

A Toolkit for Numerical Simulation of PDEs II: Solving Generic Multiphysics Problems

Charles Boivin, Carl Ollivier-Gooch *

*Advanced Numerical Simulation Laboratory
Department of Mechanical Engineering
The University of British Columbia*

Abstract

Numerical simulations of partial differential equations are used in a variety of domains, allowing complete testing and simulation of a product or process even before it is created. However, numerical solvers are not used in all the domains where such equations arise due to the lack of available software or knowledge of numerical methods by scientists. The difficulty of writing a numerical solver is even greater for multiphysics problems, which combine distinct but coupled physical phenomena into a single simulation.

In this article, we demonstrate how an existing high-order accurate generic numerical toolkit based on the finite-volume method was modified to allow complex multiphysics problems to be solved easily. We describe how data exchange between the different physical phenomena, the most critical process of a generic multiphysics solver, was made possible using *variable association*. The modifications to the generic numerical toolkit are detailed and numerical results to both field and interface coupling problems are presented.

Key words: Finite-volume method, generic solver, multiphysics problems, numerical solution of partial differential equations

1 Introduction

Numerical simulations of partial differential equations are used in a variety of domains including aerospace research, combustion simulation, and medical research. These simulation tools allow the scientific community to solve

* Corresponding author. 2324 Main Mall, Vancouver, BC, Canada, V6T 1Z4
Email addresses: cboivin@mech.ubc.ca (Charles Boivin), cfog@mech.ubc.ca (Carl Ollivier-Gooch).

problems of increasing complexity. This leads to an improvement of the efficiency of the design and engineering processes by allowing complete testing and simulation of a product or process even before it is created.

As powerful and useful as they are, numerical solvers are not used in all the problem domains where partial differential equation problems are encountered, due to the lack of available solvers or knowledge of numerical methods by scientists. Writing a solver is even more complicated for multiphysics problems, which combine distinct but coupled physical phenomena into a single numerical simulation. Two types of coupling are possible: *field coupling* and *interface coupling* [1]. In field coupling, physical phenomena interact over the interior of a domain; the Reynolds-averaged Navier-Stokes equations and an accompanying turbulence model are an example of field coupling. In interface coupling, different physical processes in two neighboring regions interact through a common interface; fluid-structure interaction or conjugate heat transfer problems are prime examples. Presently, multiphysics solvers are custom-written for specific problems, despite significant commonality in the coupling between physical phenomena.

In this article, we demonstrate how an existing high-order accurate generic numerical toolkit was modified to allow complex multiphysics problems to be solved. This way, scientists can simply write a small portion of code describing each of the physics of their problems — something they understand very well — and use a generic numerical toolkit to tackle the coupling and the numerical aspects of the simulation. Solutions to new and complex physical problems can then be obtained within days rather than months.

Some background information on generic solvers is first given in Section 2 and the concept of a generic finite-volume solver is covered in Section 3. Section 4 follows with a description of the modifications needed to transform the toolkit into a multiphysics solver. The `Region` class, which helps solve field coupling problems, is introduced in Section 5; sample field coupling problems are presented in Section 5.4. The `Domain` class, created to manage interface coupling, is then covered in Section 6, along with a sample interface coupling problem in Section 6.4. Finally, the results are discussed in Section 7. As a reference, the physical models used as examples in this article are given in Appendix A.

2 Background information

The main idea behind generic solvers is to separate the numerical and the physical aspects of a simulation. By modularizing the solver, it becomes easier to solve different physical phenomena with the same numerical code. One application of modularity is the research of Eyheramendy and Zimmerman [2,3,4] who have developed a toolkit for semi-automatic symbolic derivation of linear finite-element models of initial-boundary-value problems. In more recent

work [5], non-linear problems are also supported. This research is of great help for users interested in deriving discretization techniques for new physical problems, but users must still be fluent with the finite-element method: concepts such as the variational principle, weighting functions and other finite-element intricacies are used throughout the derivation. There is however no mention of multiphysics problems, where multiple element types would be used in the same simulation.

One of the most sophisticated C++ libraries for the numerical solution of partial differential equations is Diffpack [6,7,8]. This library allows the user the flexibility to interact with the finite-element solution process at different levels. Again, significant finite-element knowledge is necessary when creating custom simulation in Diffpack. In particular, the boundary conditions must be explicitly coded by the users themselves. However, users familiar with the lower levels of the library could probably combine several element types into a multiphysics simulation.

Even though research on generic numerical solvers is growing, the number of solvers dedicated specifically to supporting the solution of multiphysics problems in a generic fashion is still limited. One such commercially available solver is FEMLab [9]. As its name implies, FEMLab is also based on the finite-element method. The generic nature of FEMLab comes from its “PDE mode”, where the user can enter the parameters of partial differential equations in a generic fashion. The solver can then use the equations in any simulation. Unfortunately, there is no control on element types used for this mode; it is impossible to know if the pre-defined element type will be adequate for the physics defined by the user.

PHYSICA [10] is a multiphysics solver based on the finite-volume method. The solver provides several levels of abstraction available to the user. Most users only interact with the highest level, the model level, to implement new physical problems, but it is possible to implement new algorithms due to the modular nature of the software. This solver provides most of the functionality scientists would look for in a multiphysics solver. However, only second-order accurate methods are available, and given the fact that only about 75% of the code is available when purchasing a developer’s license, implementing high-order methods could be quite challenging.

3 Generic Finite-Volume Solver

The main drawback with using the finite-element method in generic solvers is that the method does not lend itself well to a complete separation of the physical and numerical aspects of a problem. This is highlighted by the fact that all the finite-element packages described above require good knowledge of the finite-element method from the user to develop new applications with

them effectively. The finite-volume method, however, lends itself very well to a decoupling of the numerics and the physics, because the physics of the problem mainly come into play in the calculation of fluxes. These fluxes are straightforward to identify and can be computed easily.

The generic numerical toolkit used in this research uses the finite-volume method for this very reason. Users with a good knowledge of the physics of a problem will be able to easily implement short functions to describe the fluxes, source terms and boundary conditions that accurately describe the physical phenomena. The numerical method does not affect the description of the problem. In contrast, generic solvers using the finite-element method require the user to get involved in the numerical details of the simulation.

The multiphysics solver presented in this paper is based on a high-order generic numerical toolkit named ANSLib [11]. In this section, we provide a short summary of the implementation both to benefit the reader and to document the various changes that have been made to the framework to support multiphysics problems. Section 3.1 reviews the general finite-volume formulation of a conservation law and establishes terminology used throughout the rest of the article. Section 3.2 outlines how a generic finite-volume solver can be created, using a standard interface between the solver and problem physics. We implement this interface in the form of a `Physics` class in C++, as described in Section 3.2.1. Implementation of boundary conditions is summarized in Section 3.2.2.

3.1 General finite-volume formulation

The canonical finite-volume form of a two-dimensional conservation law can be written as

$$\frac{\partial \bar{U}_i}{\partial t} = -\frac{1}{A_i} \oint_{\partial A_i} F_x \hat{n}_x + F_y \hat{n}_y ds + \frac{1}{A_i} \int_{A_i} S dA \equiv \text{FI}_i \quad (1)$$

where U is a vector of unknowns, or *flux variables*, for the problem; F_x and F_y are fluxes in the x - and y -directions, S is the source term vector, A_i is the area of the i^{th} control volume, and \hat{n} is an outward unit normal vector. Also, $\bar{U}_i = \frac{1}{A_i} \int_{A_i} U dA_i$. We will call FI_i the flux integral for control volume i , even though it also includes the source term integral.

Numerical solution of Equation 1, for any physical problem, repeatedly performs the following steps:

- (1) For boundary conditions implemented as constraints on the solution reconstruction, set up these constraints.
- (2) Perform reconstruction over all control volumes (necessary for higher than

- first-order accuracy). We use k-exact least-squares reconstruction.
- (3) Evaluate the fluxes along all control volume boundaries.
 - (4) Accumulate the fluxes in the proper control volumes, using Gauss quadrature.
 - (5) Evaluate the source term integral over each control volume, also using Gauss quadrature.
 - (6) Update the solution using a time advance or relaxation scheme.

3.2 Generic finite-volume toolkit

The generic finite-volume solver used in this research is based on the observation that the physics of the problem appears only in steps 1, 3, and 5: setting up boundary constraints and computing fluxes and source terms. Steps 2, 4, and 6 are all strictly numerical, and no knowledge of the type of problem being solved is necessary to perform these steps properly. Figure 1 illustrates the physical information required for each step in the process.

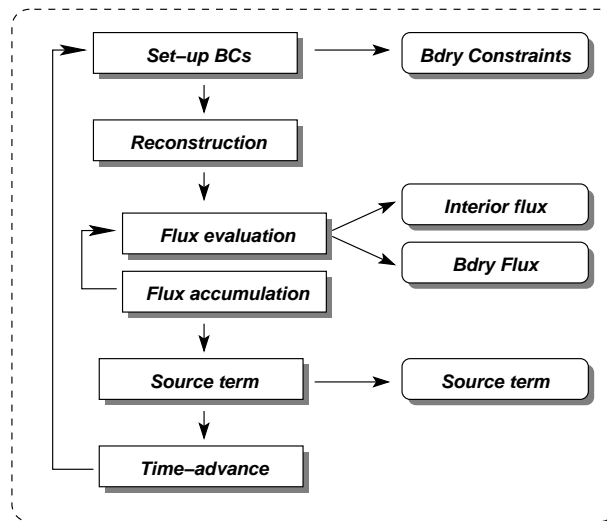


Figure 1. Detail of the physical information needed in the finite-volume method

In this research, all the numerical parts of the problem are handled by a generic finite-volume toolkit. This toolkit makes calls to external physical packages as needed to receive the relevant information about the problem being solved. The use of a standard interface between the toolkit and the physics packages allows the toolkit to interact in precisely the same way with each such package. In our implementation, the physics packages are all implemented using a `Physics` class.

3.2.1 The `Physics` class

The `Physics` class must provide all the information regarding the physical aspects of the simulation to the solver. In particular, it must provide the data

for the number of flux variables, functions for computing interior fluxes and source terms, and boundary and initial conditions. As a new addition to this interface (since [11]), the `Physics` class can also provide quantities related to the physics of the problem on demand. For example, a heat conduction `Physics` class is able to compute the wall heat flux q_n anywhere on the boundary of the domain. These quantities can then be used for post-processing purposes.

3.2.2 Boundary condition types

Boundary condition treatment has also been improved in the process of upgrading the generic solver for multiphysics problems. Each boundary face in a mesh now has a boundary condition number associated with it. This number is associated at runtime with a boundary condition *type*, specific to the `Physics` class. This boundary condition type identifies what physical boundary condition or conditions should be enforced on that part of the boundary. Because this association can be changed from one geometry to another, boundary conditions are more easily decoupled from the mesh generation process.

The boundary condition types can be defined using either a specific boundary flux, or by specifying constraints on some reconstruction at the boundary. It is possible to assign several boundary types to a boundary condition number. For example, using the solid mechanics package described in Section A.2, displacements in both x and y can be imposed on the same surface using two different constraints. The only requirement is that only *one* boundary flux be specified for a given boundary condition number. The code places no limits on the number of constraints assigned to a given boundary condition number.

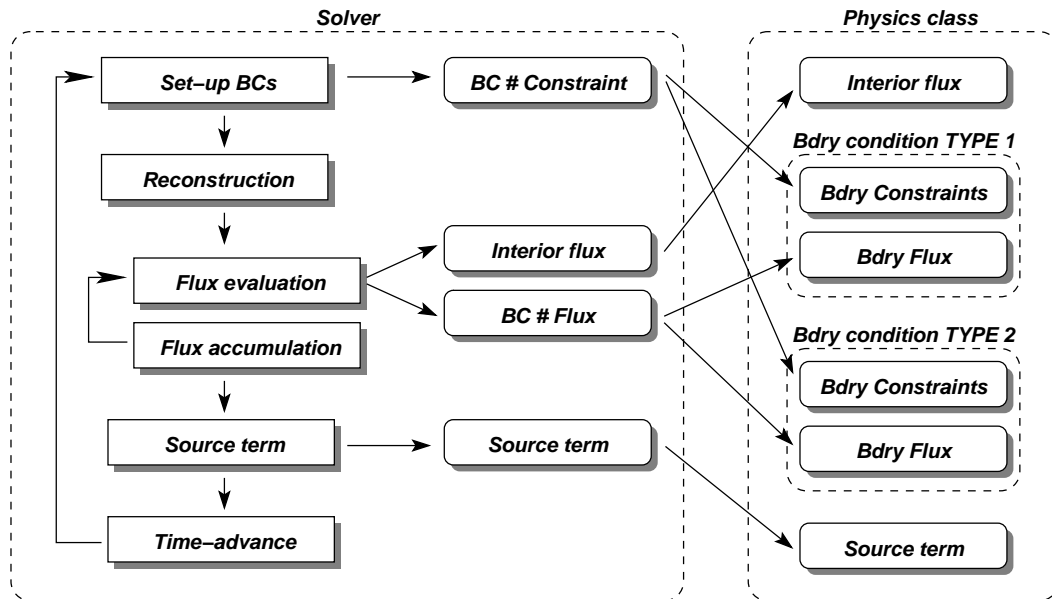


Figure 2. Solver making calls to external `Physics` class

Figure 2 shows schematically how the solver interacts with an external `Physics`

class whenever physical information is needed. The `Physics` class is shown with two boundary condition types for illustration purposes.

Using this generic interface of the solver the solver is able to solve single-physics problems as varied as heat conduction, solid mechanics, and incompressible fluid flow. Physical models for these problems, used as examples throughout this article, are described in details in Appendix A.

4 Generic Multiphysics Solver

In multiphysics problems, several physical phenomena interact to produce a coupled solution. Each combination of physical phenomena have a different way of interacting with each other, but some similarities can be seen. In particular, it is possible to categorize the interactions using the location of the coupling, and these two categories are: *field coupling* and *interface coupling* [1].

Field coupling In field coupling, the physical phenomena interact over the interior of a domain. One physical phenomenon requires information from the other phenomenon over the entire domain, and vice-versa. The interaction between the Reynolds-averaged Navier-Stokes (RANS) equations and an accompanying turbulence model is an example of field coupling. At every location in the domain, the turbulence model equations require information about the mean flow from the RANS equations, and the RANS equations need the eddy viscosity ν_T to compute Reynolds stresses.

Interface coupling In interface coupling, multiple physical processes in two neighboring regions interact through a common interface. In this case, the information exchange only happens at the boundary between the two regions. Fluid-structure or conjugate heat transfer problems are examples of interface coupling. In a conjugate heat transfer problem, the coupling is relatively simple, as both the temperature and the normal heat flux from the neighbor regions must match at the boundary. The coupling for fluid-structure problems is far more complex. Analogously to the conjugate heat transfer problem, the normal and shear stresses in the fluid and solid must match at the boundary. In addition, deformation of the solid influences the flow by way of deformation of the interface: the fluid domain changes over time, and the velocity of the interface is used to impose the no-slip condition in the fluid region (if a transient solution is required).

Numerical simulation Writing a generic multiphysics solver poses a number of challenges. Even though the work done for this research was based on an already proven generic solver, much of the interface with the `Physics`

class for this solver had to be re-written [12,13,14] to allow multiple `Physics` classes in a domain (for field coupling); multiple subdomains, or *regions*, each with their own set of `Physics` classes (for interface coupling); and to provide a mechanism for exchange of information between different `Physics` classes (for both field and interface coupling). Single-physics problems are now treated as a special case of multiphysics problems.

4.1 *Physical variables*

The first requirement for data exchange is that the solver must know what information each physical package can provide to others. This is accomplished by the use of physical variables, defined using the `PhysVar` class. Every piece of data the `Physics` class needs or can provide to other class has an accompanying variable. The `Physics` class keeps a list of `PhysVars`.¹

The items in the list of variables for a given `Physics` class are created at the same time the class is created. Each `PhysVar` contains all the information the solver needs to know about a variable: an identification number, a name, a symbol, units, and a variable type. The values of variables can be accessed from within the `Physics` class using the `vecValues` array, a `Physics` class member variable. For example, the variable with identification number `eDensity` is located in `vecValues[eDensity]`. The different variable types are listed below.

Flux variables This variable type is used to let the solver know about the conserved variables of the problem. The user defines as many of this type of variable as there are items in the solution vector. The solver will size the different arrays needed in the simulation using the information given by the number of flux variables in a `Physics` class. Furthermore, the identification number assigned to each of the flux variables is used in the `Physics` class itself by the reconstruction variables and the boundary constraints. However, the value of the flux variable is never stored using this identification number, so this variable type is used only for information purposes. Reconstruction variables (see below) are used to access the values of the flux variables.

As an example, the heat conduction package only has one flux variable: the temperature T .

Required variables Variables of this type are used to indicate information that is needed by the `Physics` class to compute such things as the fluxes or the source terms.

¹ It should be noted that this class is only used to *describe* the data each `Physics` class needs or provides, and that the actual value of the data is *not* stored in the `PhysVar` class.

The heat conduction package used in our example has three required variables: the conductivity k , the density ρ and the specific heat c_p .

Boundary condition variables The boundary condition variables are a special type of required variables used by the boundary flux or the boundary constraint functions for data specific to a boundary condition type. This variable requires an extra parameter that tells the solver which boundary condition type it is associated with.

For example, in the heat conduction package, one of the boundary condition types imposes a specific temperature on the boundary. This is done using a boundary constraint. A boundary condition variable T_b is then created to contain the value of the temperature at the boundary, and the constraint is set to that value.

Reconstruction variables This type of variable is used by the `Physics` class to access reconstruction data computed by the `Recon` object. The reconstruction variables require some extra parameters. The first one is the type of reconstruction data that is needed. The choices are the following: *Location*, *Normal*, *Solution*, or *Gradient*.

The *Location* type tells the solver that some information about the location of the Gauss points is needed. The *Normal* type is used to indicate that information about the normal of the control volume face is required. Both of these types require one extra parameter which tells the solver which item in the location or the normal vector is required (i.e. x , y , or z).

The *Solution* type is used whenever the value of one of the flux variables is needed. It requires two extra parameters: the first one is the side of the data to be returned (i.e. *left* or *right* of the control volume boundary), and the second one tells which flux variable should be used. This last parameter uses the identification number of one of the flux variables.

The *Gradient* type tells the solver that the gradient of one of the flux variables is needed. This type requires three extra parameters. Similar to the *Solution* type, the side and the flux variable identification number are needed. The gradient direction is also needed (x , y , or z).

In the heat conduction package, the interior flux needs access to the temperature gradients. Four reconstruction variables of type *Gradient* are needed to compute the flux: two for $\partial T/\partial x$ (left and right), the other two for $\partial T/\partial y$ (left and right).

All of the previous variable types have been for data that was required by the `Physics` class. The next two types are for data that can be provided by the `Physics` class to the solver, the user, or to other `Physics` packages.

Computed variables A computed variable is another quantity besides fluxes and source terms that the `Physics` class can compute. This type of variable is useful for field coupling and for boundary conditions in multiphysics problems. For a quantity to be exported from the `Physics` class, a computed variable must be defined for it.

As an example, when solving a conjugate heat transfer problem, the heat fluxes from both sides must match at the boundary, so the normal heat flux $q_n = k \frac{\partial T}{\partial n}$ from the heat conduction package is exported using a computed variable. This information can then be used by other packages. For example, in conjugate heat transfer problems, this value can be used to couple the fluid and the solid regions at the interface between the two, since both the heat fluxes and the temperature must match at that location.

Computed variables can also be used to avoid repetitive computations. For example, in the heat conduction package, the heat diffusivity α is required for flux computations. As diffusivity can be computed from k , ρ , and c_p , a computed variable is created for it. This avoids having to compute the diffusivity twice during the flux calculations. The same principle can also apply to reconstruction variables. In the heat conduction package, using the average of the left and right reconstruction data is adequate. Computed variables are created for the average of temperature gradients, and simplify the flux definition. Similarly, it is possible to export the value of temperature at some face by creating a computed variable returning the average of the left and right values of the temperature on that face.

Computed variables get their own function, just like the interior flux function for example, where the value of the variable is actually computed by the `Physics` class. This function returns the values for all the computed variables and uses the identification number of the variable to know which one is needed.

Constant variables Constant variables are also values that can be exported from a `Physics` class. They are a special case of computed variables, and allow the solver to fetch data in a more efficient way. Using a large number of constant variables, it is possible to create a “database” `Physics` class, whose only function is to provide constants to other `Physics` classes.

4.2 Variable association

Once all the `Physics` classes have listed their variable requirements, coupling the physics packages together is a straightforward process conceptually. For this task, *variable association* between a required variables in one physics package and a provided variable in another is used. The association is done by having the solver link the two variables together using a pointer. The

solver then automatically uses the value of the associated provided variables whenever the value of a given required variable is needed.

4.3 Variable dependencies and dependency trees

In certain cases, the value for some variables can not be obtained until the value of other variables are available. These other variables are called *dependencies* of the original variable. Here is a list of the dependencies for each type of variable.

Flux variables: Flux variables are only used for information purposes (see Section 4.1). They have no dependency.

Required variables: Required variables have only one dependency: their associated variable.

Boundary condition variables: Since boundary condition variables are a special case of required variables, they too only have their associated variable as a dependency.

Reconstruction variables: The values for reconstruction variables can always be obtained by the solver. This variable type therefore does not have any dependency.

Computed variables: Computed variables can have any number of dependencies. The dependencies are always variables that are local to the `Physics` class. The `Physics` class must inform the solver of the dependencies of each of its computed variables.

Constant variables: Constant variables provide values to other `Physics` classes. They do not have any dependency.

The solver uses the value of an associated provided variable whenever the value of a required variable is needed. Using this approach blindly can have some serious flaws. First, there is no guarantee that the value of the associated variable will be available at the time it is needed: a variable value is available only if the values for all of its dependencies are also available. Second, several variables can be associated to the same required variable. The solver would then fetch (or compute) the value of the associated variable several times, with a serious impact on efficiency.

This problem was solved by using *dependency trees*. A separate dependency tree is created for each of the following tasks: computing the interior flux, computing the boundary flux, setting the boundary constraints, and computing the source term. Dependency trees determine the order in which variable values are fetched, and help avoid availability issues and unnecessary computations.

Take for example the interior flux dependency tree for the solid mechanics package (described in Appendix A.2), shown in Figure 3. The interior fluxes for this package are the stresses in the problem. The tree is built by inserting

all the variables needed for the interior flux. In these trees, the shaded items have dependencies; the items with a white background do not. Whenever an item is inserted in the tree, it is checked for dependencies. If the variable inserted in the tree depends on other variables, then these variables are recursively inserted in the tree as well, below that original node. Figure 3a shows how this applies for σ_{xx} . The stress first depends on the strain ϵ_{xx} . This, in turn, depends on the average value of $\partial u/\partial x$, which is computed from both the left and right values obtained from reconstruction. The dependencies for ϵ_{yy} are then introduced in Figure 3b. Figure 3c shows the rest of the σ_{xx} dependencies being added to the tree; constants E and ν are needed to compute the stress. The values for these constants is set by the user, as indicated by the dashed arrows to the constant variables. As can be seen, the chain ends whenever a reconstruction or a constant variable is inserted, neither of which have dependencies. If an item is already present in the tree, a link to that node is made instead of inserting a second copy. Figure 3d demonstrates this: inserting σ_{yy} as a dependency does not add any nodes to the tree, but new links are created. The dependencies for σ_{xy} are inserted in Figure 3e. Finally, Figure 3f shows the complete tree, with the addition of the components of the normal vector, also needed for the interior flux computation.

Whenever interior fluxes need to be computed, the solver goes through the dependency tree, starting at the lowest level. Each variable is fetched, and stored in the appropriate location. Since dependencies are always stored at a lower level, variables needed by nodes at a higher level will be available when the time comes to compute them. This method also ensures that each variable will only be fetched or computed once. In our example, variables such as E and ν are each needed by multiple variables; using a dependency tree prevents fetching these constants three times.

This dependency tree will be used to compute the interior flux at all the Gauss points on the control volume boundaries. As stated above, using the dependency tree prevents multiple variable lookups at a given Gauss point. However, there is still a possibility that constant variables will be fetched several times during the course of a simulation, even though their values do not change from location to location. For this reason, the constant values are fetched and stored only once, before the simulation starts. The constant variables are then removed from the dependency tree. Since constant variables are always located at the end of a branch, removing that node does not affect the rest of the tree. The updated dependency tree for the solid mechanics package is shown in Figure 4.

4.4 *Modifications to the Physics class*

The following is a summary of all the modifications made to the Physics class to make use of PhysVars.

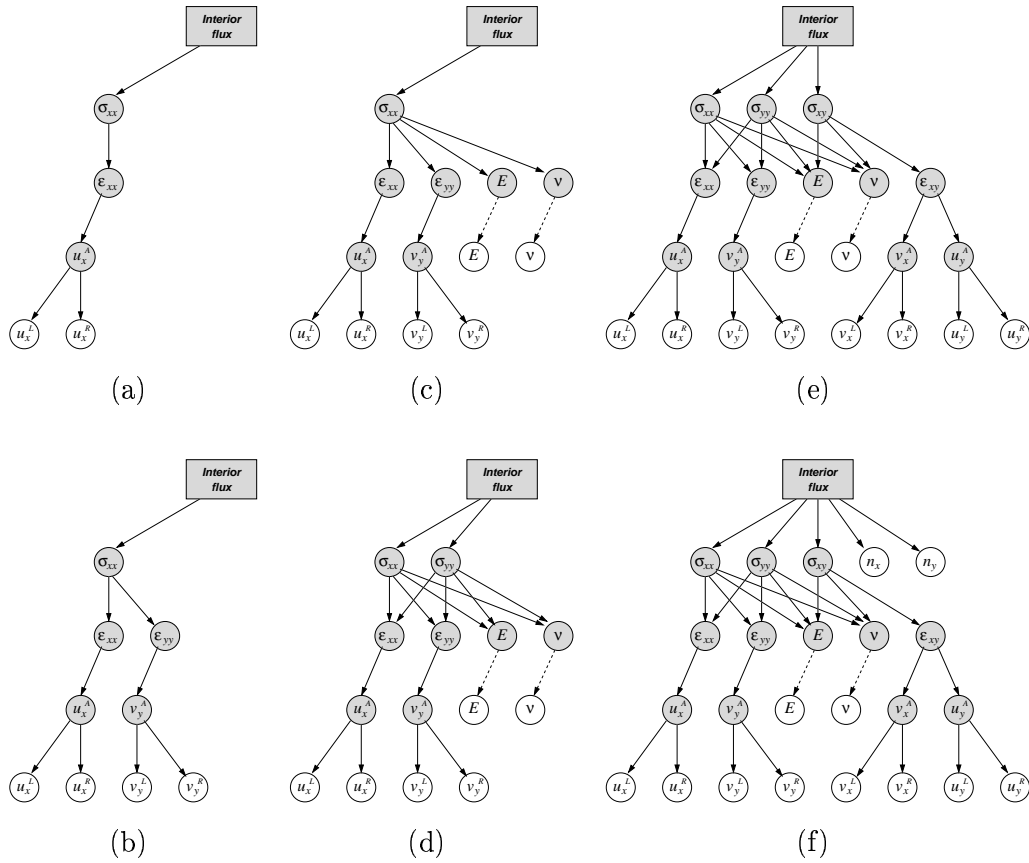


Figure 3. Dependency tree for the interior flux computation of the solid mechanics package

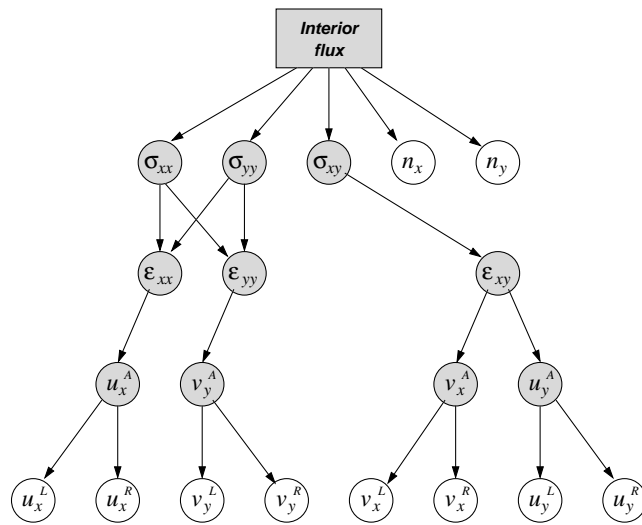


Figure 4. Optimized dependency tree for the solid mechanics package with constants removed

4.4.1 *Vectors of variables and variable values*

The `Physics` class now owns a vector which contains all the `PhysVars` it is using. The variables are stored according to their identification number. In parallel, the `Physics` class also has a vector of equal length that is used to store the values of each of the `PhysVars`.

4.4.2 *Dependency information functions*

The `Physics` class already had four functions that handled all the physical aspects of a numerical simulation: the interior flux, the boundary flux, the boundary constraints and the source term functions. These functions are called the *computing functions*. Each of these now has an associated dependency information function. This is the function that the solver calls to find out what variables are needed to compute the interior flux, for example. The dependency information functions require one parameter, a vector of boolean variables all initially set to *false*. The vector has the same size as the variable vector. The function sets the boolean variable to *true* for the variables needed to compute the flux.

4.4.3 *Modifications to the computing functions*

The computing functions now use the values stored in the `vecValues[]` vector to perform their computation. The proper values are always available when needed, thanks to the dependency information function and precomputation.

4.4.4 *Support for computed variables*

The `Physics` class now contains functions that allow variables related to the physical equations it describes to be computed. Two functions are needed: one to compute the variables (the *computing* function), and the other to inform the solver about the dependencies of the computed variables (the *dependency information* function). The computing function requires only one parameter: the identification number of the computed variable. It returns the value of the computation. The dependency information function requires two parameters: the identification number, and the vector of boolean variables indicating dependencies.

5 **Field coupling: the Region class**

Multiphysics problems invariably require multiple `Physics` classes interacting with each other. For field coupling problems, these `Physics` packages interact over the same region. In fact, it would be possible to write a super-`Physics`

class that combines the effects of multiple physical phenomena, and solve it as a single-physics problems. There are several reasons to avoid this approach.

One of the reasons is that it would require creating a highly-specialized `Physics` class. This class could then only be used when all of the physical phenomena are to be solved together; it would be impossible to use some of the separate physical aspects in a different simulation.

The main reason to avoid this approach, however, is the amount of work it would take to create a truly useable package that would combine all these physical phenomena together. In particular, the number of boundary conditions types needed would fast become unmanageable. Imagine combining just two `Physics` classes in a single super-`Physics` class. Each of the original `Physics` classes have a number of boundary condition types. Suppose that one has four boundary condition types, the other five. A super-`Physics` class might require all possible combinations of boundary condition types, which, in this case, could result in having to write twenty different boundary condition types!

A better approach is to use the same separate `Physics` packages that can be used in a single-physics problem, and solve them all at the same time. The `Region` class was created as a managing layer between the solver and the multiple `Physics` classes. The solver does not interact with the `Physics` classes directly. The `Region` class acts as a subordinate solver over a particular subdomain, and calls each `Physics` class in turn. The `Region` class also handles all the details of field coupling between the different physical packages on that subdomain. This new framework is represented schematically in Figure 5.

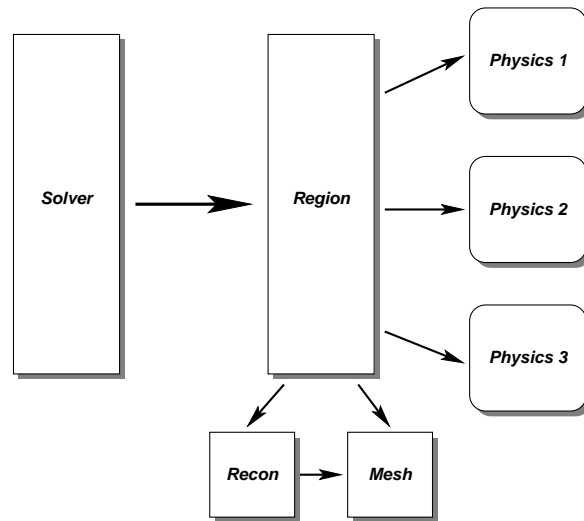


Figure 5. Schematic of the generic multiphysics framework for field coupling

The main tasks of the `Region` class are to provide interaction with the mesh, to manage a list of `Physics` classes and to support the evaluation of the fluxes on all these `Physics` classes. These tasks are described in more details in the sections below.

5.1 *Mesh interaction*

The `Region` class owns the `Mesh` object over which its `Physics` classes are being used. The `Mesh` object is passed to the `Region` at initialization. The `Region` class then extracts information from the mesh: it obtains the order of accuracy of the simulation from the mesh (as a mesh object is initialized with a particular order of accuracy), and automatically creates the proper reconstruction (`Recon`) object. The `Region` class also obtains the boundary condition information from the mesh. The `Region` class then knows the number of boundary conditions in the mesh, and can ensure that the association of boundary condition number to boundary condition type in the `Physics` classes is correct.

5.2 *Multiple Physics classes*

The `Region` class owns a list of `Physics` classes. There is no limit on the number of `Physics` classes that can be stored. The class also has functions to add, remove, and access `Physics` classes. The boundary condition types from the various `Physics` classes are associated to the boundary condition numbers in the mesh using functions in the `Region` class.

The `Region` class handles the manipulation of flux variables. The flux variables from all the `Physics` classes are combined into a single vector and sent to the appropriate `Recon` object (which the `Region` class also owns). This way, all of the flux variables are reconstructed at once, just as they would be if a single `Physics` class was used. This minimizes the impact on efficiency caused by having multiple `Physics` classes.

Variables are also managed by the `Region` class. Once all the `Physics` classes have been assigned to the `Region`, the variable list from each `Physics` class is combined into a master list. The master list contains some additional information, such as the origin of the variable. Variable association between different `Physics` classes can then take place by storing pointers from one variable to the other in the master list.

The `Region` class stores the dependency trees needed for interior flux, boundary flux, boundary constraint and source term evaluation as well. The trees are built by recursively adding the dependencies from all the `Physics` classes. The trees are then optimized to remove constant items from them. The constant values are stored at their corresponding index in the variable value vector of their `Physics` class of origin.

5.3 Flux evaluations

The most used functions in the `Region` class are the functions that read the dependency tree and store the variable values in the `Physics` classes. There are four such functions, corresponding to the four dependency trees stored for interior flux, boundary flux, boundary constraints and source term evaluation. Each of these functions is called once at each iteration, before the fluxes (or the source term) are computed in each of the `Physics` classes.

The functions go through the dependency tree, starting at the lowest level. At every node they encounter, the functions fetch the variable value (either by using data from the reconstruction object, or by asking a particular `Physics` class to compute it). The value is then stored in the variable value vector of the appropriate `Physics` class.

Now that the `Physics` classes have the right values stored in their variable value vector, the flux functions in each `Physics` class are called in turn. The `Physics` classes are not aware of other `Physics` classes being used. Therefore, the flux vectors they return are sized according to their own number of flux variables. These fluxes are then stored at the right location in the global flux vector, which is sized according to the total number of flux variables in the `Region`.

5.4 Field coupling results

Variable association and the implementation of the `Region` class allow the numerical toolkit to solve field coupling problems. Two such problems are presented here to demonstrate the effectiveness of the field coupling approach. These problems both use two `Physics` classes that are presented in Appendix A.

5.4.1 Solid mechanics and heat conduction

Consider the problem of a bar subjected to a vertical temperature gradient, as shown in Figure 6.

A temperature T_0 is imposed on the bottom surface, and a temperature T_1 is imposed on the top surface. The left and right surfaces are considered insulated. The problem is symmetric, so only the right half of the bar is used in the simulation, and a symmetry constraint is imposed on the left surface, i.e. $u = 0$ and $\frac{\partial v}{\partial x} = 0$. The top and right surfaces are free to move. The bar is fixed at $(0, 0)$.

The temperature distribution will be linear:

$$T(x, y) = T_0 + \frac{y}{H}(T_1 - T_0)$$

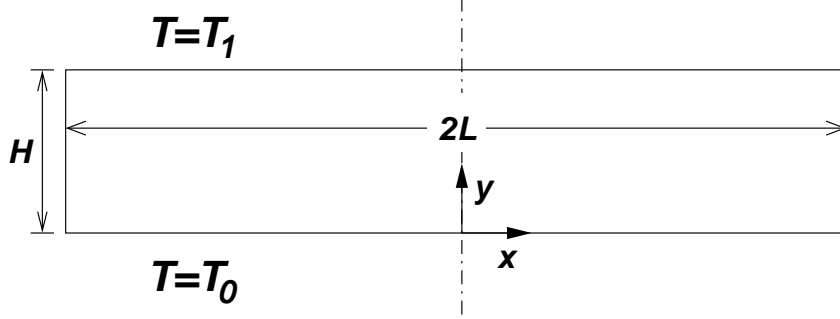


Figure 6. Sample solid mechanics with thermal strains problem

Because of this temperature distribution, the bar will change shape; the displacements in the bar are given by:

$$u(x, y) = \frac{\alpha_T(T_1 - T_0)xy}{H}$$

$$v(x, y) = \frac{\alpha_T(T_1 - T_0)}{2H} (y^2 - x^2)$$

where α_T is the coefficient of thermal expansion, and ν is Poisson's ratio. The value for $v(x, 0)$ is imposed as a mechanical displacement on the bottom surface. This should have the effect of relieving all stresses in the bar.

For the simulation, values of $L = 3$, $H = 0.5$, $E = 7000$, $\nu = 0.2$, $\alpha_T = 2 \times 10^{-5}$, $T_0 = 0$ and $T_1 = 10$ were used. A third-order accurate scheme was used. The mesh used for this problem is shown in Figure 7.

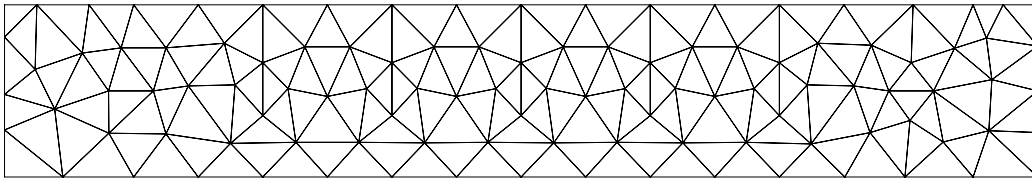


Figure 7. Mesh used for the heat conduction and solid mechanics simulation

Note that it would be possible to run such a simulation using a single specialized `Physics` class that contains both the solid mechanics and the heat conduction equations with every possible combination of boundary conditions implemented. However, this simulation was run as a multiphysics problem by using completely separate `Physics` classes, with the heat conduction class providing temperature information to the solid mechanics class.

Results The displacements in the beam matched the exact solution well, as can be seen in Figures 8 and 9. This indicates that the temperature results were also accurate, the displacements being caused by the temperature field. This

was expected, since a third-order accurate scheme should recover a quadratic solution exactly.

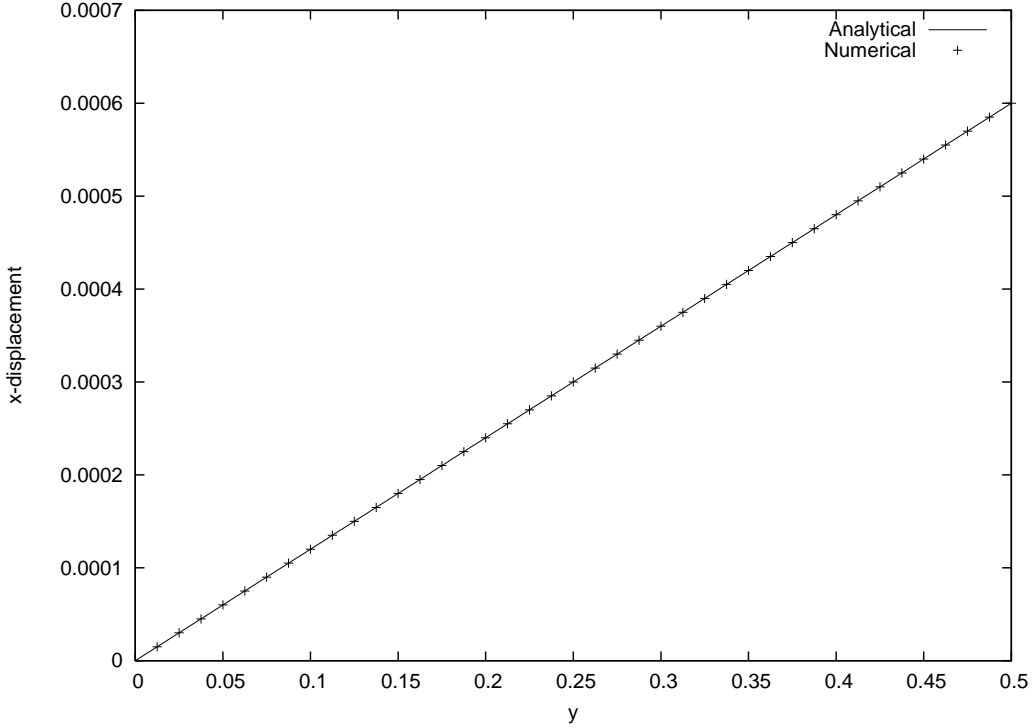


Figure 8. Displacement in x along $x = L$

The error on the displacements only depends on the convergence level of the solution. In this case, the residuals of the problems were converged below 1×10^{-13} . The largest errors in the magnitude of the stresses were 2.75×10^{-10} for σ_{xx} , 4.15×10^{-11} for σ_{yy} , and 1.46×10^{-11} for σ_{xy} .

5.4.2 Incompressible Navier-Stokes and the energy equation

The incompressible energy equation will be solved in conjunction with the Navier-Stokes package. For low-speed flows, the dissipation term can usually be neglected. However, for this example, it will be included in the simulation.

In a channel flow of height H and length L with a fully-developed profile of $u = 6\frac{y}{H}(1 - \frac{y}{H})$ and wall temperatures of $T|_{y=0} = T_0$ and $T|_{y=H} = T_H$, the fully-developed solution for the temperature profile is:

$$T = T_H \left(\frac{y}{H} \right) + T_0 \left(1 - \frac{y}{H} \right) + 6Pr \cdot Ec \phi \quad (2)$$

where:

$$\phi = \left(\frac{y}{H} \right) - 3 \left(\frac{y}{H} \right)^2 + 4 \left(\frac{y}{H} \right)^3 - 2 \left(\frac{y}{H} \right)^4$$

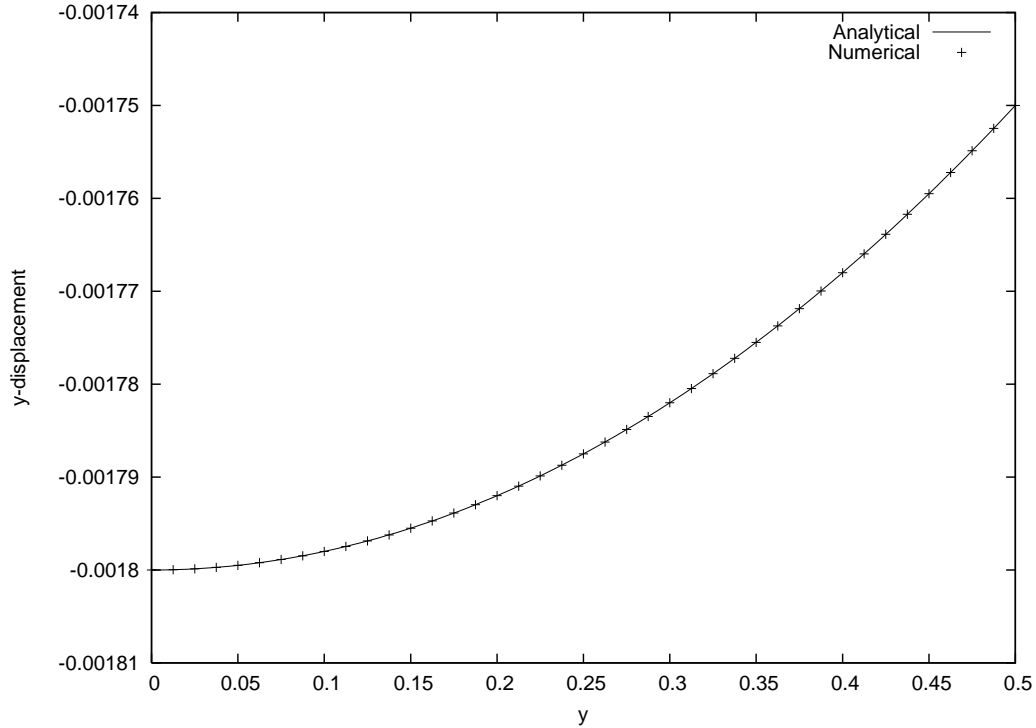


Figure 9. Displacement in y along $x = L$

For this problem, values of $L = 10$, $H = 1$, $T_0 = 0$, and $T_1 = 1$ were used. The temperature profile is then:

$$T = y + 6Pr \cdot Ec (y - 3y^2 + 4y^3 - 2y^4)$$

The Prandtl number Pr was set to 0.5 and the Eckert number Ec was set to 0.4324. Note that this is several orders of magnitude larger than typical Ec values. This was done to verify the accuracy of the source term. The input temperature profile was set to $T = y + 5.5Pr \cdot Ec(y - 3y^2 + 4y^3 - 2y^4)$ to ensure the temperature reached its fully-developed profile within the length of the channel. A fourth-order accurate reconstruction scheme was used to better approximate the quartic temperature profile. The mesh used in this simulation is shown in Figure 10.

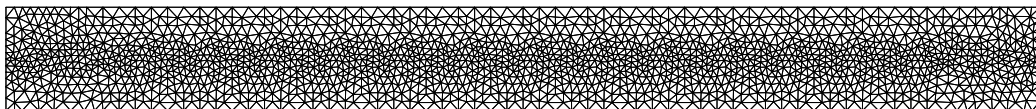


Figure 10. Mesh used for the Navier-Stokes and energy equation simulation

Results The results shown in Figure 11 were taken along $x = 9$, where the temperature had reached its fully-developed profile.

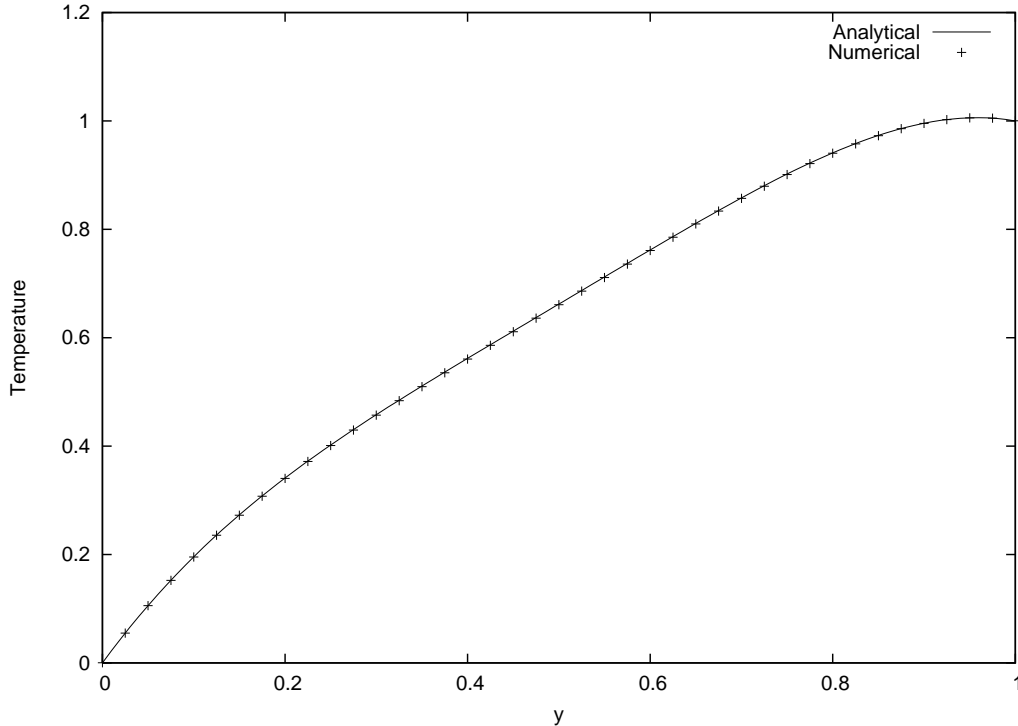


Figure 11. Temperature profile along $x = 9$ for the Navier-Stokes and energy equation simulation

The numerical solution obtained matches the analytical solution very well; this indicates that the strains computed from the Navier-Stokes solution (i.e. $\partial u/\partial x$, $\partial v/\partial y$, $\partial u/\partial y$, and $\partial v/\partial x$) were also computed accurately, and that the field coupling in the problem worked as expected. The maximum error on the temperature profile was less than 0.1% of the maximum temperature. This error is caused by the fact that the reconstruction scheme cannot recover the quartic temperature profile exactly.

6 Interface coupling: the Domain class

Interface coupling requires the ability to handle multiple subdomains, on which field coupling might still occur, and to allow coupling between `Physics` classes located on different subdomains, through a physical interface between the two regions. The `Domain` class was created to address these needs.

The `Domain` class is an example of the high leverage possible with the object-oriented approach. It was mentioned in Section 5 that the `Region` class is in fact a subordinate solver that manages its own mesh, reconstruction data, and set of `Physics` classes. By simply creating as many instances of the `Region` class as there are subdomains, and assigning each `Region` its own set of `Physics` classes, `Mesh` and `Recon` object, the generic solver can now solve problems on various subdomains. The only additional code needed handles the interactions

between different subdomains. The `Domain` class simply asks each `Region` class it owns to solve the problem on their respective subdomains in turn, as is schematically depicted in Figure 12. The interface coupling is handled through boundary conditions. More details are given in the section below.

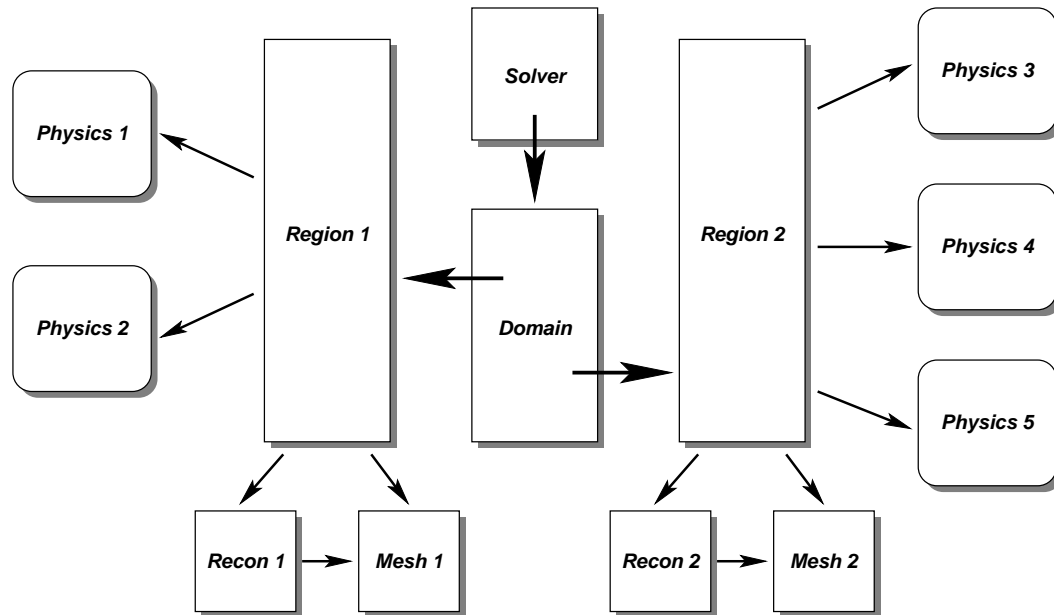


Figure 12. Schematic of a problem with both field and interface coupling

6.1 Multiple *Region* classes

The `Domain` class is initialized with a list of `Mesh` classes. Each of these meshes represents a subdomain. It should be noted that the meshes need not be of the same type, nor must the order of accuracy be the same in each subdomain. Nothing prevents the user from selecting a third-order accurate unstructured mesh in one subdomain, and coupling it with a second-order accurate structured mesh in another subdomain. In fact, this might be the most efficient way of solving some problems.

The `Domain` class goes through the list of meshes, and creates a `Region` object for each mesh. These `Region` classes are kept in an internal list. The `Domain` class also provides functions allowing direct access to the different `Region` objects.

6.2 Interface coupling

The coupling between different subdomains is done through boundary conditions. The difference from regular boundary conditions here is that the boundary condition values at the interface will be changing throughout the

simulation. For example, when solving a heat conduction problem with two subdomains with different conductivities, both the temperature and the heat flux should match at the interface. This is accomplished by using a temperature boundary condition on say, the left side, and assign it the value of the temperature at that location from the right side. Meanwhile, on the right side, a heat flux boundary condition is used at the interface, and it is assigned the value of the heat flux computed on the left side.

As with field coupling, interface coupling is accomplished using variable association. In this case, the association will always be between a boundary condition variable in one `Physics` class, and a provided variable in a `Physics` class in some other region. Since the `Domain` class has access to all the `Region` objects, it can access the master list of variables for each, and assign pointers from a required variable in a `Region` class to a provided variable in another. Since we are using pointers to variables to do the association, a `Region` class can access the proper variable, even if it is stored in a different `Region` class. The `Region` classes remain independent of each other, except for the values of their boundary condition at the common interface, which is determined by another `Region` class.

6.3 Coupling techniques

The proper choice of boundary conditions for interface coupling is critical for problem convergence. The boundary conditions must enforce all the physical couplings taking place at the boundary. For heat transfer between different media, for example, the temperature T and the normal heat flux q_n from both sides of the interface must match. Multiple coupling conditions, such as these, cannot be enforced simultaneously on both sides of the interface.²

The approach used in this research is to enforce one physical coupling from each side, as shown in Figure 13 for the heat transfer case. In this case, Region 2 enforces matching temperatures (by using a temperature boundary condition with the value determined by the temperature from Region 1), and Region 1 enforces matching heat fluxes (using a heat flux boundary condition, with the value of the heat flux computed by Region 2). The converged solution will satisfy both boundary conditions at the interface.

Careful selection of which physical coupling to place on each side is also necessary. This is to ensure that the problem on each subdomain remains well-posed. One should avoid, for example, cases where Neumann boundary conditions are

² Imposing a temperature, based on the value of temperature at the interface from the other side, on both sides at the same time would result in a constant temperature at the interface throughout the simulation. The value at the interface would be determined by the initial condition.

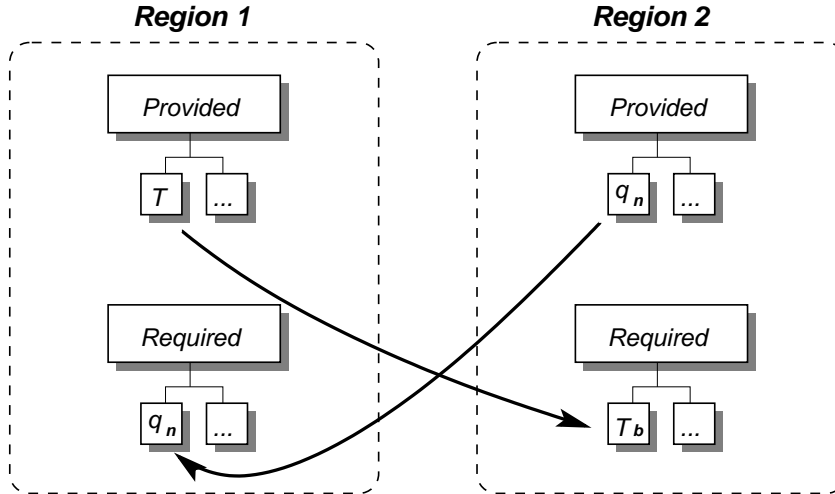


Figure 13. Interface coupling technique for a heat conduction problem

imposed on all boundary faces of a subdomain. This can usually be avoided by exchanging the side on which the couplings are enforced.

The way the boundary conditions at the interface are enforced also demands some attention. Boundary conditions can either be enforced with a constraint on the reconstruction, or through the use of a specific boundary flux. It was observed on several occasions that using constraints on the same variable (such as on the temperature T and its gradient $\partial T/\partial n$, through the normal heat flux) on both sides of a common interface resulted in unstable behavior at the boundary. The simulation would then diverge rather quickly. Invariably, enforcing one of the boundary conditions using a boundary flux (which can easily be done for the heat flux boundary condition, for example) removed the instability and resulted in a converged solution.

Lastly, the interface coupling between two subdomains where the same physical equations using the same physical constants are being solved (such as in multi-block problems) can be treated in a special way. These interfaces are arbitrary, in that they only define a boundary in the mesh, not in the problem itself. These interfaces can then use the interior flux as their “boundary” flux. The only difference will be the origin of the reconstruction data. The interior flux is computed at a control volume boundary using reconstruction data from the *left* and *right* control volumes. The boundary flux will be computed the the same way, except that the data from one side is coming from a different subdomain. Meshes need not match at the interface for this to work, as the `Recon` object can return the value of reconstruction data anywhere on the mesh. This technique is used in the problem presented in Section 6.4.

6.4 Interface coupling results

The problem solved in this section will again make use of the `Physics` classes introduced in Section A; it is a conjugate heat transfer problem. The domain, along with the `Physics` classes being used in each subdomain, is represented in Figure 14. Regions are identified using roman numerals.

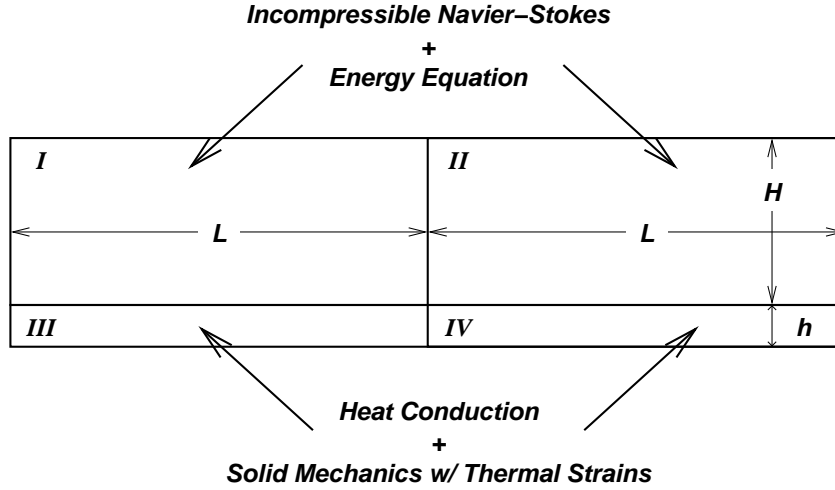


Figure 14. Domain for the interface coupling problem

Regions *I* and *II* both solve the incompressible Navier-Stokes equations along with the energy equations, as in the problem solved in Section 5.4.2. Region *I* uses a cell-centered scheme, whereas region *II* uses a vertex-centered scheme; both are solved using a third-order reconstruction scheme. Both regions *III* and *IV* use a structured mesh and a second-order scheme, and solve the heat conduction equation combined with the solid mechanics equations with thermal strains, as the problem described in Section 5.4.1.

The fluid domain The fluid domain is a multi-block domain, and is made up of regions *I* and *II*. The problem solved in the fluid domain is that of a developing channel flow. The left boundary face of region *I* imposes a uniform inflow boundary condition, with $u = 1$, $v = 0$, $\frac{\partial P}{\partial n} = 0$ and $T = 0$. The top and bottom boundary faces are considered stationary walls, and impose a no-slip boundary condition. The top wall has an imposed temperature of $T_b = 0$. The bottom wall also imposes a temperature on the flow. The value of that temperature, however, is determined from the solid domain. The right boundary face for region *II* imposes the fully-developed condition, i.e. $\frac{\partial u}{\partial x} = 0$ and $v = 0$ as well as $P = 0$. There are no constraints on the temperature at the outlet. Interface *I-II* is an arbitrary internal boundary face. The interior flux is used at this interface for the Navier-Stokes equations, as described in Section 6.3. For the energy equation, a heat flux boundary condition is used in region *I* and a temperature boundary condition is used in region *II*.

The simulation uses values of $H = 1.0$, $Re = 50$, $Pr = 0.5$, $Ec = 0.4324$, $k = 145$, and $\beta = 5$ for the fluid domain.

The solid domain The solid domain also is a multi-block domain made up of regions *III* and *IV*. The heat equation is solved in the domain, and is used to determine the thermal strains in the solid region. The left boundary face of region *III* is considered insulated, and also imposes a null x -displacement. The bottom boundary faces of regions *III* and *IV* have an imposed temperature of $T_b = 1$ and are free to move. The top boundary faces have an imposed heat flux determined by the heat flux in the fluid region, and impose a null displacement in the vertical direction. The right boundary face of region *IV* is considered insulated and free to move. Interface *III-IV* is an arbitrary internal boundary face. Matching temperature are imposed in region *III* and matching heat fluxes are imposed in region *IV*; the interior flux is used at this interface for the solid mechanics equations.

The simulation uses values of $h = 0.2$, $k = 204$, $\rho = 2707$, $c_p = 0.896$, $E = 7000$, $\nu = 0.2$, $\alpha_T = 2 \times 10^{-4}$ and $T_{ref} = 0$ in the solid domain.

Results The flow in this problem is not affected by the presence of a solid region (since displacements along the interface are constrained), so the velocity profile at the exit is simply the parabolic fully-developed profile. For this reason, the flow velocity results will not be presented here.

The temperature profile at $x = 4.7$ is used to determine convergence of the temperature field. Since the analytical solution for this problem is unknown, a mesh refinement study was done. The mesh in region *I* was kept constant, and the meshes in region *II*, *III*, and *IV* were progressively refined. The decision to keep the mesh in region *I* constant was taken in order to keep the effects of the singularities at the left corners constant. For these two corners, conflicting boundary conditions from the vertical and horizontal boundaries cause the flow to be disturbed. A refinement of the mesh in this region would have changed the effects of these singularities, and would have affected the rest of the developing flow. Table 1 lists the number of cells in all the meshes used.

#	Mesh <i>I</i>	Mesh <i>II</i>	Mesh <i>III</i>	Mesh <i>IV</i>
1	507	235	25	25
2	507	348	100	100
3	507	1117	225	225
4	507	1374	400	400

Table 1
Number of cells used in the meshes

The temperature profiles at $x = 4.7$ for the different refinement levels are shown in Figure 15. It can be seen that the temperature profile from all refinement levels is quite similar. Some discrepancies appear in the lower refinement levels, but have disappeared from the more refined tests. The meshes used in refinement level 4 were used to obtain the remainder of the results in this section.

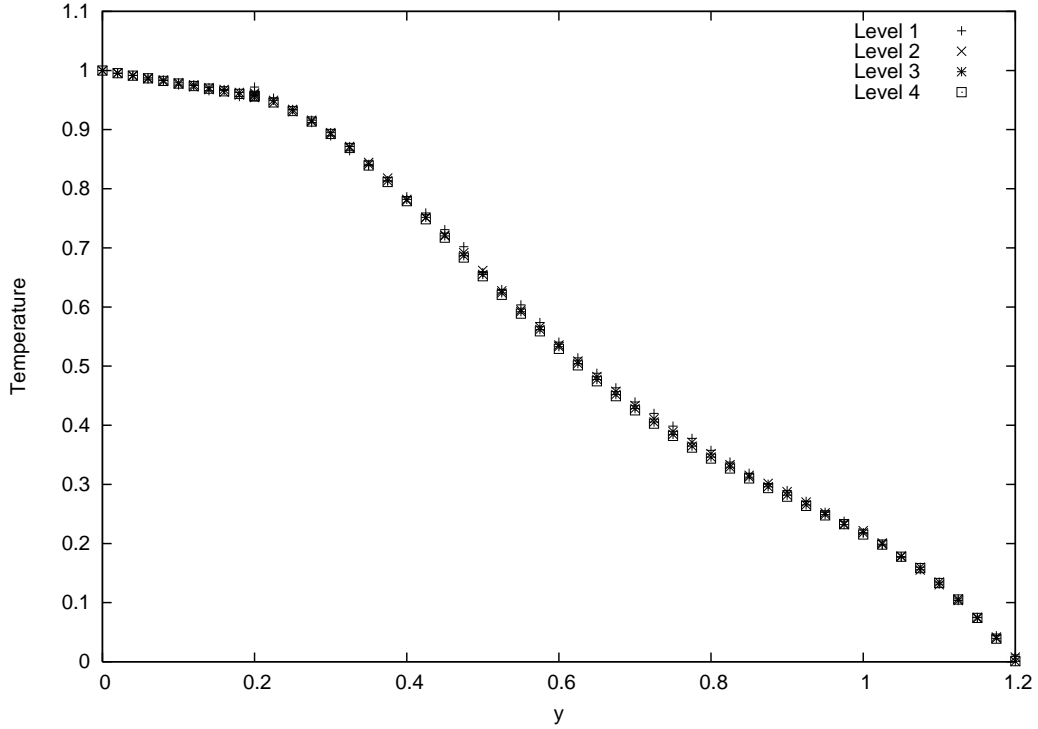


Figure 15. Temperature profiles at $x = 4.7$ for the interface coupling problem for various refinement levels

The temperature field is plotted in Figure 16.

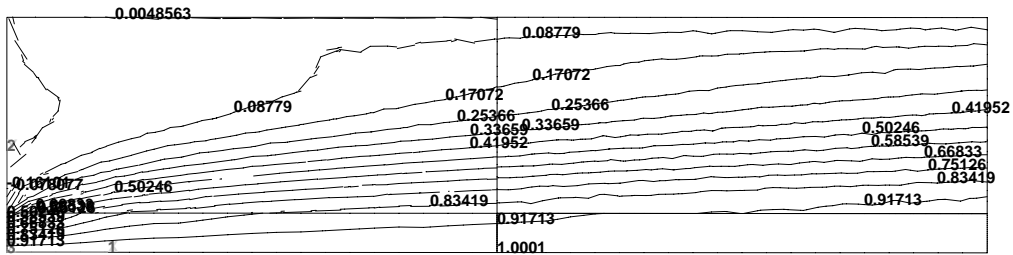


Figure 16. Temperature field for the interface coupling problem

Interface coupling The temperature field had to satisfy two conditions at the interface between the solid and the fluid: the temperatures, and the normal heat transfer from both regions had to match. Figure 17 displays the

temperature along $y = 0.2$ at the interface between regions *II* and *IV*, and Figure 18 shows the normal heat transfer along the same interface.

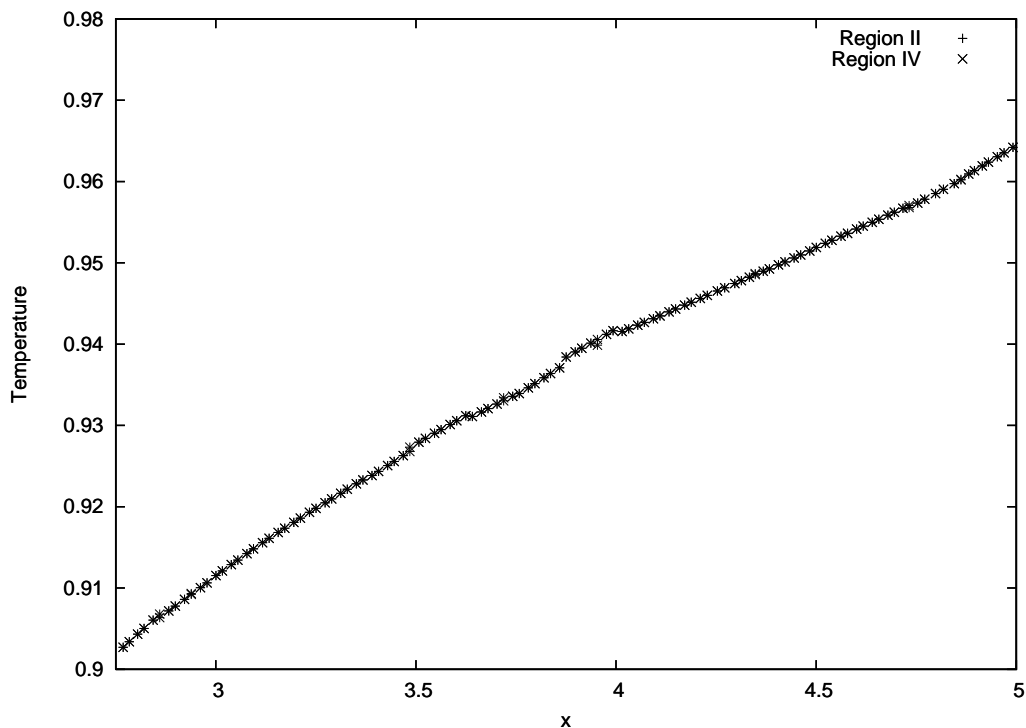


Figure 17. Temperature along the interface at $y = 0.2$

From Figure 17, it can be seen that the temperature profiles from both regions match very well at the interface. The results for the heat flux, from Figure 18, also show that the trend from both regions is the same. However, the heat flux is not matched as well as the temperature is. The second-order accurate structured region *II* can only yield constant values of heat flux for each control volume; this limits the accuracy of the computation of the heat flux in region *II*. However, it was observed that the discrepancies in the heat flux decreased as the mesh was refined.

Thermal strains in the solid region The maximum displacements observed in the solid region were $u = 9.17 \times 10^{-4}$ and $v = -3.93 \times 10^{-5}$. These displacements are reasonable, considering that a solid region with a uniform temperature of 1 would have produced a maximum displacement of $u = 1 \times 10^{-3}$ and $v = -4 \times 10^{-5}$. The displacements for the interface coupling problem were expected to fall below these.

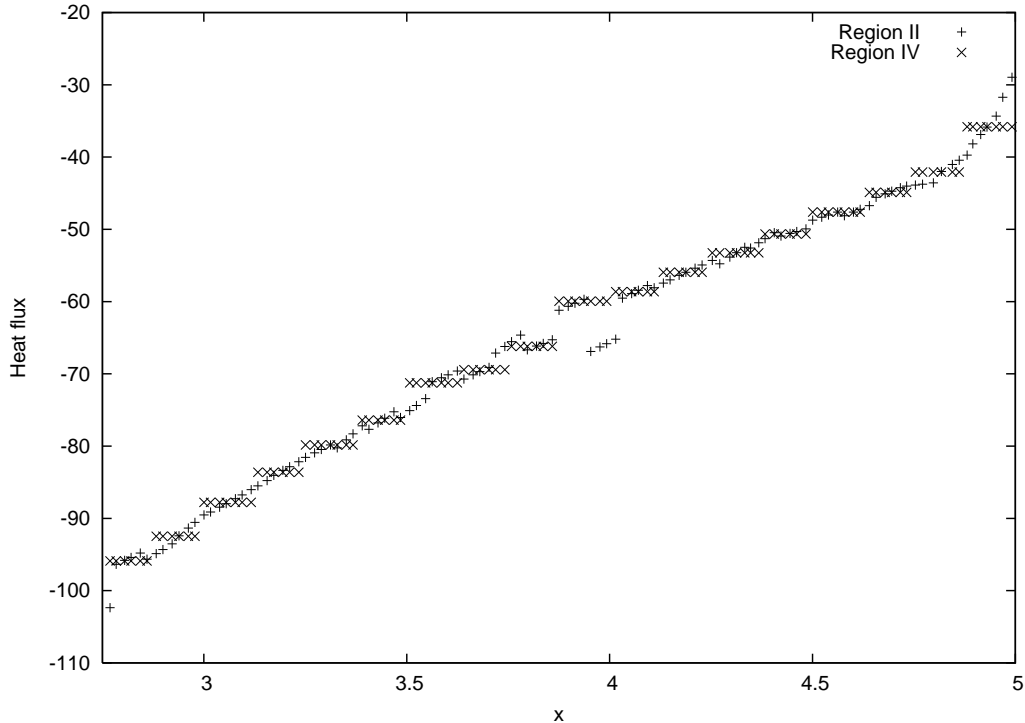


Figure 18. Normal heat flux along the interface at $y = 0.2$

7 Discussion

7.1 Deformable meshes

Even though physical packages for both fluid and solid mechanics have been written and tested successfully, realistic fluid-structure simulations cannot be performed with the multiphysics solver currently. Fluid-structure simulations require that the geometry of the problem changes throughout the simulation: the solid domain changes shape due to the forces applied to it, and this deformation in turn affects the flow field.

The possibility of the domain changing shape requires numerous changes in the way the solver operates. First, there is the problem of the mesh itself: if the domain is deformed, the mesh must also be deformed. Several approaches to perform this deformation exist in the literature, but at the moment, none have been implemented in ANSLib. Specific problems include preserving the integrity of the mesh (i.e. ensuring the cells retain a positive area), and possibly even the re-meshing of part of the domain when the mesh gets too distorted. A way to determine if the domain needs re-meshing is also needed.

The deformable meshes also have an impact on the way the problem is solved. Fluxes are computed on the faces of the control volumes, i.e. the mesh. In this research, the meshes used were always static, so the *absolute* values of the flux variables and derivatives were used for flux computation. With deformable

meshes, the control volume faces are moving at different rates throughout the domain, so this must be accounted for in the formulation of the fluxes; *relative* values of the flux variables have to be used. The information on the rate of movement of the mesh must be available to the solver and the `Physics` classes, so that the fluxes can be computed appropriately.

7.2 Efficiency

The efficiency of the solver is the area that needs the most work to bring it up to a satisfactory level. Most of the future work planned on the solver, and its numerical toolkit, focus on improving the efficiency. The generic nature of the solver makes it slower by nature than a dedicated solver. This shortcoming is more than compensated by the ability to solve new phenomena simply by writing a short description of the physical equations of the problem.

Adding multiphysics capacities further slowed the solver down, as managing the multiple `Region` and `Physics` classes requires a certain overhead. The main computational hit, however, comes from the use of variables and their necessary dependency trees. The functions responsible for going through the dependency trees and fetching/computing variables sometimes take up to 40% of the total simulation time. This is not to say that the multiphysics solver is 40% slower than the single-physics solver it is based on however: some of these computations were taking place in other portions of the code in the original solver, they have simply been centralized in the multiphysics solver. The “data fetching” functions of the multiphysics solver are unfortunately a necessary evil; much has been done to optimize them, with only limited success. There is relatively small hope that the situation would improve without a complete overhaul of the multiphysics framework.

8 Conclusion

We have presented how a generic finite-volume numerical toolkit was modified to allow multiphysics problems to be solved. Modifications included adding the `Region` class, a new layer in the framework managing multiple `Physics` classes on a particular subdomain; this helped solve field coupling problems. The `Domain` class was also created to help manage the multiple subdomains that interface coupling problems contain. The most important modification, however, was the introduction of physical variables and the concept of variable association. This association of a required variable in one `Physics` class to a provided variable in another allowed the data exchange that is essential to a generic multiphysics solver.

Now that the solver has been proven to be accurate for generic multiphysics

problems, we turn our attention to making it more efficient. In particular, projects to incorporate implicit time-advance methods and parallel computing capabilities into the toolkit are already underway. Work on both these projects is being done in parallel and given the modular nature of the generic numerical toolkit, we believe it will be possible to combine the benefits of both projects to further improve efficiency.

References

- [1] C. A. Felippa, K. Park, C. Farhat, Partitioned analysis of coupled mechanical systems, *Computer Methods in Applied Mechanics and Engineering* 190 (2001) 3247–3270.
- [2] T. Zimmermann, D. Eyheramendy, Object-oriented finite elements: I. Principles of symbolic derivations and automatic programming, *Computational Methods in Applied Mechanics and Engineering* 132 (1996) 259–276.
- [3] D. Eyheramendy, T. Zimmermann, Object-oriented finite elements: II. A symbolic environment for automatic programming, *Computational Methods in Applied Mechanics and Engineering* 132 (1996) 277–304.
- [4] D. Eyheramendy, T. Zimmermann, Object-oriented finite elements: III. Theory and application of automatic programming, *Computational Methods in Applied Mechanics and Engineering* 154 (1998) 41–68.
- [5] D. Eyheramendy, T. Zimmermann, Object-oriented finite elements: IV. Symbolic derivations and automatic programming of nonlinear formulations, *Computational Methods in Applied Mechanics and Engineering* 190 (2001) 2729–2751.
- [6] A. M. Bruaset, H. P. Langtangen, A comprehensive set of tools for solving partial differential equations: Diffpack, in: M. Dæhlen, A. Tveito (Eds.), *Numerical Methods and Software Tools in Industrial Mathematics*, Birkhäuser, 1997, pp. 63–92.
- [7] H. P. Langtangen, *Computational Partial Differential Equations — Numerical Methods and Diffpack Programming*, Lecture Notes in Computational Science and Engineering, Springer-Verlag, 1999.
- [8] A. M. Bruaset, E. J. Holm, H. P. Langtangen, Increasing the efficiency and reliability of software development for systems of PDEs, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), *Modern Software Tools for Scientific Computing*, Birkhäuser, 1997, pp. 247–268.
- [9] FEMLAB: An Introductory Course, Available from the FEMLAB website, <http://www.femlab.com> (2002).
- [10] C. Bailey, G. Taylor, S. M. Bounds, G. Moran, M. Cross, PHYSICA: A multiphysics computational framework and its application to casting

simulations, in: Computational Fluid Dynamics in Mineral & Metal Processing and Power Generation, CSIRO Division of Minerals, 1997, pp. 419–425.

- [11] C. F. Ollivier-Gooch, A toolkit for numerical simulation of PDEs I: Fundamentals of generic finite-volume simulation, *Computer Methods in Applied Mechanics and Engineering* 192 (2003) 1147–1175.
- [12] C. Boivin, C. F. Ollivier-Gooch, A generic finite-volume solver for multiphysics problems I: Field coupling, in: Proceedings of the Tenth Annual Conference of the Computational Fluid Dynamics Society of Canada, 2002, pp. 49–54.
- [13] C. Boivin, C. F. Ollivier-Gooch, A generic finite-volume solver for multiphysics problems II: Interface coupling, in: Proceedings of the Tenth Annual Conference of the Computational Fluid Dynamics Society of Canada, 2002, pp. 55–60.
- [14] C. Boivin, Infrastructure for solving generic multiphysics problems, Ph.D. thesis, Dept. of Mechanical Engineering, University of British Columbia (May 2003).
- [15] J. H. Ferziger, M. Peric, *Computational Methods for Fluid Dynamics*, 2nd Edition, Springer-Verlag, 1999.

A Physical models

A.1 Heat conduction

A.1.1 Flux and source term

Using T as the temperature, \dot{q} as the energy generated per unit volume, k as the thermal conductivity, ρ as the density, c_p as the specific heat and $\alpha = \frac{k}{\rho c_p}$ as the heat diffusivity of the material, it is possible to express the heat equation in the form of Equation 1 with:

$$U = T \tag{A.1}$$

$$F_x = -\alpha \frac{\partial T}{\partial x} \tag{A.2}$$

$$F_y = -\alpha \frac{\partial T}{\partial y} \tag{A.3}$$

$$S = \frac{\dot{q}}{\rho c_p} \tag{A.4}$$

A.1.2 Boundary condition types

Imposed temperature Imposing a temperature value at a wall is done by setting a constraint for the temperature value at the wall. The boundary flux is simply $\vec{F}_b = F_x \cdot \hat{n}_x + F_y \cdot \hat{n}_y$.

Imposed heat flux The second boundary condition type imposes a specific heat flux q_n at the wall. This boundary condition is defined by specifying a boundary flux.

$$\vec{F}_b = -\frac{q_n}{\rho c_p}$$

This boundary condition type can be used to specify an insulated wall by setting the normal heat flux to zero.

A.2 Solid mechanics: plane stress

A.2.1 Flux and source term

Using the plane stress assumption dictating that:

$$\begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{pmatrix} = \frac{E}{1-\nu^2} \begin{pmatrix} \epsilon_{xx} + \nu\epsilon_{yy} \\ \epsilon_{yy} + \nu\epsilon_{xx} \\ \left(\frac{1-\nu}{2}\right)\epsilon_{xy} \end{pmatrix} \quad (\text{A.5})$$

where E is the modulus of elasticity and ν is Poisson's ratio, it is possible to express the the differential equations of motion of a deformable solid as:

$$U = \begin{pmatrix} u \\ v \end{pmatrix} \quad (\text{A.6})$$

$$F_x = - \begin{pmatrix} \frac{E}{1-\nu^2} \left(\frac{\partial u}{\partial x} + \nu \frac{\partial v}{\partial y} - (1+\nu)\alpha_T(T - T_{ref}) \right) \\ \frac{E}{4(1+\nu)} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \end{pmatrix} \quad (\text{A.7})$$

$$F_y = - \begin{pmatrix} \frac{E}{4(1+\nu)} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \\ \frac{E}{1-\nu^2} \left(\frac{\partial v}{\partial y} + \nu \frac{\partial u}{\partial x} - (1+\nu)\alpha_T(T - T_{ref}) \right) \end{pmatrix} \quad (\text{A.8})$$

$$S = - \begin{pmatrix} B_x \\ B_y \end{pmatrix} \quad (\text{A.9})$$

A.2.2 Boundary condition types

Imposed x -displacement This boundary condition type imposes a value for displacement u at the boundary. This boundary condition type is imposed using a boundary constraint. The boundary flux used is the same as the interior flux \vec{F} .

Imposed y -displacement This boundary condition type imposes a value for displacement v at the boundary. Since any number of boundary constraints can be used, a displacement in both the x - and the y -direction can be imposed at a boundary by using a combination of the x - and y -displacement boundary conditions.

Imposed stresses The normal (σ_b) and shear (τ_b) stresses at the boundary can be enforced using a boundary flux. The flux then becomes:

$$\vec{F}_b = \begin{pmatrix} -\sigma_b \cdot \hat{n}_x + \tau_b \cdot \hat{n}_y \\ -\tau_b \cdot \hat{n}_x - \sigma_b \cdot \hat{n}_y \end{pmatrix}$$

The signs were chosen to agree with the sign convention for stresses (tensile stress is positive, compressive stress is negative).

Symmetry The symmetry boundary condition imposes two things: zero displacement in the direction normal to the boundary, zero normal derivative of the displacement tangent to the boundary. Both of these restrictions are imposed using boundary constraints. For example, for a vertical boundary face, the two constraints would be $u = 0$ and $\frac{\partial v}{\partial x} = 0$.

A.3 *Laminar incompressible Navier-Stokes*

A.3.1 *Flux and source term*

The incompressible Navier-Stokes equations can be put in the form of Equation 1 by adding a time-dependent pressure term in the continuity equation, as described in [15]. The continuity equation now looks like this:

$$\frac{\partial P}{\partial t} + \frac{1}{\beta} \frac{\partial u_i}{\partial x_i} = 0 \tag{A.10}$$

where β is an artificial compressibility parameter. Note that this method can only be used to obtain steady-state solutions. If one converts this equation and the momentum equations to the form given by Equation 1, the following relationships are obtained for a two-dimensional problem:

$$U = \begin{pmatrix} P \\ u \\ v \end{pmatrix} \tag{A.11}$$

$$F_x = \begin{pmatrix} \frac{u}{\beta} \\ u^2 + P - \frac{1}{Re} \frac{\partial u}{\partial x} \\ uv - \frac{1}{Re} \frac{\partial v}{\partial x} \end{pmatrix} \quad (\text{A.12})$$

$$F_y = \begin{pmatrix} \frac{v}{\beta} \\ uv - \frac{1}{Re} \frac{\partial u}{\partial y} \\ v^2 + P - \frac{1}{Re} \frac{\partial v}{\partial y} \end{pmatrix} \quad (\text{A.13})$$

where the source term is zero, Re is the Reynolds number and u and v are the flow velocities in the x - and y -direction respectively. In this case, writing the flux vector $\vec{F} = F_x \cdot \hat{n}_x + F_y \cdot \hat{n}_y$ allows some simplification:

$$\vec{F} = \begin{pmatrix} \frac{V}{\beta} \\ Vu + P\hat{n}_x - \frac{1}{Re} \frac{\partial u}{\partial n} \\ Vv + P\hat{n}_y - \frac{1}{Re} \frac{\partial v}{\partial n} \end{pmatrix} \quad (\text{A.14})$$

where $V = u \cdot \hat{n}_x + v \cdot \hat{n}_y$ is the flow velocity normal to a control volume boundary.

A.3.2 Boundary condition types

Fully-developed inflow This boundary condition type imposes a fully-developed normal velocity profile at the boundary. This condition can be determined from the wall-normal momentum equation which, at the inflow simplifies to:

$$\frac{\partial P}{\partial n} = \frac{1}{Re} \frac{\partial^2 u_n}{\partial c^2} \quad (\text{A.15})$$

Since u_n at the inlet is a known value, the condition on pressure can be computed. The user must specify both the normal velocity profile and the pressure gradient at the boundary. All conditions are imposed using constraints on the reconstruction.

Stationary wall At the wall, velocities u and v are zero, due to the no-slip condition. The boundary condition on pressure can also be determined from the wall-normal momentum equation, i.e.:

$$\frac{\partial P}{\partial n} = \frac{1}{Re} \frac{\partial^2 u_n}{\partial n^2} \quad (\text{A.16})$$

This second derivative of u_n is typically small and can be neglected. Note that for second-order accurate reconstruction, the second derivative of u_n would always be zero.

These conditions ($u = 0$, $v = 0$, and $\frac{\partial P}{\partial n} = 0$) are enforced using boundary constraints, and the interior flux is used on that boundary face.

Outflow For this boundary condition, the pressure is enforced to a value of zero. The boundary conditions also enforces fully-developed flow by constraining the tangential velocity and the normal gradient of the normal velocity to zero. These three conditions are imposed as constraints on the reconstruction.

A.4 Incompressible Energy Equation

A.4.1 Flux and source term

The incompressible energy equation can be solved in conjunction with the Navier-Stokes package above. However, the energy equation will have its own `Physics` package. Upon conversion to the form given by Equation 1, the energy equation is:

$$U = T \tag{A.17}$$

$$F_x = uT - \frac{1}{Re \cdot Pr} \frac{\partial T}{\partial x} \tag{A.18}$$

$$F_y = vT - \frac{1}{Re \cdot Pr} \frac{\partial T}{\partial y} \tag{A.19}$$

$$S = \frac{Ec}{Re} \left(2 \left(\frac{\partial u}{\partial x} \right)^2 + 2 \left(\frac{\partial v}{\partial y} \right)^2 + \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right)^2 \right) \tag{A.20}$$

where u and v are the flow velocities defined in Section A.3, Re is the Reynolds number, Pr is the Prandtl number, and Ec is the Eckert number. Once again, it is advantageous to define the flux normal to a control volume boundary $\vec{F} = F_x \cdot \hat{n}_x + F_y \cdot \hat{n}_y$. This results in:

$$\vec{F} = \vec{V}T - \frac{1}{Re \cdot Pr} \left(\frac{\partial T}{\partial n} \right)$$

where $\vec{V} = u \cdot \hat{n}_x + v \cdot \hat{n}_y$ is the flow velocity normal to a control volume boundary face and $\frac{\partial T}{\partial n}$ is the temperature gradient in the normal direction.

A.4.2 Boundary condition types

Outflow This boundary condition type does not impose any physical restriction on the temperature field, so the boundary flux is the same as the interior flux, i.e.

$$\vec{F}_b = \vec{V}T - \frac{1}{Re \cdot Pr} \left(\frac{\partial T}{\partial n} \right)$$

Imposed temperature This boundary condition is imposed by setting a constraint for the temperature value at the boundary. The boundary flux is simply $\vec{F}_b \cdot \hat{n} = F_x \cdot \hat{n}_x + F_y \cdot \hat{n}_y$.

Imposed heat flux The second boundary condition type imposes a specific heat flux q_n at a wall, i.e.

$$\vec{F}_b = \vec{V}T - \frac{1}{Re \cdot Pr} \left(\frac{q_n}{k} \right)$$

This boundary condition type can be used to specify an insulated wall by setting the normal heat flux to zero.