reinitialize this object.

Protected Member Functions

- void **getFromInput** (boost::shared_ptr< AMP::Database > db)

Protected Attributes

- boost::shared_ptr< AMP::MeshUtils > **d_MeshUtils**
- boost::shared_ptr< AMP::Database > **d_db**
- boost::shared_ptr< std::vector< double > > **d_SpecificPower**
- int **d_numTimeSteps**
- double **d_time**
- std::vector< double > **d_power**
- std::vector< double > **d_timeSteps**

1.8.1 Detailed Description

A class for representing the neutronics source operator.

Definition at line 24 of file NeutronicsSource.h.

1.8.2 Member Function Documentation

1.8.2.1 void AMP::NeutronicsSource::putToDatabase (boost::shared_ptr< AMP::Database > *db*)

Write out state of object to given database.

When assertion checking is active, the database pointer must be non-null.

Definition at line 111 of file NeutronicsSource.cc.

1.8.2.2 void AMP::NeutronicsSource::apply (const boost::shared_ptr< Vector > & *f*, const boost::shared_ptr< Vector > & *u*, boost::shared_ptr< Vector > & *r*, const double *a* = `1.0`, const double *b* = `0.0`)

The function that computes the residual.

Parameters:

f,: rhs vector for A(u)=f, this may be a null pointer if f=0.

u,: multivector of the state.

r,: specific power in Watts per gram The result of apply is r = b∗f+a∗A(u)

Definition at line 162 of file NeutronicsSource.cc.

The documentation for this class was generated from the following files:

- NeutronicsSource.h
- NeutronicsSource.cc

1.9 AMP::NeutronicsSourceParameters Class Reference

A class for encapsulating the parameters that are required for constructing the neutronics source operator.

`#include <NeutronicsSourceParameters.h>`

Inherits AMP::OperatorParameters.

Public Member Functions

- **NeutronicsSourceParameters** (const boost::shared_ptr< AMP::Database > &db)

Public Attributes

- boost::shared_ptr< AMP::MeshUtils > **d_MeshUtils**

1.9.1 Detailed Description

A class for encapsulating the parameters that are required for constructing the neutronics source operator.

See also:

NeutronicsSource

Definition at line 21 of file NeutronicsSourceParameters.h.

The documentation for this class was generated from the following file:

- NeutronicsSourceParameters.h

1.10 AMP::NonlinearFEOperator Class Reference

A class for representing a nonlinear finite element operator.

`#include <NonlinearFEOperator.h>`

Inherits AMP::Operator.

Public Member Functions

- **NonlinearFEOperator** (const boost::shared_ptr< AssemblyParameters > &params)
- virtual void **preElementOperation** ()
- virtual void **postElementOperation** ()
- virtual void **preAssembly** ()
- virtual void **postAssembly** ()
- virtual void apply (const boost::shared_ptr< Vector > &f, const boost::shared_ptr< Vector > &u, boost::shared_ptr< Vector > &r, const double a=-1.0, const double b=1.0)

  The function that computes the residual.

Protected Attributes

- boost::shared_ptr< ElementOperation > **d_elemOp**

1.10.1 Detailed Description

A class for representing a nonlinear finite element operator.

Definition at line 18 of file NonlinearFEOperator.h.

1.10.2 Member Function Documentation

1.10.2.1 void AMP::NonlinearFEOperator::apply (const boost::shared_ptr< Vector > & *f*, const boost::shared_ptr< Vector > & *u*, boost::shared_ptr< Vector > & *r*, const double *a* = `-1.0`, const double *b* = `1.0`) `[virtual]`

The function that computes the residual.

A default implementation is provided, which can be overidden by the derived classes.

Parameters:

f,: rhs vector for A(u)=f, this may be a null pointer if f=0.

u,: input vector for u.

r,: output vector The result of apply is r = b∗f+a∗A(u)

Definition at line 7 of file NonlinearFEOperator.cc.

The documentation for this class was generated from the following files:

- NonlinearFEOperator.h
- NonlinearFEOperator.cc

1.11 AMP::Operator Class Reference

Class Operator is an abstract base class for representing a discrete operator which may be linear or nonlinear.

`#include <Operator.h>`

Inherited by AMP::ColumnOperator, AMP::CompositionOperator, AMP::DirichletMatrixCorrection, AMP::DirichletVectorCorrection, AMP::LinearOperator, AMP::NeutronicsSource, AMP::NonlinearFEOperator, AMP::SmallStrainPlasticResidual, and AMP::ThermalResidual.

Public Types

- typedef boost::shared_ptr< Operator > **shared_ptr**

Public Member Functions

- Operator (const boost::shared_ptr< OperatorParameters > &params)

  Constructor.

- virtual ~Operator ()

  Destructor.

- virtual void reset (const boost::shared_ptr< OperatorParameters > &params)

  This function is useful for re-initializing an operator.

- virtual void setDebugPrintInfoLevel (int print_level)

  Specify level of diagnostic information printed during iterations.

- virtual void apply (const Vector::shared_ptr &f, const Vector::shared_ptr &u, Vector::shared_ptr &r, const double a=-1.0, const double b=1.0)=0

  apply() takes in a AMP vector u which may be defined over a subset of the levels in the hierarchy and returns A(u) in f where A may be a time dependent, nonlinear or linear operator.

- virtual boost::shared_ptr< OperatorParameters > getJacobianParameters (const boost::shared_ptr< Vector > &)

  This function returns a OperatorParameters object constructed by the operator which contains parameters from which the Jacobian or portions of the Jacobian required by solvers and preconditioners can be constructed.

- Variable::shared_ptr getInputVariable ()

  This function returns the variable description of the type of variable the operator acts on.

- Variable::shared_ptr getOutputVariable ()

  This function returns the variable description of the type of variable the operator acts on.

- void **setInputVariable** (Variable::shared_ptr var)
- void **setOutputVariable** (Variable::shared_ptr var)
- virtual void **checkVariable** ()

Protected Member Functions

- void **computeResidual** (const Vector::shared_ptr &f, Vector::shared_ptr &r, const double a, const double b)
- void **getFromInput** (const boost::shared_ptr< AMP::Database > &db)

Protected Attributes

- int **d_iDebugPrintInfoLevel**
- int **d_iObject_id**
- Variable::shared_ptr **d_InputVariable**
- Variable::shared_ptr **d_OutputVariable**
- MeshManager::Adapter::shared_ptr **d_MeshAdapter**

Static Protected Attributes

- static int **d_iInstance_id** = 0

1.11.1 Detailed Description

Class Operator is an abstract base class for representing a discrete operator which may be linear or nonlinear.

Concrete implementations must include an implementation of the apply() function. The constructor for the class takes a pointer to a OperatorParameters object.

Definition at line 30 of file Operator.h.

1.11.2 Member Function Documentation

1.11.2.1 void AMP::Operator::reset (const boost::shared_ptr< OperatorParameters > & *params*) [virtual]

This function is useful for re-initializing an operator.

Parameters:

params parameter object containing parameters to change

Reimplemented in AMP::ColumnOperator, AMP::LinearElasticityOperator, AMP::LinearFEOperator, AMP::MassMatrix, AMP::NeutronicsSource, AMP::SmallStrainPlasticJacobian, AMP::SmallStrainPlasticResidual, AMP::ThermalJacobian, and AMP::ThermalResidual.

Definition at line 48 of file Operator.cc.

1.11.2.2 virtual void AMP::Operator::setDebugPrintInfoLevel (int *print_level*) [inline, virtual]

Specify level of diagnostic information printed during iterations.

Parameters:

print_level zero prints none or minimal information, higher numbers provide increasingly verbose debugging information.

Definition at line 59 of file Operator.h.

1.11.2.3 virtual void AMP::Operator::apply (const Vector::shared_ptr & *f*, const Vector::shared_ptr & *u*, Vector::shared_ptr & *r*, const double *a* = -1.0, const double *b* = 1.0) [pure virtual]

apply() takes in a AMP vector u which may be defined over a subset of the levels in the hierarchy and returns A(u) in f where A may be a time dependent, nonlinear or linear operator.

A concrete implementation has to be provided in the derived operators for the TimeIntegrators to work with these operators.

Parameters:

f,: rhs vector for A(u)=f, this may be a null pointer if f=0. The result of apply is r = b∗f+a∗A(u)

f,: input vector for rhs

u,: input vector for u

r,: output vector containing A(u)

Implemented in AMP::ColumnOperator, and AMP::LinearOperator.

1.11.2.4 virtual boost::shared_ptr<OperatorParameters> AMP::Operator::getJacobianParameters (const boost::shared_ptr< Vector > &) [inline, virtual]

This function returns a OperatorParameters object constructed by the operator which contains parameters from which the Jacobian or portions of the Jacobian required by solvers and preconditioners can be constructed.

Returning a parameter object instead of the Jacobian itself is meant to give users more flexibility.

Reimplemented in AMP::SmallStrainPlasticResidual, and AMP::ThermalResidual.

Definition at line 85 of file Operator.h.

1.11.2.5 Variable::shared_ptr AMP::Operator::getInputVariable ()

`[inline]`

This function returns the variable description of the type of variable the operator acts on.

For instance, A:x->y, this operation returns a description of x.

Definition at line 95 of file Operator.h.

Referenced by AMP::ThermalResidual::apply(), AMP::LinearOperator::apply(), AMP::ThermalResidual::getJacobianParameters(), AMP::ThermalJacobian::reset(), and AMP::LinearFEOperator::reset().

1.11.2.6 Variable::shared_ptr AMP::Operator::getOutputVariable ()

`[inline]`

This function returns the variable description of the type of variable the operator acts on.

For instance, A:x->y, this operation returns a description of y. If A is "square," x and y should be the same.

Definition at line 105 of file Operator.h.

Referenced by AMP::LinearOperator::apply(), and AMP::LinearFEOperator::reset().

The documentation for this class was generated from the following files:

- Operator.h
- Operator.cc

1.12 AMP::OperatorParameters Class Reference

OperatorParameters encapsulates parameters used to initialize level operators.

`#include <MassMatrixParameters.h>`

Inherited by AMP::AssemblyParameters, AMP::ColumnOperatorParameters, AMP::CompositionOperatorParameters, AMP::DirichletMatrixCorrectionParameters, AMP::DirichletVectorCorrectionParameters, AMP::LinearElasticityParameters, AMP::MassMatrixParameters, AMP::NeutronicsSourceParameters, AMP::PlasticJacobianParameters, AMP::PlasticResidualParameters, AMP::ThermalJacobianParameters, and AMP::ThermalResidualParameters.

Public Member Functions

- OperatorParameters (const boost::shared_ptr< AMP::Database > &db)

  Construct and initialize a parameter list according to input data.

- virtual ~OperatorParameters ()

  Destructor.

Public Attributes

- boost::shared_ptr< AMP::Database > d_db

  Database object which needs to be initialized specific to the solver.

- Variable::shared_ptr **d_InputVariable**
- Variable::shared_ptr **d_OutputVariable**
- MeshManager::Adapter::shared_ptr **d_MeshAdapter**

1.12.1 Detailed Description

OperatorParameters encapsulates parameters used to initialize level operators.

OperatorParameters encapsulates parameters used to initialize operators.

It is an abstract base class.

Definition at line 23 of file OperatorParameters.h.

1.12.2 Constructor & Destructor Documentation

1.12.2.1 AMP::OperatorParameters::OperatorParameters (const boost::shared_ptr< AMP::Database > & *db*) [inline]

Construct and initialize a parameter list according to input data.

Guess what the required and optional keywords are.

Definition at line 30 of file OperatorParameters.h.

1.12.3 Member Data Documentation

1.12.3.1 boost::shared_ptr<AMP::Database> AMP::OperatorParameters::d_db

Database object which needs to be initialized specific to the solver.

Documentation for parameters required by each solver can be found in the documentation for the solver.

Definition at line 43 of file OperatorParameters.h.

Referenced by AMP::LinearElasticityOperator::reset().

The documentation for this class was generated from the following file:

- OperatorParameters.h

1.13 AMP::PlasticJacobianParameters Class Reference

A class that encapsulates the parameters required to construct the jacobian operator for small strain plasticity.

`#include <PlasticJacobianParameters.h>`

Inherits AMP::OperatorParameters.

Public Member Functions

- **PlasticJacobianParameters** (const boost::shared_ptr< AMP::Database > &db)

Public Attributes

- boost::shared_ptr< AMP::MeshUtils > **d_MeshUtils**
- std::vector< std::vector< double > > d_ConsistentTangent

  The consistent tangent values for each gauss point.

1.13.1 Detailed Description

A class that encapsulates the parameters required to construct the jacobian operator for small strain plasticity.

See also:

SmallStrainPlasticJacobian

Definition at line 22 of file PlasticJacobianParameters.h.

The documentation for this class was generated from the following file:

- PlasticJacobianParameters.h

1.14 AMP::PlasticResidualParameters Class Reference

A class for encapsulating the parameters that are required for constructing the plasticity residual operator.

`#include <PlasticResidualParameters.h>`

Inherits AMP::OperatorParameters.

Public Member Functions

- **PlasticResidualParameters** (const boost::shared_ptr< AMP::Database > &db)

Public Attributes

- boost::shared_ptr< AMP::MeshUtils > **d_MeshUtils**
- boost::shared_ptr< Vector > **d_InitDisp**

1.14.1 Detailed Description

A class for encapsulating the parameters that are required for constructing the plasticity residual operator.

See also:

SmallStrainPlasticResidual

Definition at line 21 of file PlasticResidualParameters.h.

The documentation for this class was generated from the following file:

- PlasticResidualParameters.h

1.15 AMP::SmallStrainPlasticJacobian Class Reference

A class for representing the jacobian operator for small strain plasticity.

`#include <SmallStrainPlasticJacobian.h>`

Inherits AMP::LinearOperator.

Public Member Functions

- SmallStrainPlasticJacobian (const boost::shared_ptr< PlasticJacobianParameters > &params)

  params must be convertible to a pointer of type PlasticJacobianParameters

- void reset (const boost::shared_ptr< OperatorParameters > &params)

  params must be convertible to a pointer of type PlasticJacobianParameters

1.15.1 Detailed Description

A class for representing the jacobian operator for small strain plasticity.

Definition at line 16 of file SmallStrainPlasticJacobian.h.

The documentation for this class was generated from the following files:

- SmallStrainPlasticJacobian.h
- SmallStrainPlasticJacobian.cc

1.16 AMP::SmallStrainPlasticResidual Class Reference

A class for representing the residual operator for small strain plasticity.

`#include <SmallStrainPlasticResidual.h>`

Inherits AMP::Operator.

Public Member Functions

- **SmallStrainPlasticResidual** (const boost::shared_ptr< PlasticResidualParameters > &params)
- void apply (const boost::shared_ptr< Vector > &f, const boost::shared_ptr< Vector > &u, boost::shared_ptr< Vector > &r, const double a=-1.0, const double b=1.0)

  The function that computes the residual.

- void reset (const boost::shared_ptr< OperatorParameters > &params)

  A function to reinitialize this object.

- boost::shared_ptr< OperatorParameters > getJacobianParameters (const boost::shared_ptr< Vector > &u)

  A function for computing the information necessary to construct the jacobian.

Protected Member Functions

- void bodyForce (const Point &pt, double force[3])

  Evaluates the body force at the specified coordinate.

- int Von_Mises_Radial_Return (double E, double Nu, double H, double Sig0, double *stre_np1, double *stre_n, double *stra_np1, double *stra_n, double *ystre_np1, double ystre_n, double *eph_bar_plas_np1, double eph_bar_plas_n, double *lambda, int *el_or_pl)

This is the RADIAL RETURN for Von-Mises plasticity models.

- void Von_Mises_Consistent_Tangent (double E, double Nu, double H, double Sig0, double *stre_np1, double ystre_np1, double eph_bar_plas_np1, double lambda, int el_or_pl, double d[6][6])

  This function calculates the Consistent Tangent for the Von Mises plasticity model.

Protected Attributes

- std::vector< double > **d_InitStressXX**
- std::vector< double > **d_InitStressYY**
- std::vector< double > **d_InitStressZZ**
- std::vector< double > **d_InitStressYZ**
- std::vector< double > **d_InitStressZX**
- std::vector< double > **d_InitStressXY**
- std::vector< double > **d_InitStrainXX**
- std::vector< double > **d_InitStrainYY**
- std::vector< double > **d_InitStrainZZ**
- std::vector< double > **d_InitStrainYZ**
- std::vector< double > **d_InitStrainZX**
- std::vector< double > **d_InitStrainXY**
- std::vector< double > **d_InitYieldStress**
- std::vector< double > **d_InitEffPlasticStrain**
- boost::shared_ptr< AMP::MeshUtils > **d_MeshUtils**
- double **d_dE**
- double **d_dNu**
- double **d_dH**
- double **d_dSig0**
- double **d_dDensity**
- double **d_dGravity**
- boost::shared_ptr< AMP::Database > **d_db**
- bool **d_bInitialized**

1.16.1 Detailed Description

A class for representing the residual operator for small strain plasticity.

Currently uses the Von Mises plasticity model.

Definition at line 22 of file SmallStrainPlasticResidual.h.

1.16.2 Member Function Documentation

1.16.2.1 void AMP::SmallStrainPlasticResidual::apply (const boost::shared_ptr< Vector > & *f*, const boost::shared_ptr< Vector > & *u*, boost::shared_ptr< Vector > & *r*, const double *a* = -1.0, const double *b* = 1.0)

The function that computes the residual.

Parameters:

f,: rhs vector for A(u)=f, this may be a null pointer if f=0.

u,: input vector for u.

r,: output vector The result of apply is r = b∗f+a∗A(u)

end for qp

Definition at line 443 of file SmallStrainPlasticResidual.cc.

References bodyForce(), and Von_Mises_Radial_Return().

1.16.2.2 boost::shared_ptr< OperatorParameters > AMP::SmallStrainPlasticResidual::getJacobianParameters (const boost::shared_ptr< Vector > & *u*) [virtual]

A function for computing the information necessary to construct the jacobian.

Parameters:

u The solution vector that is used to construct the jacobian

Returns:

The parameters required to construct the jacobian. In this case, it is the list of consistent tangent values at each gauss point.

Reimplemented from AMP::Operator.

Definition at line 287 of file SmallStrainPlasticResidual.cc.

References Von_Mises_Consistent_Tangent(), and Von_Mises_Radial_Return().

1.16.2.3 void AMP::SmallStrainPlasticResidual::bodyForce (const Point & *pt*, double *force*[3]) [protected]

Evaluates the body force at the specified coordinate.

Currently, the only body force is due to gravity.

Definition at line 701 of file SmallStrainPlasticResidual.cc.

Referenced by apply().

1.16.2.4 int AMP::SmallStrainPlasticResidual::Von_Mises_Radial_Return (double *E*, double *Nu*, double *H*, double *Sig0*, double * *stre_np1*, double * *stre_n*, double * *stra_np1*, double * *stra_n*, double * *ystre_np1*, double *ystre_n*, double * *eph_bar_plas_np1*, double *eph_bar_plas_n*, double * *lambda*, int * *el_or_pl*) [protected]

This is the RADIAL RETURN for Von-Mises plasticity models.

Parameters:

stre_n Stress at n-th or previous time step (Input)

stra_n Strain at n-th or previous time step (Input)

stra_np1 Strain at current time step (Input)

ystre_n Yield Stress at the n-th time step (different than sig0) (Input)

eph_bar_plas_n Effective plastic strain at n-th time step (Input)

el_or_pl Keeps track of whether the Gauss point is in Elastic or Plastic range (Output)

stre_np1 Stress at (n+1)-th or current time step (Output)

ystre_np1 Yield Stress at the (n+1)-th time step (Output)

eph_bar_plas_n+1 Effective plastic strain at (n+1)-th time step (Output)

lambda Plastic multipler (sometime used in the Consistent Tangent evaluation) (Output)

Returns:

"1" if the code converges properly.

Definition at line 707 of file SmallStrainPlasticResidual.cc.

Referenced by apply(), getJacobianParameters(), and reset().

1.16.2.5 void AMP::SmallStrainPlasticResidual::Von_Mises_Consistent_Tangent (double *E*, double *Nu*, double *H*, double *Sig0*, double * *stre_np1*, double *ystre_np1*, double *eph_bar_plas_np1*, double *lambda*, int *el_or_pl*, double *d*[6][6]) [protected]

This function calculates the Consistent Tangent for the Von Mises plasticity model.

Parameters:

stre_np1 Stress at (n+1)-th or current time step (calculated from Radial Return algorithm) (Input)

ystre_np1 Yield Stress at the (n+1)-th time step (calculated from Radial Return algorithm) (Input)

eph_bar_plas_np1 Effective plastic strain at (n+1)-th time step (calculated from Radial Return algorithm) (Input)

lambda Plastic multipler (calculated from Radial Return algorithm) (Input)

el_or_pl Elastic/Plastic indicator at a Gauss point (Updated at the Radial Return algorithm) (Input)

d The Consistent Tangent Matrix (Output)

Definition at line 809 of file SmallStrainPlasticResidual.cc.

Referenced by getJacobianParameters().

The documentation for this class was generated from the following files:

- SmallStrainPlasticResidual.h
- SmallStrainPlasticResidual.cc

1.17 AMP::ThermalJacobian Class Reference

A class for representing the thermal jacobian operator.

`#include <ThermalJacobian.h>`

Inherits AMP::LinearOperator.

Public Types

- typedef ThermalJacobianParameters **Parameters**
- typedef NodalScalarVariable **OutputVariable**
- typedef NodalScalarVariable **InputVariable**

Public Member Functions

- ThermalJacobian (const boost::shared_ptr< ThermalJacobianParameters > &params)

  params must be convertible to a pointer of type ThermalJacobianParameters

- void reset (const boost::shared_ptr< OperatorParameters > &params)

  params must be convertible to a pointer of type ThermalJacobianParameters

1.17.1 Detailed Description

A class for representing the thermal jacobian operator.

Definition at line 19 of file ThermalJacobian.h.

The documentation for this class was generated from the following files:

- ThermalJacobian.h
- ThermalJacobian.cc

1.18 AMP::ThermalJacobianParameters Class Reference

A class that encapsulates the parameters required to construct the thermal jacobian operator.

`#include <ThermalJacobianParameters.h>`

Inherits AMP::OperatorParameters.

Public Member Functions

- **ThermalJacobianParameters** (const boost::shared_ptr< AMP::Database > &db)

Public Attributes

- boost::shared_ptr< std::vector< double > > d_ConductivityGauss

  The conductivity values for each gauss point.

- Vector::shared_ptr **d_ConductivityNodal**

1.18.1 Detailed Description

A class that encapsulates the parameters required to construct the thermal jacobian operator.

See also:

ThermalJacobian

Definition at line 22 of file ThermalJacobianParameters.h.

The documentation for this class was generated from the following file:

- ThermalJacobianParameters.h

1.19 AMP::ThermalResidual Class Reference

A class for representing the thermal residual operator.

`#include <ThermalResidual.h>`

Inherits AMP::Operator.

Public Types

- typedef ThermalJacobian **Jacobian**
- typedef ThermalResidualParameters **Parameters**
- typedef NodalScalarVariable **OutputVariable**
- typedef NodalScalarVariable **InputVariable**

Public Member Functions

- **ThermalResidual** (const boost::shared_ptr< ThermalResidualParameters > &params)
- void apply (const boost::shared_ptr< Vector > &f, const boost::shared_ptr< Vector > &u, boost::shared_ptr< Vector > &r, const double a=-1.0, const double b=1.0)

  The function that computes the residual.

- void reset (const boost::shared_ptr< OperatorParameters > &params)

  A function to reinitialize this object.

- boost::shared_ptr< OperatorParameters > getJacobianParameters (const boost::shared_ptr< Vector > &u)

  A function for computing the information necessary to construct the jacobian.

Static Public Member Functions

- static const char * **DBName** ()

Protected Attributes

- boost::shared_ptr< AMP::Database > **d_db**
- boost::shared_ptr< std::vector< double > > **d_ConductivityGauss**
- Vector::shared_ptr **d_ConductivityNodal**

1.19.1 Detailed Description

A class for representing the thermal residual operator.

Currently uses the Emperical relationfor the Conductivity.

Definition at line 23 of file ThermalResidual.h.

1.19.2 Member Function Documentation

1.19.2.1 void AMP::ThermalResidual::apply (const boost::shared_ptr< Vector > & *f*, const boost::shared_ptr< Vector > & *u*, boost::shared_ptr< Vector > & *r*, const double *a* = `-1.0`, const double *b* = `1.0`)

The function that computes the residual.

Parameters:

f,: rhs vector for A(u)=f, this may be a null pointer if f=0.

u,: input vector for u.

r,: output vector The result of apply is r = b∗f+a∗A(u)

end for qp

Definition at line 242 of file ThermalResidual.cc.

References AMP::Operator::getInputVariable().

1.19.2.2 boost::shared_ptr< OperatorParameters > AMP::ThermalResidual::getJacobianParameters (const boost::shared_ptr< Vector > & *u*) `[virtual]`

A function for computing the information necessary to construct the jacobian.

Parameters:

u The solution vector that is used to construct the jacobian

Returns:

The parameters required to construct the jacobian. In this case, it is the list of conductivity values at each gauss point.

Reimplemented from AMP::Operator.

Definition at line 58 of file ThermalResidual.cc.

References AMP::Operator::getInputVariable().

The documentation for this class was generated from the following files:

- ThermalResidual.h
- ThermalResidual.cc

1.20 AMP::ThermalResidualParameters Class Reference

A class for encapsulating the parameters that are required for constructing the thermal residual operator.

`#include <ThermalResidualParameters.h>`

Inherits AMP::OperatorParameters.

Public Member Functions

- **ThermalResidualParameters** (const boost::shared_ptr< AMP::Database > &db)

1.20.1 Detailed Description

A class for encapsulating the parameters that are required for constructing the thermal residual operator.

See also:

ThermalResidual

Definition at line 21 of file ThermalResidualParameters.h.

The documentation for this class was generated from the following file:

- ThermalResidualParameters.h

Index

solvers

amp-0_0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:28:04 2010

Contents

1 Class Documentation

1.1 AMP::NonlinearKrylovAcceleratorParameters Class Reference

Class NonlinearKrylovAcceleratorParameters provides a uniform mechanism to pass initialization parameters when constructing a Application.

`#include <NonlinearKrylovAcceleratorParameters.h>`

Inherits AMP::SolverStrategyParameters.

Public Member Functions

- NonlinearKrylovAcceleratorParameters ()

  Empty constructor.

- NonlinearKrylovAcceleratorParameters (const boost::shared_ptr< AMP::Database > &database)

  Construct and initialize a parameter list according to input data.

- virtual ~NonlinearKrylovAcceleratorParameters ()

  Destructor.

Public Attributes

- boost::shared_ptr< SolverStrategy > **d_pPreconditioner**
- boost::shared_ptr< **Vector** > **d_pvInitialGuess**

1.1.1 Detailed Description

Class NonlinearKrylovAcceleratorParameters provides a uniform mechanism to pass initialization parameters when constructing a Application.

Definition at line 22 of file NonlinearKrylovAcceleratorParameters.h.

1.1.2 Constructor & Destructor Documentation

1.1.2.1 AMP::NonlinearKrylovAcceleratorParameters::NonlinearKrylovAcceleratorParameters (const boost::shared_ptr< AMP::Database > & *database*)

Construct and initialize a parameter list according to input data.

See Application for a list of required and optional keywords.

Definition at line 10 of file NonlinearKrylovAcceleratorParameters.cc.

The documentation for this class was generated from the following files:

- NonlinearKrylovAcceleratorParameters.h
- NonlinearKrylovAcceleratorParameters.cc

1.2 AMP::SolverStrategy Class Reference

Class SolverStrategy is a base class for methods to solve problems on a SAMR hierarchy.

`#include <SolverStrategy.h>`

Inherited by AMP::NonlinearKrylovAccelerator, AMP::PetscKrylovSolver, AMP::PetscSNESSolver, and AMP::TrilinosMLSolver.

Public Member Functions

- **SolverStrategy** (boost::shared_ptr< SolverStrategyParameters > parameters)
- virtual void solve (boost::shared_ptr< **Vector** > f, boost::shared_ptr< **Vector** > u)=0

  Solve the system $A(u) = f$.

- virtual void initialize (boost::shared_ptr< SolverStrategyParameters > const parameters)

  Initialize the solution vector and potentially create internal vectors needed for solution.

- virtual void **setInitialGuess** (boost::shared_ptr< **Vector** > initialGuess)=0
- virtual void setConvergenceTolerance (const int max_iterations, const double max_error)

  Specify stopping criteria.

- virtual void setDebugPrintInfoLevel (int print_level)

  Specify level of diagnostic information printed during iterations.

- virtual int getIterations (void) const

  Return the number of iterations taken by the solver to converge.

- virtual void setZeroInitialGuess (bool use_zero_guess)

  Tells the solver to use an initial guess of zero and not try to copy an initial guess into the solution std::vector.

- virtual void registerOperator (const boost::shared_ptr< Operator > op)

  Register the operator that the solver will use during solves.

- virtual void resetOperator (const boost::shared_ptr< OperatorParameters > params)

  Resets the associated operator internally with new parameters if necessary.

- virtual void reset (boost::shared_ptr< SolverStrategyParameters > parameters)

  Resets the solver internally with new parameters if necessary.

- virtual boost::shared_ptr< Operator > **getOperator** (void)

Protected Member Functions

- void **getFromInput** (const boost::shared_ptr< AMP::Database > &db)

Protected Attributes

- int **d_iNumberIterations**
- double **d_dResidualNorm**
- int **d_iMaxIterations**

- double **d_dMaxRhs**
- double **d_dMaxError**
- int **d_iDebugPrintInfoLevel**
- bool **d_bUseZeroInitialGuess**
- int **d_iObjectId**
- boost::shared_ptr< AMP::Operator > **d_pOperator**

Static Protected Attributes

- static int **d_iInstanceId** = 0

1.2.1 Detailed Description

Class SolverStrategy is a base class for methods to solve problems on a SAMR hierarchy.

Definition at line 38 of file SolverStrategy.h.

1.2.2 Member Function Documentation

1.2.2.1 void AMP::SolverStrategy::resetOperator (const boost::shared_ptr< OperatorParameters > *params*) [virtual]

Resets the associated operator internally with new parameters if necessary.

Parameters:

parameters OperatorParameters object that is NULL by default

Definition at line 85 of file SolverStrategy.cc.

1.2.2.2 virtual void AMP::SolverStrategy::reset (boost::shared_ptr< SolverStrategyParameters > *parameters*) [inline, virtual]

Resets the solver internally with new parameters if necessary.

Parameters:

parameters SolverStrategyParameters object that is NULL by default

Definition at line 98 of file SolverStrategy.h.

The documentation for this class was generated from the following files:

- SolverStrategy.h
- SolverStrategy.cc

1.3 AMP::SolverStrategyParameters Class Reference

SolverStrategyParameters encapsulates parameters used to initialize SolverStrategy objects.

`#include <SolverStrategyParameters.h>`

Inherited by AMP::NonlinearKrylovAcceleratorParameters, AMP::PetscKrylovSolverParameters, and AMP::PetscSNESSolverParameters.

Public Member Functions

- SolverStrategyParameters ()
  Empty constructor.
- SolverStrategyParameters (const boost::shared_ptr< AMP::Database > &db)
  Construct and initialize a parameter list according to input data.
- virtual ~SolverStrategyParameters ()
  Destructor.

Public Attributes

- boost::shared_ptr< AMP::Database > d_db
  Pointer to database object which needs to be initialized specific to the solver.
- boost::shared_ptr< AMP::Operator > **d_pOperator**

1.3.1 Detailed Description

SolverStrategyParameters encapsulates parameters used to initialize SolverStrategy objects.

Definition at line 33 of file SolverStrategyParameters.h.

1.3.2 Constructor & Destructor Documentation

1.3.2.1 AMP::SolverStrategyParameters::SolverStrategyParameters (const boost::shared_ptr< AMP::Database > & *db*)

Construct and initialize a parameter list according to input data.

Guess what the required and optional keywords are.

Definition at line 11 of file SolverStrategyParameters.cc.

1.3.3 Member Data Documentation

1.3.3.1 boost::shared_ptr<AMP::Database> AMP::SolverStrategyParameters::d_db

Pointer to database object which needs to be initialized specific to the solver.

Documentation for parameters required by each solver can be found in the documentation for the solver.

Definition at line 57 of file SolverStrategyParameters.h.

The documentation for this class was generated from the following files:

- SolverStrategyParameters.h
- SolverStrategyParameters.cc

Index

time_integrators

amp-0_0_0

Generated by Doxygen 1.5.6

Wed Feb 24 16:28:05 2010

Contents

1 Overview of the time_integrators package

Version:

amp-0_0_0

Time_integrators package in amp

2 Class Documentation

2.1 AMP::BackwardEulerTimeIntegrator Class Reference

Class BackwardEulerTimeIntegrator is a concrete time integrator that implements the backward Euler method.

`#include <BackwardEulerTimeIntegrator.h>`

Inherits AMP::ImplicitTimeIntegrator.

Public Member Functions

- BackwardEulerTimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Constructor that accepts parameter list.

- ~BackwardEulerTimeIntegrator ()

  Destructor.

- void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize from parameter list.

- void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Resets the internal state of the time integrator as needed.

- double getInitialDt ()

  Specify initial time step.

- double getNextDt (const bool good_solution)

  Specify next time step to use.

- void setInitialGuess (const bool first_step, const double current_time, const double current_dt, const double old_dt)

  Set an initial guess for the time advanced solution.

- void updateSolution (void)

  Update state of the solution.

- bool checkNewSolution (void) const

  Check time advanced solution to determine whether it is acceptable.

Protected Member Functions

- void **initializeTimeOperator** (boost::shared_ptr< TimeIntegratorParameters > parameters)

2.1.1 Detailed Description

Class BackwardEulerTimeIntegrator is a concrete time integrator that implements the backward Euler method.

Definition at line 17 of file BackwardEulerTimeIntegrator.h.

2.1.2 Member Function Documentation

2.1.2.1 void AMP::BackwardEulerTimeIntegrator::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implements AMP::ImplicitTimeIntegrator.

Definition at line 66 of file BackwardEulerTimeIntegrator.cc.

2.1.2.2 bool AMP::BackwardEulerTimeIntegrator::checkNewSolution (void) const [virtual]

Check time advanced solution to determine whether it is acceptable.

Return true if the solution is acceptable; return false otherwise. The integer argument is the return code generated by the call to the nonlinear solver "solve" routine. The meaning of this value depends on the particular nonlinear solver in use and must be intepreted properly by the user-supplied solution checking routine.

Implements AMP::ImplicitTimeIntegrator.

Definition at line 135 of file BackwardEulerTimeIntegrator.cc.

The documentation for this class was generated from the following files:

- BackwardEulerTimeIntegrator.h
- BackwardEulerTimeIntegrator.cc

2.2 AMP::ExplicitEuler Class Reference

Class ExplicitEuler is a concrete time integrator that implements the explicit Runge-Kutta second order (RK2) method.

`#include <ExplicitEuler.h>`

Inherits AMP::TimeIntegrator.

Public Member Functions

- ExplicitEuler (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Constructor that accepts parameter list.

- ~ExplicitEuler ()

  Destructor.

- void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize from parameter list.

- void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Resets the internal state of the time integrator as needed.

- double getInitialDt ()

  Specify initial time step.

- double getNextDt (const bool good_solution)

  Specify next time step to use.

- bool checkNewSolution (void) const

  Determine whether time advanced solution is satisfactory.

- void updateSolution (void)

  Update state of the solution.

- int advanceSolution (const double dt, const bool first_step)

  Integrate entire patch hierarchy through the specified time increment.

2.2.1 Detailed Description

Class ExplicitEuler is a concrete time integrator that implements the explicit Runge-Kutta second order (RK2) method.

Definition at line 17 of file ExplicitEuler.h.

2.2.2 Member Function Documentation

2.2.2.1 void AMP::ExplicitEuler::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implements AMP::TimeIntegrator.

Definition at line 61 of file ExplicitEuler.cc.

2.2.2.2 int AMP::ExplicitEuler::advanceSolution (const double *dt*, const bool *first_step*) [virtual]

Integrate entire patch hierarchy through the specified time increment.

Integrate entire patch hierarchy through the specified time increment.

The boolean first_step argument is true when this is the very first call to the advance function or if the call occurs immediately after the hierarchy has changed due to re-gridding. Otherwise it is false. Note that, when the argument is true, the use of extrapolation to construct the initial guess for the advanced solution may not be possible.

Parameters:

dt Time step size

first_step Whether this is the first step after grid change

Returns:

value is the return code generated by the particular solver package in use

Implements AMP::TimeIntegrator.

Definition at line 88 of file ExplicitEuler.cc.

References AMP::TimeIntegrator::d_operator, and AMP::TimeIntegrator::stepsRemaining().

The documentation for this class was generated from the following files:

- ExplicitEuler.h
- ExplicitEuler.cc

2.3 AMP::IDATimeIntegrator Class Reference

Class IDATimeIntegrator is a concrete time integrator that implements the backward Euler method.

`#include <IDATimeIntegrator.h>`

Inherits AMP::TimeIntegrator.

Public Member Functions

- IDATimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Constructor that accepts parameter list.

- ~IDATimeIntegrator ()

  Destructor.

- void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize from parameter list.

- void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Resets the internal state of the time integrator as needed.

- double getInitialDt ()

  Specify initial time step.

- double getNextDt (const bool good_solution)

  Specify next time step to use.

- double getCurrentTime ()

  Return current integration time.

- void setInitialGuess (const bool first_step, const double current_time, const double current_dt, const double old_dt)

  Set an initial guess for the time advanced solution.

- int advanceSolution (const double dt, const bool first_step)

  Integrate entire patch hierarchy through the specified time increment.

- void updateSolution (void)

  Update state of the solution.

- bool checkNewSolution (void) const

  Check time advanced solution to determine whether it is acceptable.

- boost::shared_ptr< MassMatrix > **getMassMatrix** () const
- boost::shared_ptr< Operator > **getOperator** () const
- boost::shared_ptr< Vector > **getResidualVector** () const
- boost::shared_ptr< Vector > **getTempVec1** () const
- boost::shared_ptr< Vector > **getTempVec2** () const
- boost::shared_ptr< SolverStrategy > **getPreconditioner** (void)
- void * **getIDAMem** (void)

Public Attributes

- boost::shared_ptr< Vector > **d_residual**
- boost::shared_ptr< Vector > **d_temp_vec_1**
- boost::shared_ptr< Vector > **d_temp_vec_2**

2.3.1 Detailed Description

Class IDATimeIntegrator is a concrete time integrator that implements the backward Euler method.

Definition at line 45 of file IDATimeIntegrator.h.

2.3.2 Member Function Documentation

2.3.2.1 void AMP::IDATimeIntegrator::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implements AMP::TimeIntegrator.

Definition at line 144 of file IDATimeIntegrator.cc.

2.3.2.2 int AMP::IDATimeIntegrator::advanceSolution (const double *dt*, const bool *first_step*) [virtual]

Integrate entire patch hierarchy through the specified time increment.

Integrate entire patch hierarchy through the specified time increment.

The boolean first_step argument is true when this is the very first call to the advance function or if the call occurs immediately after the hierarchy has changed due to regridding. Otherwise it is false. Note that, when the argument is true, the use of extrapolation to construct the initial guess for the advanced solution may not be possible.

Parameters:

dt Time step size

first_step Whether this is the first step after grid change

Returns:

value is the return code generated by the particular solver package in use

Implements AMP::TimeIntegrator.

Definition at line 250 of file IDATimeIntegrator.cc.

2.3.2.3 bool AMP::IDATimeIntegrator::checkNewSolution (void) const [virtual]

Check time advanced solution to determine whether it is acceptable.

Return true if the solution is acceptable; return false otherwise. The integer argument is the return code generated by the call to the nonlinear solver "solve" routine. The meaning of this value depends on the particular nonlinear solver in use and must be intepreted properly by the user-supplied solution checking routine.

Implements AMP::TimeIntegrator.

Definition at line 310 of file IDATimeIntegrator.cc.

The documentation for this class was generated from the following files:

- IDATimeIntegrator.h
- IDATimeIntegrator.cc

2.4 AMP::IDATimeIntegratorParameters Class Reference

TimeIntegratorParameters is a base class for providing parameters for the TimeIntegrator's. The Database object contained must contain the following entries:.

`#include <IDATimeIntegratorParameters.h>`

Inherits AMP::TimeIntegratorParameters.

Public Member Functions

- **IDATimeIntegratorParameters** (const boost::shared_ptr< AMP::Database > db)

Public Attributes

- boost::shared_ptr< Vector > **d_ic_vector_prime**
- boost::shared_ptr< MassMatrix > **d_mass_matrix**
- boost::shared_ptr< SolverStrategy > **d_pPreconditioner**

2.4.1 Detailed Description

TimeIntegratorParameters is a base class for providing parameters for the TimeIntegrator's. The Database object contained must contain the following entries:.

Required input keys and data types:

Parameters:

initial_time double value for the initial simulation time.

final_time double value for the final simulation time.

max_integrator_steps integer value for the maximum number of timesteps allowed.

All input data items described above, except for initial_time, may be overwritten by new input values when continuing from restart.

Definition at line 35 of file IDATimeIntegratorParameters.h.

The documentation for this class was generated from the following files:

- IDATimeIntegratorParameters.h
- IDATimeIntegratorParameters.cc

2.5 AMP::ImplicitTimeIntegrator Class Reference

Manage implicit time integration.

`#include <ImplicitTimeIntegrator.h>`

Inherits AMP::TimeIntegrator.

Inherited by AMP::BackwardEulerTimeIntegrator.

Public Member Functions

- ImplicitTimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > parameters)

  The constructor for ImplicitTimeIntegrator initializes the default state of the integrator.

- virtual ~ImplicitTimeIntegrator ()

  Empty destructor for ImplicitTimeIntegrator.

- void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize state of time integrator.

- virtual void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)=0

  Resets the internal state of the time integrator as needed.

- virtual int advanceSolution (const double dt, const bool first_step)

  Integrate entire patch hierarchy through the specified time increment.

- virtual double getNextDt (const bool good_solution)=0

  Return time increment for next solution advance.

- virtual bool checkNewSolution (void) const =0

  Check time advanced solution to determine whether it is acceptable.

- virtual void updateSolution (void)=0

  Update solution (e.g., reset pointers for solution data, update dependent variables, etc.

- void printClassData (std::ostream &os) const

  Print out all members of integrator instance to given output stream.

- void putToDatabase (boost::shared_ptr< AMP::Database > db)

  Write out state of object to given database.

Protected Member Functions

- virtual void **initializeTimeOperator** (boost::shared_ptr< TimeIntegratorParameters > parameters)

Protected Attributes

- boost::shared_ptr< SolverStrategy > **d_solver**
- boost::shared_ptr< TimeOperatorParameters > **d_pTimeOperatorParameters**

2.5.1 Detailed Description

Manage implicit time integration.

Class ImplicitTimeIntegrator manages implicit time integration. It maintains references to a TimeDependentOperator and SolverStrategy objects. The TimeDependentOperator describe the implicit equations at each time step and the SolverStrategy solves the problem at each time step. the same time increment.

Initialization of an ImplicitTimeIntegrator object is performed via a combination of default parameters and values read from input. Data read from input is summarized as follows:

Required input keys and data types:

Parameters:

initial_time double value for the initial simulation time.

final_time double value for the final simulation time.

max_integrator_steps integer value for the maximum number of timesteps allowed.

All input data items described above, except for initial_time, may be overwritten by new input values when continuing from restart.

A sample input file entry might look like:

```
initial_time = 0.0
final_time   = 1.0
max_integrator_steps = 100
```

See also:

Operator
SolverStrategy

Definition at line 68 of file ImplicitTimeIntegrator.h.

2.5.2 Constructor & Destructor Documentation

2.5.2.1 AMP::ImplicitTimeIntegrator::ImplicitTimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > *parameters*)

The constructor for ImplicitTimeIntegrator initializes the default state of the integrator.

The integrator is configured with the concrete strategy objects in the argument list that provide operations related to the nonlinear solver and implicit equations to solve. Data members are initialized from the input and restart databases.

Note that no vectors are created in the constructor. Vectors are created and the nonlinear solver is initialized in the initialize() member function.

Definition at line 40 of file ImplicitTimeIntegrator.cc.

References initialize().

2.5.3 Member Function Documentation

2.5.3.1 void AMP::ImplicitTimeIntegrator::initialize (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Initialize state of time integrator.

This includes creating solution vector and initializing solver components.

Reimplemented from AMP::TimeIntegrator.

Reimplemented in AMP::BackwardEulerTimeIntegrator.

Definition at line 63 of file ImplicitTimeIntegrator.cc.

References AMP::TimeIntegrator::d_operator.

Referenced by ImplicitTimeIntegrator(), and AMP::BackwardEulerTimeIntegrator::initialize().

2.5.3.2 virtual void AMP::ImplicitTimeIntegrator::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [pure virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implements AMP::TimeIntegrator.

Implemented in AMP::BackwardEulerTimeIntegrator.

2.5.3.3 int AMP::ImplicitTimeIntegrator::advanceSolution (const double *dt*, const bool *first_step*) [virtual]

Integrate entire patch hierarchy through the specified time increment.

Integrate entire patch hierarchy through the specified time increment. The time advance assumes the use of a nonlinear solver to implicitly integrate the discrete equations. The integer return value is the return code generated by the particular solver package in use. It is the user's responsibility to interpret this code in a manner consistent with the solver she is using.

The boolean first_step argument is true when this is the very first call to the advance function or if the call occurs immediately after the hierarchy has changed due to re-gridding. Otherwise it is false. Note that, when the argument is true, the use of extrapolation to construct the initial guess for the advanced solution may not be possible.

Parameters:

dt Time step size

first_step Whether this is the first step after grid change

Returns:

value is the return code generated by the particular solver package in use

Implements AMP::TimeIntegrator.

Definition at line 103 of file ImplicitTimeIntegrator.cc.

References AMP::TimeIntegrator::d_operator, and AMP::TimeIntegrator::stepsRemaining().

2.5.3.4 virtual double AMP::ImplicitTimeIntegrator::getNextDt (const bool *good_solution*) [pure virtual]

Return time increment for next solution advance.

Timestep selection is generally based on whether the nonlinear solution iteration converged and, if so, whether the solution meets some user-defined criteria. This routine assumes that, before it is called, the routine checkNewSolution() was called. The boolean argument is the return value from that call. The integer argument is the return code generated by the nonlinear solver package that computed the solution.

Implements AMP::TimeIntegrator.

Implemented in AMP::BackwardEulerTimeIntegrator.

2.5.3.5 virtual bool AMP::ImplicitTimeIntegrator::checkNewSolution (void) const [pure virtual]

Check time advanced solution to determine whether it is acceptable.

Return true if the solution is acceptable; return false otherwise. The integer argument is the return code generated by the call to the nonlinear solver "solve" routine. The meaning of this value depends on the particular nonlinear solver in use and must be intepreted properly by the user-supplied solution checking routine.

Implements AMP::TimeIntegrator.

Implemented in AMP::BackwardEulerTimeIntegrator.

2.5.3.6 virtual void AMP::ImplicitTimeIntegrator::updateSolution (void) [pure virtual]

Update solution (e.g., reset pointers for solution data, update dependent variables, etc.

) after time advance. It is assumed that when this routine is invoked, an acceptable new solution has been computed. The double return value is the simulation time corresponding to the advanced solution.

Implements AMP::TimeIntegrator.

Implemented in AMP::BackwardEulerTimeIntegrator.

2.5.3.7 void AMP::ImplicitTimeIntegrator::putToDatabase (boost::shared_ptr< AMP::Database > *db*)

Write out state of object to given database.

When assertion checking is active, the database pointer must be non-null.

Reimplemented from AMP::TimeIntegrator.

Definition at line 156 of file ImplicitTimeIntegrator.cc.

References AMP::TimeIntegrator::putToDatabase().

The documentation for this class was generated from the following files:

- ImplicitTimeIntegrator.h
- ImplicitTimeIntegrator.cc

2.6 AMP::ImplicitTimeIntegratorParameters Class Reference

Parameter class for implicit time integrators.

`#include <ImplicitTimeIntegratorParameters.h>`

Inherits AMP::TimeIntegratorParameters.

Public Member Functions

- **ImplicitTimeIntegratorParameters** (boost::shared_ptr< AMP::Database > db)

Public Attributes

- boost::shared_ptr< SolverStrategy > d_solver

  Pointers to implicit equation and solver strategy objects and patch hierarchy.

2.6.1 Detailed Description

Parameter class for implicit time integrators.

Class ImplicitTimeIntegratorParameters contains the parameters to initialize an implicit time integrator class. It contains a Database object and a pointer to a SolverStrategy object.

Parameters:

d_solver pointer to SolverStrategy

See also:

SolverStrategy

Definition at line 36 of file ImplicitTimeIntegratorParameters.h.

2.6.2 Member Data Documentation

2.6.2.1 boost::shared_ptr< SolverStrategy > AMP::ImplicitTimeIntegratorParameters::d_solver

Pointers to implicit equation and solver strategy objects and patch hierarchy.

The strategies provide nonlinear equation and solver routines for treating the nonlinear problem on the hierarchy.

Definition at line 48 of file ImplicitTimeIntegratorParameters.h.

The documentation for this class was generated from the following files:

- ImplicitTimeIntegratorParameters.h
- ImplicitTimeIntegratorParameters.cc

2.7 AMP::RK23TimeIntegrator Class Reference

Class RK23TimeIntegrator is a concrete time integrator that implements the explicit Bogacki-Shampine adaptive Runge-Kutta (Matlab ode23) method.

`#include <RK23TimeIntegrator.h>`

Inherits AMP::TimeIntegrator.

Public Member Functions

- RK23TimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Constructor that accepts parameter list.

- ~RK23TimeIntegrator ()

  Destructor.

- void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize from parameter list.

- void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Resets the internal state of the time integrator as needed.

- double getInitialDt ()

  Specify initial time step.

- double getNextDt (const bool good_solution)

  Specify next time step to use.

- bool checkNewSolution (void) const

  Determine whether time advanced solution is satisfactory.

- void updateSolution (void)

  Update state of the solution.

- int advanceSolution (const double dt, const bool first_step)

  Integrate entire patch hierarchy through the specified time increment.

2.7.1 Detailed Description

Class RK23TimeIntegrator is a concrete time integrator that implements the explicit Bogacki-Shampine adaptive Runge-Kutta (Matlab ode23) method.

Definition at line 21 of file RK23TimeIntegrator.h.

2.7.2 Member Function Documentation

2.7.2.1 void AMP::RK23TimeIntegrator::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implements AMP::TimeIntegrator.

Definition at line 61 of file RK23TimeIntegrator.cc.

2.7.2.2 int AMP::RK23TimeIntegrator::advanceSolution (const double *dt*, const bool *first_step*) [virtual]

Integrate entire patch hierarchy through the specified time increment.

Integrate entire patch hierarchy through the specified time increment.

The boolean first_step argument is true when this is the very first call to the advance function or if the call occurs immediately after the hierarchy has changed due to re-gridding. Otherwise it is false. Note that, when the argument is true, the use of extrapolation to construct the initial guess for the advanced solution may not be possible.

Parameters:

dt Time step size

first_step Whether this is the first step after grid change

Returns:

value is the return code generated by the particular solver package in use

Implements AMP::TimeIntegrator.

Definition at line 100 of file RK23TimeIntegrator.cc.

References AMP::TimeIntegrator::d_operator.

The documentation for this class was generated from the following files:

- RK23TimeIntegrator.h
- RK23TimeIntegrator.cc

2.8 AMP::RK2TimeIntegrator Class Reference

Class RK2TimeIntegrator is a concrete time integrator that implements the explicit Runge-Kutta second order (RK2) method.

`#include <RK2TimeIntegrator.h>`

Inherits AMP::TimeIntegrator.

Public Member Functions

- RK2TimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Constructor that accepts parameter list.

- ~RK2TimeIntegrator ()

  Destructor.

- void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize from parameter list.

- void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Resets the internal state of the time integrator as needed.

- double getInitialDt ()

  Specify initial time step.

- double getNextDt (const bool good_solution)

  Specify next time step to use.

- bool checkNewSolution (void) const

  Determine whether time advanced solution is satisfactory.

- void updateSolution (void)

Update state of the solution.

- int advanceSolution (const double dt, const bool first_step)

  Integrate entire patch hierarchy through the specified time increment.

2.8.1 Detailed Description

Class RK2TimeIntegrator is a concrete time integrator that implements the explicit Runge-Kutta second order (RK2) method.

Definition at line 17 of file RK2TimeIntegrator.h.

2.8.2 Member Function Documentation

2.8.2.1 void AMP::RK2TimeIntegrator::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implements AMP::TimeIntegrator.

Definition at line 61 of file RK2TimeIntegrator.cc.

2.8.2.2 int AMP::RK2TimeIntegrator::advanceSolution (const double *dt*, const bool *first_step*) [virtual]

Integrate entire patch hierarchy through the specified time increment.

Integrate entire patch hierarchy through the specified time increment.

The boolean first_step argument is true when this is the very first call to the advance function or if the call occurs immediately after the hierarchy has changed due to regridding. Otherwise it is false. Note that, when the argument is true, the use of extrapolation to construct the initial guess for the advanced solution may not be possible.

Parameters:

dt Time step size

first_step Whether this is the first step after grid change

Returns:

value is the return code generated by the particular solver package in use

Implements AMP::TimeIntegrator.

Definition at line 91 of file RK2TimeIntegrator.cc.

References AMP::TimeIntegrator::d_operator.

The documentation for this class was generated from the following files:

- RK2TimeIntegrator.h
- RK2TimeIntegrator.cc

2.9 AMP::RK4TimeIntegrator Class Reference

Class RK4TimeIntegrator is a concrete time integrator that implements the explicit Runge-Kutta fourth order (RK4) method.

`#include <RK4TimeIntegrator.h>`

Inherits AMP::TimeIntegrator.

Public Member Functions

- RK4TimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Constructor that accepts parameter list.

- ~RK4TimeIntegrator ()

  Destructor.

- void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize from parameter list.

- void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Resets the internal state of the time integrator as needed.

- double getInitialDt ()

  Specify initial time step.

- double getNextDt (const bool good_solution)

  Specify next time step to use.

- bool checkNewSolution (void) const

  Determine whether time advanced solution is satisfactory.

- void updateSolution (void)

Update state of the solution.

- int advanceSolution (const double dt, const bool first_step)

  Integrate entire patch hierarchy through the specified time increment.

2.9.1 Detailed Description

Class RK4TimeIntegrator is a concrete time integrator that implements the explicit Runge-Kutta fourth order (RK4) method.

Definition at line 17 of file RK4TimeIntegrator.h.

2.9.2 Member Function Documentation

2.9.2.1 void AMP::RK4TimeIntegrator::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implements AMP::TimeIntegrator.

Definition at line 61 of file RK4TimeIntegrator.cc.

2.9.2.2 int AMP::RK4TimeIntegrator::advanceSolution (const double *dt*, const bool *first_step*) [virtual]

Integrate entire patch hierarchy through the specified time increment.

Integrate entire patch hierarchy through the specified time increment.

The boolean first_step argument is true when this is the very first call to the advance function or if the call occurs immediately after the hierarchy has changed due to regridding. Otherwise it is false. Note that, when the argument is true, the use of extrapolation to construct the initial guess for the advanced solution may not be possible.

Parameters:

dt Time step size

first_step Whether this is the first step after grid change

Returns:

value is the return code generated by the particular solver package in use

Implements AMP::TimeIntegrator.

Definition at line 97 of file RK4TimeIntegrator.cc.

References AMP::TimeIntegrator::d_operator.

The documentation for this class was generated from the following files:

- RK4TimeIntegrator.h
- RK4TimeIntegrator.cc

2.10 AMP::TimeIntegrator Class Reference

Abstract base class for time integration.

`#include <TimeIntegrator.h>`

Inherited by AMP::ExplicitEuler, AMP::IDATimeIntegrator, AMP::ImplicitTimeIntegrator, AMP::RK23TimeIntegrator, AMP::RK2TimeIntegrator, and AMP::RK4TimeIntegrator.

Public Member Functions

- TimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > parameters)

  The constructor for TimeIntegrator initializes the default state of the integrator.

- virtual ~TimeIntegrator ()

  Empty destructor for TimeIntegrator.

- virtual void initialize (boost::shared_ptr< TimeIntegratorParameters > parameters)

  Initialize state of time integrator.

- virtual void reset (boost::shared_ptr< TimeIntegratorParameters > parameters)=0

  Resets the internal state of the time integrator as needed.

- virtual int advanceSolution (const double dt, const bool first_step)=0

  Integrate entire patch hierarchy through the specified time increment.

- virtual bool checkNewSolution (void) const =0

  Check time advanced solution to determine whether it is acceptable.

- virtual void updateSolution (void)=0

Update solution (e.g., reset pointers for solution data, update dependent variables, etc.

- virtual boost::shared_ptr< Vector > getCurrentSolution (void)

  Retrieve the current solution.

- virtual double getNextDt (const bool good_solution)=0

  Return time increment for next solution advance.

- virtual double getInitialTime () const

  Return initial integration time.

- virtual double getFinalTime () const

  Return final integration time.

- virtual double getCurrentTime () const

  Return current integration time.

- virtual double getCurrentDt () const

  Return current timestep.

- virtual int getIntegratorStep () const

  Return current integration step number.

- virtual int getMaxIntegratorSteps () const

  Return maximum number of integration steps.

- virtual bool stepsRemaining () const

  Return true if the number of integration steps performed by the integrator has not reached the specified maximum; return false otherwise.

- void printClassData (std::ostream &os) const

  Print out all members of integrator instance to given output stream.

- void putToDatabase (boost::shared_ptr< AMP::Database > db)

  Write out state of object to given database.

- void **registerOperator** (boost::shared_ptr< Operator > op)

Protected Member Functions

- void **getFromInput** (const boost::shared_ptr< AMP::Database > db)
- void **getFromRestart** ()

Protected Attributes

- std::string **d_object_name**
- boost::shared_ptr< Vector > **d_solution**
- boost::shared_ptr< Vector > **d_pPreviousTimeSolution**
- boost::shared_ptr< Operator > d_operator

  The operator is the right hand side operator for an explicit integrator when the time integration problem is : u_t = f(u) but in the case of implicit time integrators the operator represents u_t-f(u).

- boost::shared_ptr< Operator > d_pMassOperator

  The operator is the right hand side operator for an explicit integrator when the time integration problem is : u_t = f(u) but in the case of implicit time integrators the operator represents u_t-f(u).

- double **d_initial_time**
- double **d_final_time**
- double **d_current_time**
- double **d_current_dt**
- double **d_old_dt**
- double **d_min_dt**
- double **d_max_dt**
- int **d_integrator_step**
- int **d_max_integrator_steps**

2.10.1 Detailed Description

Abstract base class for time integration.

Class TimeIntegrator is an abstract base class for managing time integration

Initialization of an TimeIntegrator object is performed through a TimeIntegratorParameters object

Definition at line 41 of file TimeIntegrator.h.

2.10.2 Constructor & Destructor Documentation

2.10.2.1 AMP::TimeIntegrator::TimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > *parameters*)

The constructor for TimeIntegrator initializes the default state of the integrator.

Data members are initialized from the input and restart databases.

Note that no vectors are created in the constructor. Vectors are created and initialized in the initialize() member function.

Definition at line 45 of file TimeIntegrator.cc.

References initialize().

2.10.3 Member Function Documentation

2.10.3.1 void AMP::TimeIntegrator::initialize (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [virtual]

Initialize state of time integrator.

This includes creating solution vector and initializing solver components.

Reimplemented in AMP::BackwardEulerTimeIntegrator, AMP::ExplicitEuler, AMP::IDATimeIntegrator, AMP::ImplicitTimeIntegrator, AMP::RK23TimeIntegrator, AMP::RK2TimeIntegrator, and AMP::RK4TimeIntegrator.

Definition at line 71 of file TimeIntegrator.cc.

References d_operator, and d_pMassOperator.

Referenced by AMP::RK4TimeIntegrator::initialize(), AMP::RK2TimeIntegrator::initialize(), AMP::RK23TimeIntegrator::initialize(), AMP::IDATimeIntegrator::initialize(), AMP::ExplicitEuler::initialize(), and TimeIntegrator().

2.10.3.2 virtual void AMP::TimeIntegrator::reset (boost::shared_ptr< TimeIntegratorParameters > *parameters*) [pure virtual]

Resets the internal state of the time integrator as needed.

A parameter argument is passed to allow for general flexibility in determining what needs to be reset Typically used after a regrid.

Implemented in AMP::BackwardEulerTimeIntegrator, AMP::ExplicitEuler, AMP::IDATimeIntegrator, AMP::ImplicitTimeIntegrator, AMP::RK23TimeIntegrator, AMP::RK2TimeIntegrator, and AMP::RK4TimeIntegrator.

2.10.3.3 virtual int AMP::TimeIntegrator::advanceSolution (const double *dt*, const bool *first_step*) [pure virtual]

Integrate entire patch hierarchy through the specified time increment.

Integrate entire patch hierarchy through the specified time increment.

The boolean first_step argument is true when this is the very first call to the advance function or if the call occurs immediately after the hierarchy has changed due to regridding. Otherwise it is false. Note that, when the argument is true, the use of extrap-

olation to construct the initial guess for the advanced solution may not be possible.

Parameters:

dt Time step size

first_step Whether this is the first step after grid change

Returns:

value is the return code generated by the particular solver package in use

Implemented in AMP::ExplicitEuler, AMP::IDATimeIntegrator, AMP::ImplicitTimeIntegrator, AMP::RK23TimeIntegrator, AMP::RK2TimeIntegrator, and AMP::RK4TimeIntegrator.

2.10.3.4 virtual bool AMP::TimeIntegrator::checkNewSolution (void) const [pure virtual]

Check time advanced solution to determine whether it is acceptable.

Return true if the solution is acceptable; return false otherwise. The meaning of this value must be intepreted properly by the user-supplied solution checking routine.

Implemented in AMP::BackwardEulerTimeIntegrator, AMP::ExplicitEuler, AMP::IDATimeIntegrator, AMP::ImplicitTimeIntegrator, AMP::RK23TimeIntegrator, AMP::RK2TimeIntegrator, and AMP::RK4TimeIntegrator.

2.10.3.5 virtual void AMP::TimeIntegrator::updateSolution (void) [pure virtual]

Update solution (e.g., reset pointers for solution data, update dependent variables, etc.

) after time advance. It is assumed that when this routine is invoked, an acceptable new solution has been computed. The double return value is the simulation time corresponding to the advanced solution.

Implemented in AMP::BackwardEulerTimeIntegrator, AMP::ExplicitEuler, AMP::IDATimeIntegrator, AMP::ImplicitTimeIntegrator, AMP::RK23TimeIntegrator, AMP::RK2TimeIntegrator, and AMP::RK4TimeIntegrator.

2.10.3.6 double AMP::TimeIntegrator::getNextDt (const bool *good_solution*) [pure virtual]

Return time increment for next solution advance.

Timestep selection is generally based on whether the solution meets some user-defined criteria. This routine assumes that, before it is called, the routine checkNewSolution() was called. The boolean argument is the return value from that call.

Implemented in AMP::BackwardEulerTimeIntegrator, AMP::ExplicitEuler, AMP::IDATimeIntegrator, AMP::ImplicitTimeIntegrator, AMP::RK23TimeIntegrator, AMP::RK2TimeIntegrator, and AMP::RK4TimeIntegrator.

Definition at line 120 of file TimeIntegrator.cc.

2.10.3.7 void AMP::TimeIntegrator::putToDatabase (boost::shared_ptr< AMP::Database > *db*)

Write out state of object to given database.

When assertion checking is active, the database pointer must be non-null.

Reimplemented in AMP::ImplicitTimeIntegrator.

Definition at line 211 of file TimeIntegrator.cc.

Referenced by AMP::ImplicitTimeIntegrator::putToDatabase().

The documentation for this class was generated from the following files:

- TimeIntegrator.h
- TimeIntegrator.cc

2.11 AMP::TimeIntegratorFactory Class Reference

TimeIntegratorFactory is a factory class that creates specific multilevel solver classes.

`#include <TimeIntegratorFactory.h>`

Public Member Functions

- TimeIntegratorFactory ()

  Constructor.

- ~TimeIntegratorFactory ()

  Destructor.

- boost::shared_ptr< TimeIntegrator > createTimeIntegrator (boost::shared_ptr< TimeIntegratorParameters > timeIntegratorParameters)

  Factory method for generating multilevel solvers with characteristics specified by parameters.

2.11.1 Detailed Description

TimeIntegratorFactory is a factory class that creates specific multilevel solver classes.

These are used to provide methods that operate on a SAMR hierarchy

Definition at line 45 of file TimeIntegratorFactory.h.

The documentation for this class was generated from the following files:

- TimeIntegratorFactory.h
- TimeIntegratorFactory.cc

2.12 AMP::TimeIntegratorParameters Class Reference

TimeIntegratorParameters is a base class for providing parameters for the TimeIntegrator's. The Database object contained must contain the following entries:.

`#include <TimeIntegratorParameters.h>`

Inherited by AMP::IDATimeIntegratorParameters, and AMP::ImplicitTimeIntegratorParameters.

Public Member Functions

- **TimeIntegratorParameters** (const boost::shared_ptr< AMP::Database > db)

Public Attributes

- boost::shared_ptr< AMP::Database > d_db

  Database object which needs to be initialized specific to the time integrator.

- std::string **d_object_name**
- boost::shared_ptr< Vector > **d_ic_vector**
- boost::shared_ptr< Operator > **d_operator**
- boost::shared_ptr< Operator > **d_pMassOperator**

2.12.1 Detailed Description

TimeIntegratorParameters is a base class for providing parameters for the TimeIntegrator's. The Database object contained must contain the following entries:.

Required input keys and data types:

Parameters:

initial_time double value for the initial simulation time.

final_time double value for the final simulation time.

max_integrator_steps integer value for the maximum number of timesteps allowed.

All input data items described above, except for initial_time, may be overwritten by new input values when continuing from restart.

Definition at line 49 of file TimeIntegratorParameters.h.

2.12.2 Member Data Documentation

2.12.2.1 boost::shared_ptr<AMP::Database> AMP::TimeIntegratorParameters::d_db

Database object which needs to be initialized specific to the time integrator.

Documentation for parameters required by each integrator can be found in the documentation for the integrator.

Definition at line 60 of file TimeIntegratorParameters.h.

The documentation for this class was generated from the following files:

- TimeIntegratorParameters.h
- TimeIntegratorParameters.cc

2.13 AMP::TimeOperator Class Reference

base class for operator class associated with ImplicitTimeIntegrator

`#include <TimeOperator.h>`

Inherited by AMP::BackwardEulerTimeOperator.

Public Member Functions

- **TimeOperator** (boost::shared_ptr< OperatorParameters > params)
- virtual void reset (const boost::shared_ptr< OperatorParameters > &params)

  This function is useful for re-initializing an operator.

- virtual void **apply** (const boost::shared_ptr< Vector > &f, const boost::shared_ptr< Vector > &u, boost::shared_ptr< Vector > &r, const double a=-1.0, const double b=1.0)
- void **registerRhsOperator** (boost::shared_ptr< Operator > op)
- void **registerMassOperator** (boost::shared_ptr< Operator > op)
- boost::shared_ptr< Operator > **getRhsOperator** (void)

- boost::shared_ptr< Operator > **getMassOperator** (void)
- void **setPreviousSolution** (boost::shared_ptr< Vector > previousSolution)
- void **setDt** (double dt)
- boost::shared_ptr< OperatorParameters > **getJacobianParameters** (const boost::shared_ptr< Vector > &u)

Protected Member Functions

- void **getFromInput** (const boost::shared_ptr< AMP::Database > &db)

Protected Attributes

- bool **d_bLinearMassOperator**
- double **d_dCurrentDt**
- boost::shared_ptr< Operator > **d_pRhsOperator**
- boost::shared_ptr< Operator > **d_pMassOperator**
- boost::shared_ptr< Vector > **d_pPreviousTimeSolution**
- boost::shared_ptr< Vector > **d_pScratchVector**

2.13.1 Detailed Description

base class for operator class associated with ImplicitTimeIntegrator

Class ImplicitTimeOperator is a base class derived from Operator. It is the operator class associated with a ImplicitTimeIntegrator. The solver associated with the ImplicitTimeIntegrator will register this object.

See also:

ImplicitTimeIntegrator
Operator
SolverStrategy

Definition at line 39 of file TimeOperator.h.

2.13.2 Member Function Documentation

2.13.2.1 void AMP::TimeOperator::reset (const boost::shared_ptr< OperatorParameters > & *params*) [virtual]

This function is useful for re-initializing an operator.

Parameters:

params parameter object containing parameters to change

Definition at line 31 of file TimeOperator.cc.

The documentation for this class was generated from the following files:

- TimeOperator.h
- TimeOperator.cc

Index