INFRASTRUCTURE FOR SOLVING GENERIC MULTIPHYSICS PROBLEMS

by

CHARLES BOIVIN

B.Eng. (Mechanical), McGill University, 1998

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(Department of Mechanical Engineering)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 2003

Department of MECHANICAL ENGINEERING

The University of British Columbia
Vancouver, Canada

Date 22/04/03

Abstract

Numerical simulations of partial differential equations problems are used in a variety of domains. Such simulation tools allow the scientific community to solve problems of increasing complexity. This allows complete testing and simulation of a product or process even before it is created. The numerical simulation process can be separated into two main steps: domain preparation and numerical computation. The first step requires the scientist to define the domain on which the problem will be solved; it is then decomposed into a group of smaller regions. This domain division is called a mesh. The mesh is subsequently used by the solver to perform the numerical computations specific to the physical problem being solved. The accuracy of the solution obtained depends on the quality of the mesh and the physical description of the problem.

As powerful and useful as they are, these numerical tools could be improved on two fronts. First, the time spent preparing a problem with a complex geometry for a simulation is sometimes very large and could be minimized by automation of the pre-processing steps. Second, numerical solvers are not used in all the problem domains where partial differential equation problems are encountered because of the difficulty in acquiring the numerical expertise needed to develop specialized solvers.

The objective of this research was to make the numerical simulation process easier and more accessible to scientists by addressing these two problems. Specifically, a mesh generator capable of generating guaranteed-quality meshes for complex geometries with curved boundaries has been written. This program completely automates the meshing process, which results in a huge gain in domain preparation efficiency. Additionally, an existing numerical toolkit has been modified to allow multiphysics problems to be solved in a generic fashion. With this solver, scientists can simply describe the physics of a problem — as well as the interactions between the different physical phenomena — and a numerical solution can be obtained within days. High-quality meshes and results from multiphysics problems are included to demonstrate the effectiveness of the current research. Finally, future improvements to the efficiency and accuracy of the solver are discussed.

Contents

List of Figures

List of Tables

Acknowledgements

This thesis benefited from the help of many, some of which I would like to thank here.

First of all, thank you to the faculty members in Mechanical Engineering, with a special mention to the faculty from the thermo-fluids group. I learned a lot from your teaching and feedback. Thank you as well to my research committee for the helpful suggestions. Special thanks go to my research supervisor, Dr. Carl Ollivier-Gooch. Carl, thank you for your time, your patience, and your advice. I have learned so much from you and yet I know I could still learn much more. I have nothing but praise for you as a research supervisor and I know many more students will benefit from your experience.

Thank you to friends I made during my stay at UBC: you all made this whole experience more enjoyable. It was fun to snowboard, party, play poker, eat pizza, and yes, even go curling with all of you. To my guitar buddies: thanks for showing me yet another thing I can do rather than work on my research. Hopefully, we will get together again soon to play.

I would also like to thank my family who have always supported me in everything I do: my sisters, and my parents, Pauline et Jean-Marc: merci beaucoup pour tout le support. Vous me manquez beaucoup et j'ai hâte de vous voir. And finally, a huge thank you to Roxanne, the woman who became my wife during this journey. I could not have accomplished this without you. Thank you for all the support and understanding, for letting me use the computer to run tests even though I said I would not need to do it, and for listening to me complain about boundary conditions whenever the code would not converge. Thank you, most importantly, for believing in me. Je t'aime.

Charles

Chapter 1

Introduction

Numerical simulations of partial differential equations problems are used in a variety of domains such as aerospace research, combustion simulation, and medical research to name a few. These simulation tools allow the scientific community to solve problems of increasing complexity. This leads to an increase in the efficiency of the design and engineering processes by allowing complete testing and simulation of a product or process even before it is created.

The numerical simulation process can be separated into two main steps: domain preparation and numerical computation. The first step requires the scientist to define the domain on which the problem will be solved. Once the boundaries of the domain have been defined, the domain must be decomposed into smaller, simpler regions called cells. This domain division is called a mesh. The mesh is then used by the solver for the second part of the process in which the numerical computations specific to the physical problem being solved are carried out. The accuracy of the solution obtained depends on the quality of the mesh and the physical description of the problem.

As powerful and useful as they are, the numerical tools available to scientists could be improved. The time spent preparing a problem with a complex geometry for a numerical simulation is sometimes very large and could be minimized by proper automation of some of the pre-processing steps. A particularly complex example given by Mavriplis [31] showed the mesh preparation time to be 45 times that required to compute the solution. This is unacceptable, especially considering the fact that the mesh is an artifact of the numerical simulation and is of no intrinsic physical significance. There are huge potential gains to be made by fully automating the meshing process.

On the other hand, numerical solvers are not used in all the problem domains where partial differential equation problems are encountered. In some areas, scientists know a large amount about the important physical processes they study, but are unable to perform numerical simulations due to the lack of available solvers or knowledge of numerical methods. Problem-domain experts must therefore either acquire (or hire) the numerical expertise needed to write a solver specialized to their problem, or wait for a commercial package to be available. This is even more complicated when multiphysics problems — problems where multiple physical phenomena interact with each other — are being investigated. A better alternative for these scientists would be to 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 aspect of the simulation. Solutions to new and complex physical problems could then be obtained within days rather than months.

The objective of this research was to make the whole numerical simulation process easier and more accessible to scientists by addressing these two problems. Specifically, I have written a fully-automatic mesh generator that can generate guaranteed-quality meshes for complex geometries. More details and background information on the mesh generator are given in Section 1.1. I have also modified an existing high-order generic numerical toolkit to allow complex, multiphysics problems to be solved. Further information about this research and other related work is presented in Section 1.2.

1.1 Unstructured mesh generation

A fully-automatic mesh generator for complex geometries must understand curved boundaries to prevent geometric errors at the boundaries and to correctly resolve boundaries based on their extent and curvature. This is especially critical given that most problems in computational science are boundary value problems and require accurate boundary information to yield an accurate solution. Equally important is mesh quality, which affects the convergence rate and solution accuracy [25, 3, 24]; an automatic meshing process therefore requires some guarantees on mesh quality. Furthermore, automatic mesh generation also requires guarantees on termination and final mesh size. These guarantees ensure that a guaranteed-quality mesh *can* be obtained, and that its size will be within a certain factor of the ideal mesh size for that problem, respectively. The long-term goal for developers of meshing tools is the generation of appropriately sized quality meshes directly from CAD models, without user interaction. This research is an important step in that direction.

1.1.1 Mesh generation from curved boundaries

A fully-automatic mesh generator must handle curved surfaces as readily as planar ones, which requires the use of the exact representation of the boundaries during the meshing process [39]. Otherwise, time is wasted discretizing these curves into sets of linear segments, a process which can also lead to an invalid representation of the boundary. For example, more cells may be necessary to properly discretize a curved boundary than the user anticipated. Because the mesh generation package at this point only relies on the linear segments, it has no knowledge of the real shape of the boundary. It can only place new vertices on a line joining two of the original discretized vertices, as was done by Mavriplis [30]. The newly inserted vertices are usually only moved back to the boundary as a post-processing step; while this is not usually extremely time-consuming, it can potentially degrade mesh quality near the boundary or even make the mesh invalid. On the other hand, if the user (or the software) over-estimates the number of vertices necessary along a curved boundary, more cells than required will be present in the mesh, which will affect simulation performance.

A better approach is to insert points directly on the boundary curves in the first place, using the underlying representation of the boundary. Tools capable of generating meshes for domains with curved boundaries are now relatively common, although each seems to handle curves differently. The advancing front Delaunay schemes [29, 23] initially discretize the boundaries and the few layers of cells surrounding them using advancing front techniques, and mesh the interior of the domain with a Delaunay approach. Another example is the 2D scheme described by Laug et al. [27], which relies on a mesh to extrapolate a curved boundary using interpolation splines. These splines are approximated next with a very large number of linear segments. Points on the boundary are then chosen on these segments whenever needed in the meshing process. Conversely, the 3D algorithm by Dey et al. [15] uses the curved representation of the boundary directly to generate extra boundary vertices and to detect possible problems, such as intersection problems. However, there are no guarantees regarding the quality of the mesh, or even the termination of any of these algorithms.

1.1.2 Guaranteed-quality mesh generation

Users of guaranteed-quality meshing tools only need to define the domain properly, and perhaps indicate a preference on the resolution required. A good mesh can then be obtained without any further user interaction. The user never needs to fix areas containing an invalid triangulation or poor quality elements.

Several guaranteed-quality algorithms have been introduced in recent years. Chew [13] introduced the first two-dimensional Delaunay insertion algorithm with a quality bound, although it only generated uniform meshes. Ruppert [38] then introduced the first Delaunay insertion scheme to guarantee high-quality two-dimensional graded meshes. Shewchuk [40] improved the angle bound of Ruppert's scheme shortly after and proved that such a modification made the algorithm equivalent to another by Chew [14]. All of these schemes insert points at the circumcenters of triangles; other authors have proposed variations on the circumcenter as the location of point insertion. Rivara [37] suggested inserting a vertex at the midpoint of the common edge of the two terminal triangles of a set of triangles called the longest-edge propagation path. More recently, Edelsbrunner and Guoy [17] proposed inserting points at *sinks*, circumcenters located inside their own triangles. Shewchuk also introduced a generalization of Ruppert's algorithm to three dimensions which showed significantly better quality bounds than a previous 3D algorithm by Mitchell and Vavasis [32]. In previous research [35], Ollivier-Gooch and Boivin extended Ruppert's and Shewchuk's work to have better control over cell grading and size, in both 2D and 3D. The common downfall of these guaranteed-quality schemes is that they all require the domain to have linear (or planar) boundaries.

1.1.3 Objective

The objective for this part of my research was to develop a guaranteed-quality unstructured mesh generator with generic support for curved boundaries; this is a major step towards guaranteed-quality mesh generation from CAD data. In Chapter 2, a generic boundary data interface that allows the use of potentially *any* boundary type, in both 2D and 3D is introduced. As a proof of concept, this interface has been implemented in a 2D meshing code; examples are also given in Chapter 2. Ruppert's proof has been extended to account for curved boundaries. Details of the proof are given in Appendix A. Most of Chapter 2 has already appeared as a journal article [9].

1.2 Generic multiphysics solver

The second part of this research focuses on writing a generic, high-order, multiphysics solver. A few other research groups have developed generic numerical toolkits in the hope of easing the solution of new physical problems numerically. A summary of these efforts is presented in Section 1.2.1. Most of the work from these research groups focuses on the finite-element method; the numerical toolkit used in this research is based on the finite-volume method, and Section 1.2.2 justifies this choice.

1.2.1 Generic solvers

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. Very often, this modularity comes from using object-oriented programming techniques in the numerical toolkit, as described by Dubois-Pèlerin and Pegon [16]. A discussion on the impact of using object-oriented languages in scientific applications is given by Arge et al. [2]; some optimization guidelines are also suggested. Most of the generic numerical toolkits described in this section use some form of object-oriented language; the toolkit used in this research follows this trend and is coded in C++.

One application of modularity is the research of Eyheramendy and Zimmerman [44, 18, 19] who have developed a toolkit for semi-automatic symbolic derivation of linear finite-element models of initial-boundary-value problems. In more recent work [20], non-linear problems are also supported. The software, developed in SmallTalk, guides the user through the various steps necessary to formulate the finite-element problem. Smalltalk or C++ simulation code can then be automatically generated, although the authors admit that efficiency problems within the generated code still remain. 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 [12, 26, 11]. This library allows the user the flexibility to interact with the finite-element solution process at different levels. At the highest level, users specify the physics of a problem through the integrand of the weak form of the PDE and boundary conditions. At lower levels, users can interface with the numerical techniques by defining new element types, shape functions, or by setting the options for the numerous iterative solvers supported. Again, significant finite-element knowledge is necessary when creating custom simulation in Diffpack. In particular, the boundary conditions must be explicitly coded by the user 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 [1]. As its name implies, FEMLab is also based on the finite-element method. The solver uses Matlab for the numerical aspects of the simulation. Multiple element types and physical phenomena can be combined in one simulation. 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.

While being valuable tools for many researchers, these approaches are likely too complicated for most scientists lacking a good finite-element background, especially when trying to develop simulations for physical phenomena that are uncommon.

PHYSICA [4] is a multiphysics solver based on the finite-volume method. It also uses object-oriented programming techniques to provide abstraction of the numerics and the physics of a problem; it is programmed in FORTRAN. 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.

1.2.2 The finite-volume method

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. This is explained by the fact that the physics of the problem mainly come in play in the calculation of fluxes. These fluxes are straightforward to identify and can be computed easily. More details on the derivation of a new problem for the finite-volume method are given in Section 3.1.

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.

1.2.3 Objective

The objective for this part of my research is to develop a high-order accurate generic multiphysics solver. The solver is based on a high-order generic numerical toolkit introduced in Chapter 3. High-order accurate methods are critical for better solutions of a variety of problems. The toolkit was then heavily modified to support multiphysics problems; the multiphysics framework is covered in Chapter 4. Chapter 5 introduces topics that are related to the multiphysics solver but did not play a large role in the design of the framework and Chapter 6 discusses the accuracy and efficiency of the solver, and the steps that could be taken to further improve both.

Chapter 2

Guaranteed-Quality Unstructured Triangular Meshing of Domains with Curved Boundaries

2.1 Introduction

In this chapter, a major step towards guaranteed-quality mesh generation from CAD data is described. A generic boundary data interface that allows the use of potentially *any* boundary type, in both 2D and 3D is introduced, and is implemented in 2D. First, some basic unstructured meshing concepts are introduced in Section 2.2. The 2D scheme presented in this research is based on Ruppert's Delaunay refinement algorithm, which is summarized in Section 2.3 in order to better highlight the changes made to it in Section 2.4. In the latter section, the generic interface between the meshing code and arbitrary boundary data is described, and some of the pitfalls associated with curved boundaries are noted. Boundary representations for line segments, circles and circular arcs, cubic parametric curves (which include Bézier cuves and B-splines), and interpolated splines have been implemented. Details of their implementation are given in Section 2.5. Examples are presented in Section 2.6 to demonstrate the capabilities of the algorithm. Conclusions about this work and insight on future implementation in 3D are discussed in Section 2.7. Finally, it has been possible to show that the modified scheme produces meshes of same quality as Ruppert's original scheme, including improved control over length scale and grading, as described in [35]; a proof of this result appears in the Appendix.

2.2 Background information

The methodology used to start a triangulation in this research will be covered in this section. Along the way, concepts critical to this research, such as the Delaunay criterion, will be explained.

2.2.1 Initial vertex insertion

As an example, the domain shown in Figure 2.1 is to be triangulated. This is accomplished by first creating a large bounding box in which all the boundary vertices of the domain can fit. These vertices are then inserted in the bounding box. Vertex insertion is covered in more detail in Section 2.3.3.

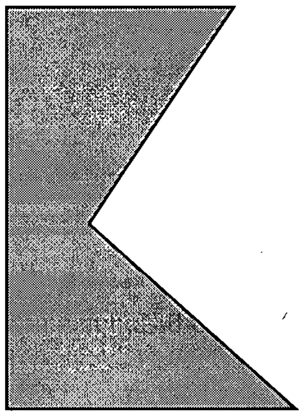


Figure 2.1: Domain to be triangulated

The initial bounding box, and the resulting triangulation after the boundary vertices have been inserted are shown in Figure 2.2.

2.2.2 The Delaunay criterion

Next, edges in the triangulation are swapped to ensure the triangulation is Delaunay. The Delaunay criterion states that no vertices can be located within a triangle circumcircle.[1] It is always possible to make a triangulation Delaunay by swapping. An example of a triangle with a vertex in its circumcircle is shown in Figure 2.3.

[1]A circumcircle is the circle that passes through each of the triangle's vertices.

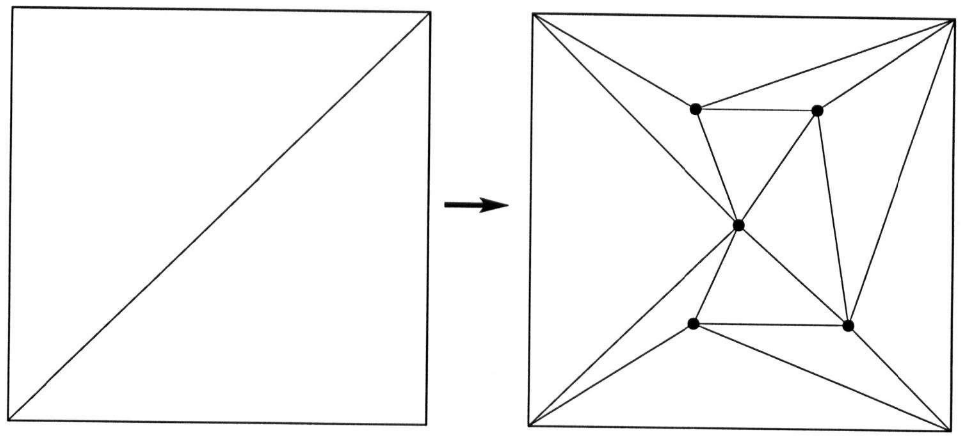


Figure 2.2: Initial bounding box and boundary vertices

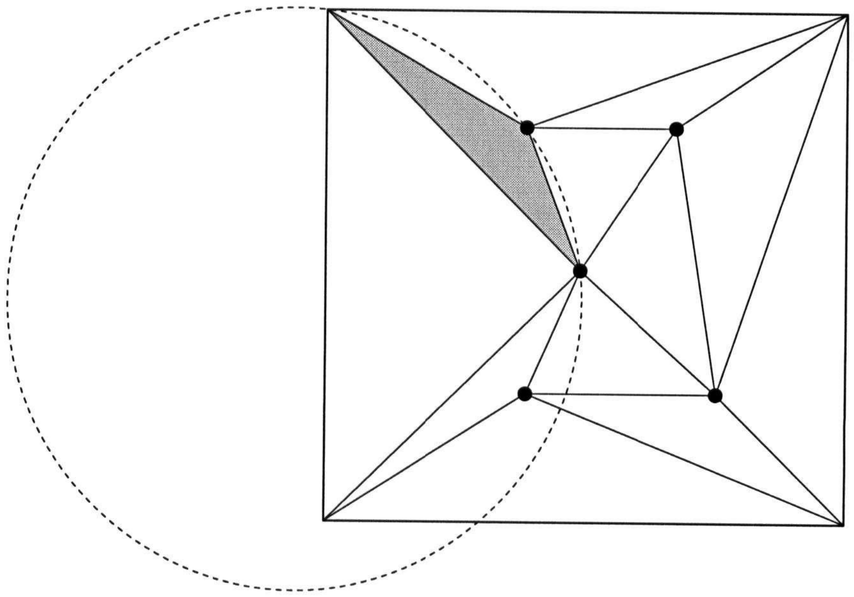


Figure 2.3: Triangle with a vertex in its circumcircle

This triangle cannot exist in a Delaunay triangulation. Figure 2.4 shows the Delaunay triangulation for this domain.

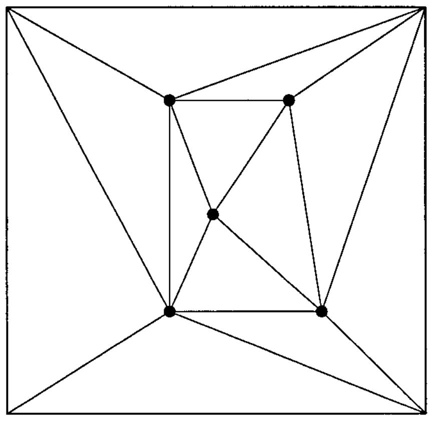


Figure 2.4: The Delaunay triangulation

2.2.3 Edge recovery

The next step consists in recovering all the boundary edges of the initial domain. In some cases, it is possible for the Delaunay triangulation not to include the boundary edges. However, these edges can *always* be recovered through swapping. The boundary edges are already all present in Figure 2.4.

With all the boundary edges present in the triangulation, the surrounding cells of the bounding box can be safely removed. The resulting triangulated domain is shown in Figure 2.5. This triangulation can then be used with Ruppert's Delaunay refinement algorithm.

2.3 Outline of Ruppert's Delaunay refinement algorithm

Ruppert's scheme [38] begins with either a constrained or unconstrained Delaunay triangulation.[2] The mesh quality is improved through point insertion. Points are inserted at the circumcenter of badly-shaped cells — cells that have an angle less than $\theta_{\min}$ — unless they *encroach* on a boundary

[2]A constrained Delaunay triangulation is a triangulation in which the Delaunay criterion is only applied to vertices that are visible to a triangle. A vertex is visible to a triangle if there are no boundary patches between them.

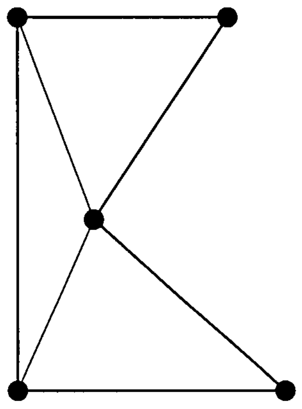


Figure 2.5: The triangulated domain

edge.[3] Ruppert states that a vertex encroaches on an edge when that vertex is located inside the circle with the edge as its diameter; this circle is called the *diametral circle.* If a proposed new point encroaches on any boundary edge, that vertex is not inserted. Instead, the encroached boundary edge(s) is(are) bisected. This process is repeated until all cells are well-shaped. Ruppert was able to show that this algorithm always terminates, and results in a mesh with minimum angle $\theta_{\text{min}} \approx 20.7°$.

Shewchuk [40] showed that a value of θ_{min} of 25.7° is possible if *diametral lenses* rather than diametral circles are used to determine if there is encroachment. The difference between the diametral circle and diametral lens is shown in Figure 2.6. In this variant of the algorithm, interior vertices lying inside the diametral circle of a boundary edge are deleted when that edge is split. The bound on θ_{min} is not tight; in practice, θ_{min} can be set to 30° and the algorithm will still terminate.

2.3.1 Initial discretization

Ruppert's algorithm can be started either with a Delaunay triangulation or a constrained Delaunay triangulation. The latter does not pose a problem because Ruppert's original encroachment rule guarantees that no vertex will be inserted outside a boundary edge.[4] A Delaunay triangulation containing all the boundary points inside a larger bounding box is first created. Boundary edges

[3]Note that in this document, a *boundary edge* is what makes up the discretized version of the boundary. When referring to the boundary geometry, the term *boundary patch* will be used. Each boundary patch has at least one boundary edge associated with it.

[4]The use of diametral lenses allows boundary triangles with a circumcenter outside the boundary edge to be present in the mesh. However, no vertex will ever be inserted at this location since it encroaches on the boundary edge.

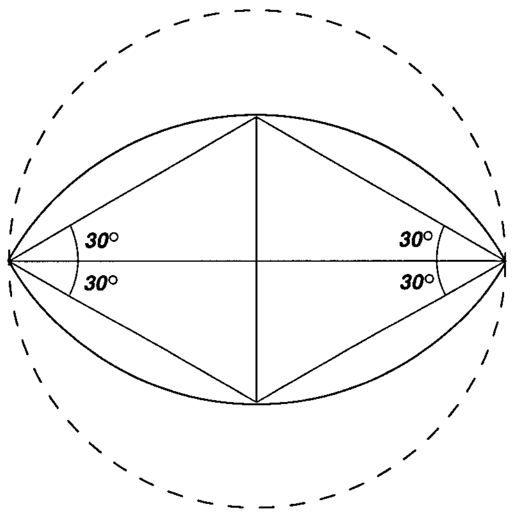


Figure 2.6: Comparison between a diametral circle (dashed) and diametral lenses. Diametral lenses allow points to be inserted closer to boundary edges.

are recovered next using the technique described in Section 2.3.2. The triangles lying outside the domain are then removed, leaving a constrained Delaunay triangulation.

The algorithm cannot be started with just any constrained Delaunay triangulation, however. No boundary edges in the initial triangulation should be encroached on. Encroached boundary edges are therefore split until they are not encroached upon anymore, as an initialization step. This is done by evaluating the angle opposite the boundary edge. If the angle is obtuse, the vertex at that corner encroaches on the boundary edge and the edge should be split. This way, only the encroachment caused by vertices visible to the boundary edge will be corrected, preventing unnecessary splitting of boundary edges and introduction of artificial small features. When no more vertices encroach on boundary edges, Ruppert's algorithm can be started.

2.3.2 Edge recovery

The boundary edges needed for the initial discretization of the boundary are recovered through swapping. It is always possible to recover all the edges without having to insert new points in the domain. Once all the boundary edges have been recovered, the boundary representation is exact.

2.3.3 Point insertion

Points are inserted into the mesh by using the Delaunay insertion method of Watson [42]. A list of all cells that contain the new vertex in their circumcircle is first created. These cells are then removed from the mesh, and the faces of the resulting hull are connected with the newly inserted point. This insertion method preserves the Delaunay nature of the mesh; no swapping is needed after the insertion. If a boundary edge is part of the hull, a check is made to ensure that the new vertex will not encroach on it. If it does, the point is not inserted. Vertices lying inside the diametral circle of the edge are removed, and the boundary edge is split at its geometric midpoint. Watson insertion is used for this split as well.

2.3.4 Length scale modifications

In previous work [35], Ollivier-Gooch and Boivin modified Ruppert's scheme to control cell size and grading. The modification defines a geometric length scale based on the *local feature size.* The local feature size was used by Ruppert to prove termination of the original algorithm, and is defined as the radius of the smallest circle centered at a point that touches two disjoint parts of the domain boundary. The length scale LS is defined in terms of the local feature size lfs as:

$$LS\,(p) = \min\left(\frac{lfs\,(p)}{R}, \min_{neighbors\, \mathrm{q}_i} LS\,(q_i) + \frac{1}{G}\left|\vec{q_i} - \vec{p}\right|\right) \tag{2.1}$$

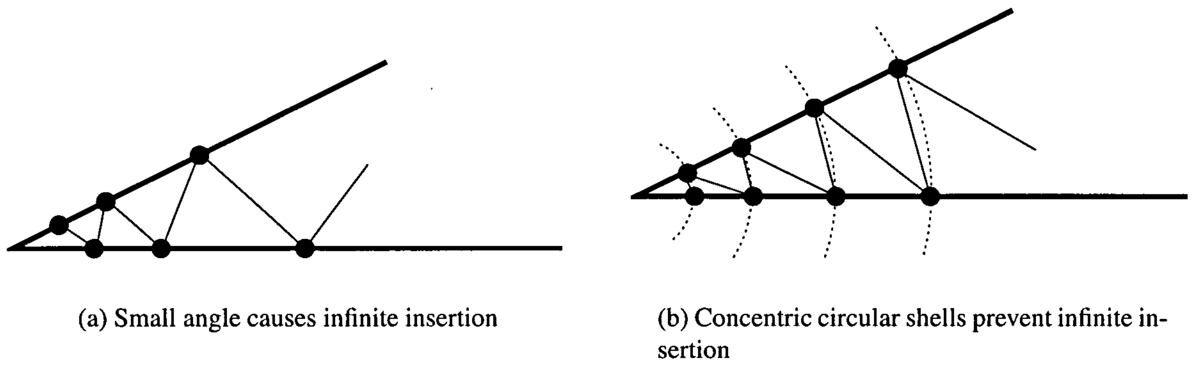
where both R and G are constants ≥ 1, and points q_i are neighbors to point p. The first constant, R, controls the ratio of input feature size to final mesh boundary edge length, with finer boundary discretization for larger values of R. The other constant, G, is used to control how rapidly the cell size can change with distance. This is an explicit imitation and generalization of the grading properties of the local feature size. A larger value of G results in slower increase in cell size over the same distance. The value of LS is stored at every vertex location.

Ruppert's scheme was modified to also split cells that are too large according to the definition of length scale in Equation 2.1. A cell is considered too large whenever the ratio of its circumradius to the average LS of its vertices is greater than $\frac{\sqrt{2}}{2}$.

Implementation details, such as how the lfs is computed, as well as a proof that the modified algorithm terminates with quality bounds comparable to Ruppert's are provided in [35].

2.3.5 Small angles in the domain

Small angles in the domain definition are problematic because they can lead to infinite recursion when trying to fix the encroachment of a boundary edge. Ruppert [38] identified this pitfall and suggested the use of *concentric circular shells* around small angles to prevent it. Figure 2.7 illustrates this. Boundary edges that are connected to a small angle vertex are split at the intersection with circular shells centered at the vertex — not at the midpoint of the edge. This has the effect of creating protective layers around the small angle boundary vertex, preventing encroachment. This technique was also used in the present research.


(a) Small angle causes infinite insertion

(b) Concentric circular shells prevent infinite insertion

Figure 2.7: Problem caused by small angles in the domain and how it can be avoided.

2.4 Generic boundary interface

To enable meshing from general curved boundaries, a framework in which the mesh generation code makes no assumptions about the underlying geometry of boundary patches is needed [39]. This implies a generic interface between mesher and geometry, in which the mesher only needs the results of several geometric queries. This is illustrated in Figure 2.8.

Whenever the mesh generation algorithm needs information about the boundary, a "question" is passed on to the proper type of boundary patch. Each boundary patch type knows how to answer all of these questions, and the answer is then passed back to the algorithm. This provides a transparent access to potentially any type of boundary patch. Using object-oriented programming, this generic interface can be implemented by using a common base class for all boundary data, with implementation of specific geometric queries in derived boundary data classes.

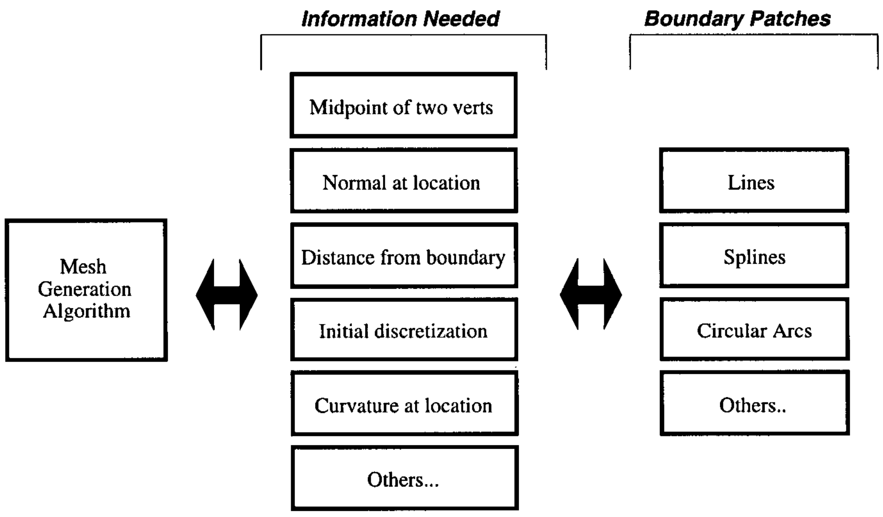


Figure 2.8: Framework used for the implementation of generic boundaries

The information required for the successful implementation of Ruppert's algorithm — curve midpoint, curvature, and original discretization information — is described in Sections 3.2 to 3.6. The other questions are needed to determine the appropriate mesh length scale LS; they will not be discussed any further in this text. See [35] for more information.

So far, classes for lines, circles, arcs, cubic parametric curves, and interpolated splines have been written. New types of boundary patches can be added by providing the proper "answers" for the given boundary patch.

2.4.1 Total variation of the tangent angle

Since the meshing code must be able to work with curved as well as linear patches, a new way of determining where splits happen along a boundary patch is necessary. One can first observe that patches with little orientation change need few, long edges for accurate geometric representation. Linear patches have no orientation change; they can be represented accurately with just one edge. In contrast, regions of a curve with a large change in orientation require a greater number of shorter edges. One must also make sure that small amplitude sine-like curves are discretized appropriately. This suggests the use of the total variation of the tangent angle of a curve to determine where to split a boundary patch.

The total variation $TV(\theta)$ is defined in the following way:

$$TV(\theta) = \int \mid d\theta \mid \tag{2.2}$$

By using the following definition of curvature:

$$\kappa(s) = \left|\frac{d\theta}{ds}\right|$$

it is possible to obtain another form for Equation 2.2:

$$TV(\theta) = \int |d\theta| = \int |\kappa(s)| \; ds$$

The total variation can therefore also be expressed as the integral of the absolute value of curvature along the arclength. Note that there is no need to compute the integral; one simply needs to compare the orientation of the curve's tangent vector at carefully chosen points along the boundary patches to get the exact value of $TV(\theta)$. More details are given for each type of boundary patches in Section 2.5.

2.4.2 Initial discretization

To obtain the initial Delaunay triangulation, each boundary patch must be initially discretized in some way. Since the exact shape of the boundary is only known by the boundary patches, the initial discretization of the corresponding curve must be computed by the patches themselves. At this point in the meshing process, curves are represented with as few edges as possible in order not to introduce artificial small features in the mesh. However, one must make sure that a valid and exact representation of the domain will be obtained and that the rules regarding the location of points inside the diametral lenses are also followed.

An arbitrary discretization of a spline curve is shown in Figure 2.9. The outside (above) of the curve is to be triangulated. Ruppert's scheme guarantees that no vertex will be inserted inside (below) the boundary edges. One must also make sure that no vertex will be inserted in the regions inside the curve but outside of the boundary edges (the shaded area in Figure 2.9). This is to prevent an invalid discretization, as the vertex inserted in the shaded area would ultimately lie outside the domain once the boundary is well-resolved.

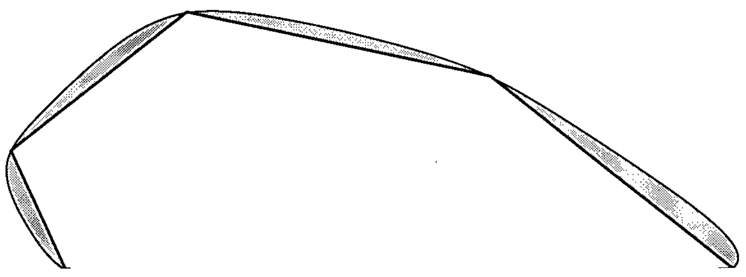


Figure 2.9: Arbitrary original discretization of a spline. No vertex should be inserted in the shaded areas.

The protection of this area can be achieved by making sure that the diametral lenses of the boundary edges completely include the curve boundary. Since points are never inserted inside the diametral lenses, this will protect the shaded region from point insertion. It is easy to calculate the total variation in orientation a curve can have to remain inside the diametral lens of a corresponding discretized edge.

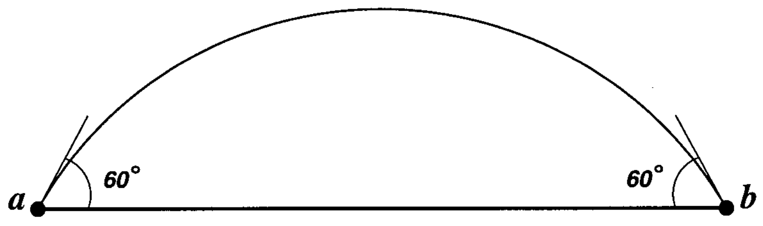


Figure 2.10: Diametral lens of edge *ab* intersects the edge at an angle of 60°.

The diametral lens, as seen in Figure 2.10 makes a 60° angle with edge *ab*. A curve passing through both points *a* and *b* can make an angle of $60° - \epsilon$ with the horizontal at point *a* and an angle of $-\epsilon$ with the horizontal at point *b* and still be completely inside the diametral lens. This results in a $TV(\theta)$ of $\pi/3$. This is the maximum total variation in orientation a curve can have in order to pass through both points *a* and *b,* and still remain inside the diametral lens. A bigger change can *potentially* put the curve outside of the lens. A valid initial discretization scheme must therefore limit the length of edges so that the $TV(\theta)$ of the curve over them does not exceed $\pi/3$, i.e. $TV(\theta)_{\max} = \pi/3$. The diametral lenses of all the boundary edges will then entirely contain their corresponding boundary patch, therefore not allowing any vertices to be inserted in the shaded areas of Figure 2.9.

In addition, whenever a new boundary point is inserted, one must make sure that the two newly created boundary edges will have diametral lenses that are point-free to prevent insertion outside the domain. In Shewchuk's modification to Ruppert's scheme, all points in a boundary edge's diametral circle are deleted before the edge is split; the same is done in this scheme.

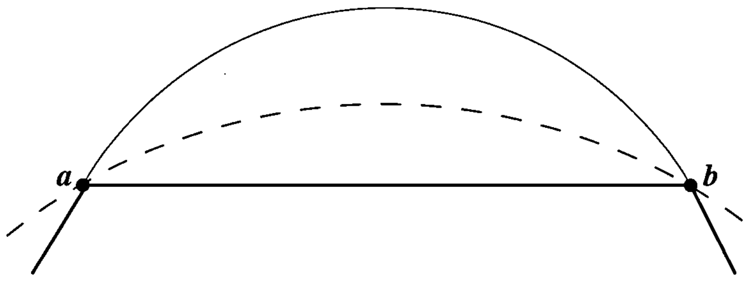


Figure 2.11: A curve with uniform curvature intersects the edge with angles of 30°.

As can be seen from Figure 2.11, a curve with uniform curvature will intersect the edge with angles of 30° at each endpoint. Knowing that the diametral circle of the original boundary edge is always point-free, it is easy to see that the diametral lenses of new boundary edges coming from this curve will also be point-free. The diametral lenses will always be contained within the diametral circle. Such a statement is not true, however, for non-uniform curvature patches. For such curves, the incident angle with the boundary edge can be arbitrarily close to 60°. This could result in a diametral lens that is not entirely contained within the diametral circle whenever that curve needs to be further split, as illustrated in Figure 2.12. The area with a white background is point-free whereas the area with a shaded background might contain points. It can be seen that part of the new diametral lens lies in the shaded area.

This pitfall is avoided by limiting $TV(\theta)_{\max}$ along a boundary edge to $\pi/6$ for boundary patches with non-uniform curvature. While this leads to twice as many boundary edges needed for curves with non-uniform curvature compared to uniform-curvature patches, this representation is still coarse enough not to introduce any artificial small feature in the mesh. More details are given in Section 2.4.5.

The general scheme for the original discretization of the boundary patches is therefore to first calculate the orientation change over the complete patch. The number of edges is then found by ensuring that each edge, when split at equal intervals of $TV(\theta)$, will cover less than the maximum allowed for a given type of boundary patch (i.e $\pi/3$ for patches with uniform curvature, $\pi/6$ for patches with non-uniform curvature). The following formula can be used for the number of edges:

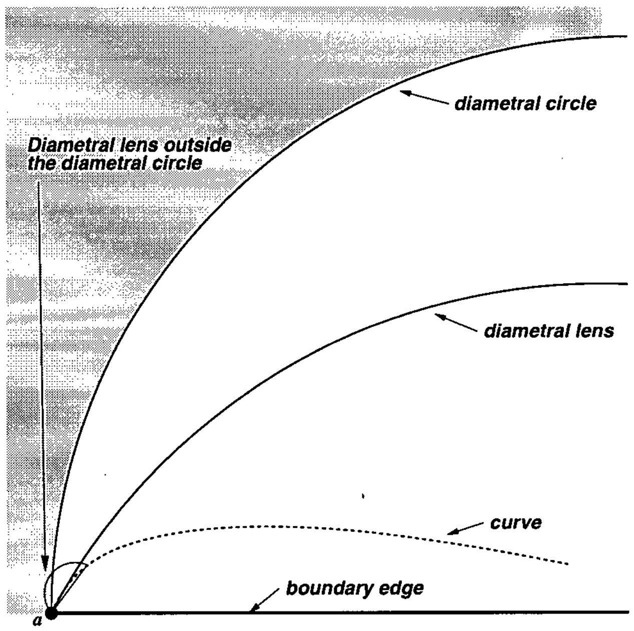


Figure 2.12: A curve with non-uniform curvature might yield a diametral lens outside the diametral circle.

$$N_e = \left\lceil \frac{TV(\theta)}{TV(\theta)_{\max}} \right\rceil \tag{2.3}$$

The new vertices will be located where the orientation change from the previous vertex is:

$$TV(\theta)_e = \frac{TV(\theta)}{N_e}$$

2.4.3 Edge recovery

Due to the very coarse representation of the boundary patches during edge recovery, some precautions must be taken in order to get a valid initial constrained Delaunay triangulation. The edge recovery process must be modified since simple recovery through swapping will fail in some cases. Two categories of such cases have been found. A description of these and an overview of the edge recovery strategy used to obtain valid initial triangulations follow.

Crossing of initial discretization edges

The initial discretization suggested by the boundary patches may result in an invalid overall discretization, because the edges that need to be recovered cross each other. Such a case is shown in Figure 2.13, which presents a square inside a circle, including the initial discretization of the circle. The top and bottom edges of the circle's discretization cross the edges of the square. Clearly, not all edges in this initial discretization can be recovered simultaneously.

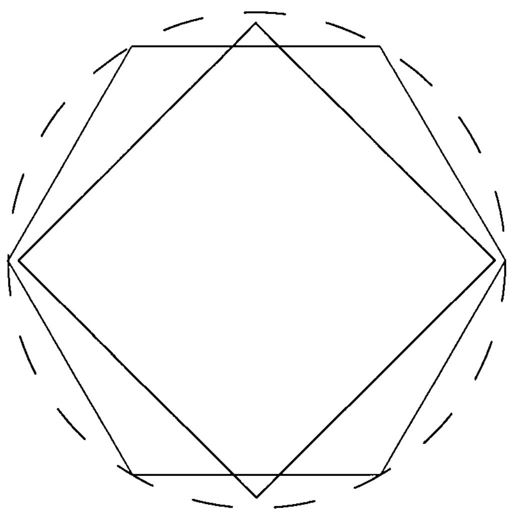


Figure 2.13: Example of an invalid initial discretization

Detecting such cases in advance can be computationally expensive. Instead, it is possible to take advantage of the fact that edges can be recovered through vertex insertion, a process known as *stitching*. If an edge is not present in the mesh, a vertex is inserted at the midpoint of its corresponding patch. If any of the two resulting edges is still absent from the mesh, then it is once again split. This method is guaranteed to recover all the edges since a vertex is always connected to its nearest neighbors in a Delaunay triangulation [40]. The spacing between the vertices of a boundary patch will eventually be small enough that the corresponding boundary edges will have to be present in the triangulation.

However, when forming a constrained Delaunay triangulation, blindly inserting vertices for a missing edge can lead to a very large number of unnecessary vertices. Consider for example the domain presented in Figure 2.14. Since vertex *a* is so close to edge *bc*, many vertices would need to be inserted on edge *bc* in order to recover the edge. This would lead to an artifical small feature in the triangulation. Obviously, this is to be avoided.

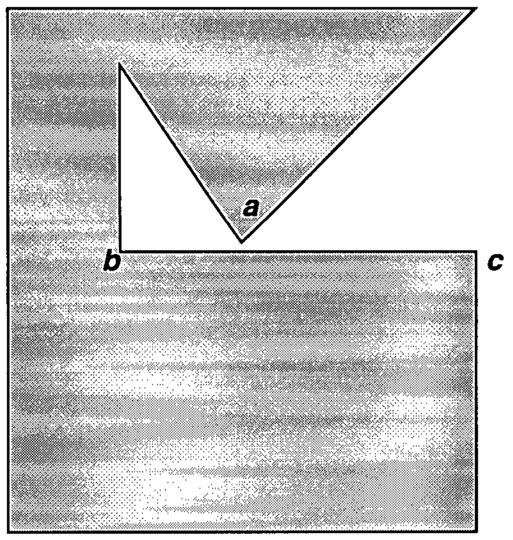


Figure 2.14: Vertex *a* should not generate a small feature on edge *bc*

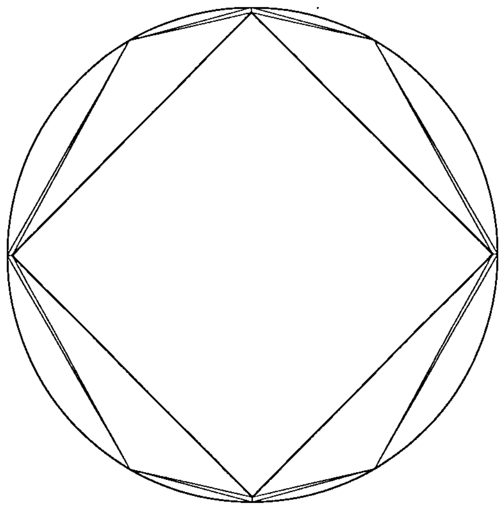


Figure 2.15: Initial discretization of a domain that had overlapping initial boundary edges

These competing requirements are balanced by inserting vertices only when swapping has failed. The mesh generator first goes through the list of edges and recovers them through swapping. Since the edges are not locked once they are recovered, it is possible that the recovery of one edge makes a previously recovered edge disappear. Any edge associated with a curved patch that is still missing after this step will have a vertex inserted at its corresponding midpoint. Recovery of edges associated with linear patches is always done through swapping. By following this method, only the necessary vertices are inserted and no artifical small feature is introduced in the constrained triangulation. The initial discretization for the case described in Figure 2.13 is shown in Figure 2.15.

Boundary edges located in wrong region

It is also possible that, due to the rather coarse discretization of curved boundary patches, entire boundary edges will be located in the wrong region. This problem has the same source as the previous one, except that in this case, the boundary edges do not overlap. In such a case, all boundary edges can be recovered, but the initial discretization is still invalid.

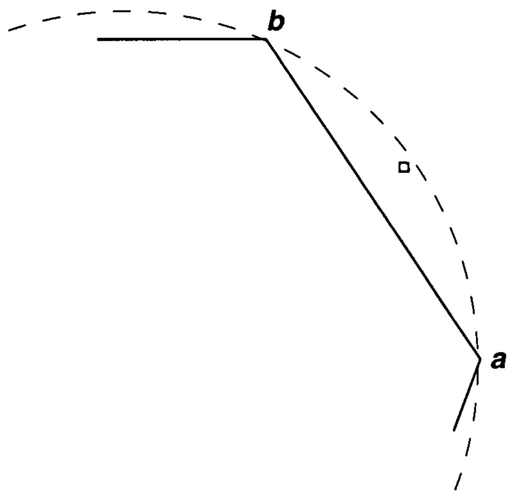


Figure 2.16: Example of a feature of the mesh (the small square) that is located in the wrong region due to the discretization of the curved boundary patch

Figure 2.16 illustrates this. The small square is located to the right of the boundary edge *ab*. However, it is located to the *left* of boundary patch *ab*. In this case, the small square would be located *outside* the triangulation, which can not be allowed.

The easy way to detect this case is to make sure that a vertex connected to edge *ab* on the *left* side is also on the *left* side of the corresponding curved boundary patch. If the two sides are different, then edge ab is split. This check must be done for both vertices located opposite each edge in the mesh associated with non-linear boundary patches.

As a summary, Figure 2.17 shows a diagram of the procedure to follow for edges to be recovered. The process is over once all the edges are recovered in one pass.

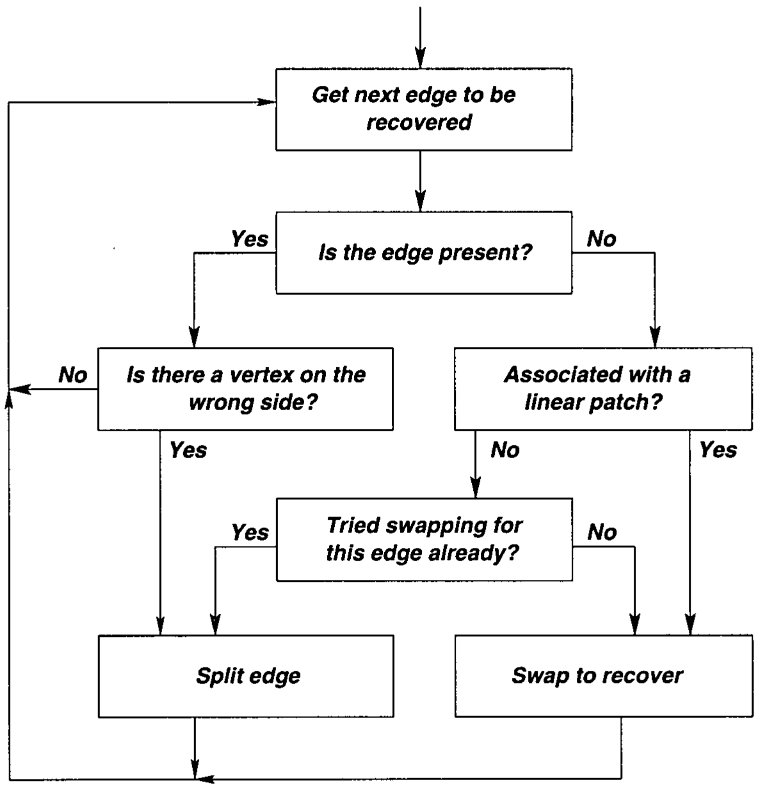

Get next edge to be recovered
Is the edge present?
Yes
No
Is there a vertex on the wrong side?
No
Yes
Associated with a linear patch?
No
Yes
Tried swapping for this edge already?
Yes
No
Split edge
Swap to recover

Figure 2.17: Procedure to follow to recover boundary edges

2.4.4 Point insertion

Point insertion in the mesh, as well as on the boundary, is still done using Watson's method. However, curved boundaries modify the way that boundary edges are split. Instead of splitting at the average location of the edge's vertices, the location of the new boundary vertex is determined by the boundary patch itself. The "midpoint" between two vertices is now found using the total variation of the tangent angle. The general technique is to first find the total variation of the tangent angle between the boundary edge's vertices a and b. The midpoint c will be located at the point on the curve where $TV(\theta)$ between a and c and between c and b is equal. This ensures that the new point is always located on the boundary and that regions of the curve with higher curvature will be discretized with more edges.

If the curvature over a given boundary edge is (almost) zero, the orientation change is negligible. In these cases, the split is made according to arclength. This ensures linear patches are split in the same fashion as before, and it also handles curves that have particularly flat regions.

The fact that the midpoints are no longer always located on the boundary edge being split can lead to problems. In some cases, boundary edges may cross nearby boundary patches. An example of such a case is shown in Figure 2.18. If the first edge of the bottom arc AB happens to be split before the first edge of the top arc CD, point E will be inserted outside the domain, which can not be allowed.

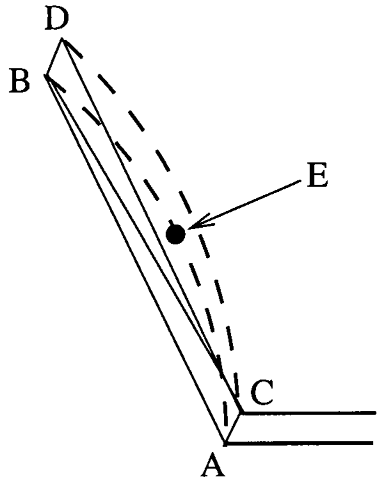


Figure 2.18: The top arc's discretization crosses over the bottom arc, but does not cross the bottom arc's discretization

The strategy to fix this problem uses the fact that the boundary vertex E inserted to split the bottom arc AB will not only lie behind edge CD but will also encroach on CD since arc CD is completely included in the diametral lens of edge CD. This fact ensures that the edge CD can always be found to test whether the new vertex E lies behind it. When the new vertex lies outside the domain and some edge CD separates the vertex from the edge AB that it is supposed to split, then CD is split first. In other situations, AB is split first; this prevents infinite recursion.

Even with this change in point placement when splitting boundary edges, it was still posible to construct a proof showing that the modified algorithm will terminate with bounds on mesh quality similar to Ruppert's original scheme. See Appendix A for the complete details of the proof.

2.4.5 Length scale modifications

Whenever a boundary edge is split, the length scale $LS(p)$ for the new boundary vertex needs to be computed using Equation 2.1. For this, the local feature size $lfs(p)$ at the new point p must take into account the curvature of the boundary. The local feature size for curved boundaries, lfs_c is defined as:

$$lfs_c(p) = \min\left(\rho\left(p\right),\, lfs\left(p\right)\right) \tag{2.4}$$

where $\rho\left(p\right) = \frac{1}{|\kappa(p)|}$ is the radius of curvature at point p. The radius of curvature therefore provides a ceiling on the value of the local feature size on the boundary. By using the radius of curvature, there will be an equal number of points per radian on the curve as per gap between objects. For curves with uniform curvature, the edge length from initial discretization and the radius of curvature are equal. For non-uniform curves, with a $TV(\theta)_{\max}$ of $\pi/6$, the edge lengths will be twice as small. This will lead to more points on the curved boundary, as needed. This factor of two will not lead to artificial small features since the LS at that point might be determined by neighbor vertices, not by $lfs_c(p)$. Furthermore, edge lengths can only be proven to be within a factor $(C_S + 1/G)$ of their ideal length. See the Appendix for details on the proof.

2.4.6 Small angles in the domain

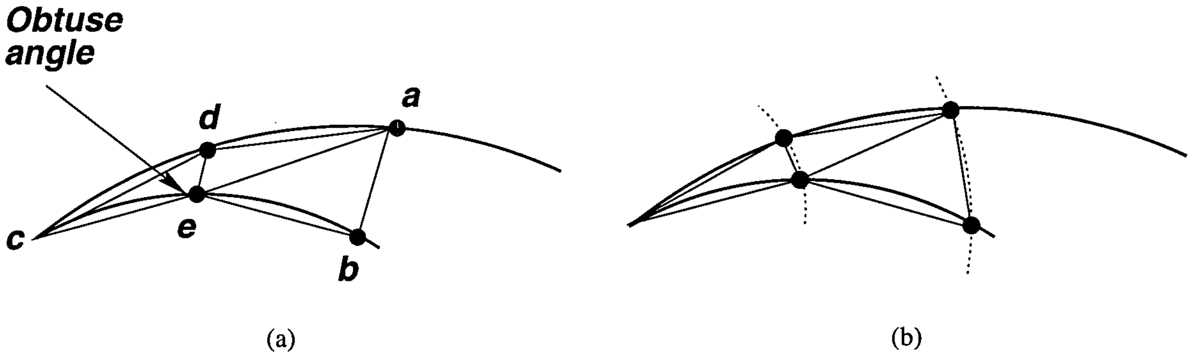


Figure 2.19: Problem associated with curves, small angles, and the use of concentric circular shells.

The concentric circular shells method described in Section 2.3.5 can also be used to prevent infinite insertion around small angles created between two curved boundaries. However, one must make sure that the split points are located using *geometric distance* from the small angle vertex. If total variation of the orientation is used instead of geometric distance, infinite encroachment is still possible, as illustrated in Figure 2.19a. Vertices a and b were first created, perhaps as part of the initial discretization. Since $\angle cba$ is obtuse, vertex d was inserted to fix encroachment on edge ca. However, d encroaches on cb, so it too must be split, using point e. However, $\angle ced$ is still obtuse, so edge cd should be split, and so should ce, etc. Such a problem does not appear when geometric distance from vertex c is used instead of orientation change, as can be seen from Figure 2.19b.

2.5 Implementation details

The evaluation of the total variation of orientation over any type of boundary patch is the cornerstone of both the initial discretization and the midpoint routines. As was mentioned in Section 2.4.1, there is no need to actually compute the integral; one only needs to take the difference in orientation between two points over which the change in orientation of the curve is monotone, i.e. it is either non-varying, or is strictly increasing or decreasing. The details on how to find these particular points are given in the following sections.

2.5.1 Linear patches

The implementation of this type of patch is straightforward. The orientation change over any linear patch is simply zero. Consequently, edges are split at the geometric midpoint of the edge.

Likewise, the initial discretization of a linear patch is trivial: only one edge is needed. The two endpoints of the edge are inserted in the mesh, and the edge between these endpoints is marked for recovery. No other vertices need to be inserted.

2.5.2 Circles and circular arc patches

The total variation of orientation for these patches is easy to obtain since the curvature of a circular arc (or circle) is a constant $1/R$. In order to exactly evaluate Equation 2.2, one only needs to use the second form of the orientation change integral and find the total arclength of the arc, a trivial computation given that the endpoints of the arc are known.

The midpoint between two vertices is calculated using the total variation of the orientation. However, since the curvature is constant, this is equivalent to splitting on arclength alone, and is simple to implement. The original discretization demands a bit more caution. First, the number of edges needed to discretize the arc is found using Equation 2.3. The value of $TV(\theta)_{\max}$ for patches with uniform curvature is $\pi/3$. The endpoints, as well as the extra vertices needed to define the proper number of edges are inserted in the mesh and the N_e edges are marked for recovery. Note that for full circles, there are no endpoints, and N_e is always six.

2.5.3 Cubic parametric curves

Cubic parametric curves are internally defined by two cubic parametric equations $x(t)$ and $y(t)$, with t varying between 0 and 1. Such definition allows the representation of cubic Bézier curves, cubic B-splines and cubic interpolated splines (see section 2.5.4 for details on splines). In order to compute $TV(\theta)$, "critical" values of t between which the orientation of the curve must be monotone are first found. This way, it is possible to simply take the difference in the orientation of the curve at these points to find the total orientation change for the curve. Furthermore, if one is careful to take points on the curve that only allow a maximum change of $\pi/2$ between them, the need to determine if the curve changed orientation by a value of β or a value of $2\pi - \beta$ is eliminated.

To achieve this, the minima and the maxima of both $x(t)$ and $y(t)$ are selected as critical points. This limits $TV(\theta)$ between two critical points to be smaller than $\pi/2$. There are as many as two such points for each cubic equation. The inflexion point for each cubic equation is also chosen as a critical point. Finally, the orientation of a cubic parametric curve might reach a maximum or a minimum at the locations where the curvature is zero. These locations are found using the following definition of curvature:

$$\kappa = \frac{|\,\mathbf{v} \times \mathbf{a}\,|}{v^3}$$

where v= $(x'$,$y')$, a= $(x''$,$y'')$, and v =$|\ \mathbf{v}\ |$. Clearly, the curvature will go to zero whenever $|\ \mathbf{v} \times \mathbf{a}\ |$ does. From this, as many as two more critical points on the curve are obtained, since $\mathbf{v} \times \mathbf{a} = 0$ simplifies to a quadratic equation for cubic parametric curves. In summary, there potentially are eight critical values of t for a 2-D cubic parametric curve: four maxima/minima, two inflexion points, and two zero-curvature points. By ordering these and the endpoints, and then taking the difference in the orientation of the curve between consecutive values, the exact answer to Equation **??** is obtained. An example of a Bézier curve with eight critical points in shown in Figure 2.20. Two critical points are very close to each other on the left part of the curve.

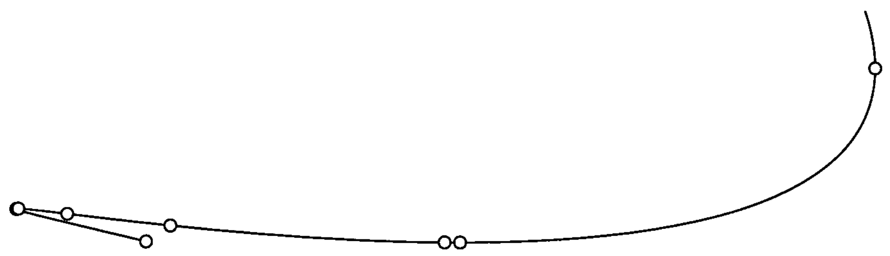


Figure 2.20: Critical points for a Bézier curve

These critical points are stored, and are used to quickly find the location of a midpoint. The location is isolated between two of these critical values (or one of them and an endpoint). Since the curve is monotone between two of these points, it is then possible to use interpolation techniques to find the exact location of the midpoint.

The initial discretization of a parametric curve follows the generic procedure outlined earlier. In this case, $TV(\theta)_{\max} = \pi/6$. The endpoints, as well as the new discretization vertices, are inserted in the mesh, and the N_e edges are marked for recovery.

Tangent vector may be null

It is possible for a cubic parametric curve to have zero values for both components of the tangent vector. This prevents the use of the tangent vector (or the normal vector, which depends on the same data) to determine the orientation of the curve at that particular point. The orientation is usually found using:

$$\theta(t) = \arctan\left(\frac{y'(t)}{x'(t)}\right)$$

The signs of $x'(t)$ and $y'(t)$ are used to determine the quadrant of $\theta(t)$. The tangent vector is null whenever both $x'(t)$ and $y'(t)$ are zero for some $t = t_o$. This makes the ratio of the two indeterminate:

$$\lim_{t\to t_o} \frac{y'(t)}{x'(t)} = \frac{0}{0}$$

By using L'Hospital's rule, the limit becomes:

$$\lim_{t \to t_o} \frac{y''(t)}{x''(t)}$$

L'Hospital's rule is used repeatedly until a finite value for the ratio is found. This finite value can then be used to determine the orientation at $t = t_o$, i.e.

$$\theta(t_o) = \arctan\left(\frac{y''(t_o)}{x''(t_o)}\right)$$

However, in order to determine the right quadrant for $\theta(t_o)$, one must still use the signs of $x'(t)$ and $y'(t)$ as they approach t_o. This can be done by evaluating them with a value of t close to t_o and still within 0 and 1.

2.5.4 Cubic interpolated splines

An interpolated spline is a collection of $n_p - 1$ cubic parametric curves, where n_p is the number of points to be interpolated. As such, its total variation of the tangent angle is just the sum of the total variations of its cubic curves. In the present research, the interpolated splines are created with "no-moment" boundary conditions, i.e. both $x''(t)$ and $y''(t)$ are set to zero at the endpoints. Note that the interpolation points are not necessarily inserted in the mesh – they only define the shape of the curve. The list of critical points for an interpolated spline includes all the critical points of its cubic curves as well as their endpoints. The midpoint is found using the same technique as described in Section 2.5.3, i.e. interpolation techniques are used once the two surrounding critical points are known.

Initial discretization of an interpolated spline is a bit more complicated, as boundary edges will now more than likely span more than one parametric cubic curve. However, the overall process is the same as for cubic parametric curves, with $TV(\theta)_{\max}$ also $\pi/6$.

2.6 Results

Figure 2.21 details the different steps involved in generating a mesh with the generic boundary interface. The domain to be discretized, shown in Figure 2.21a, consists of four linear patches, one Bézier curve (in the lower-right quadrant), and one circle. The interior of the circle is considered hollow in this case and will not be triangulated. Figure 2.21b shows the result of the initial discretization. The domain, defined by the cells, was shaded in order to provide a better idea of its shape. The circle was discretized using 6 edges. The Bézier curve, spanning 90°, was discretized with three edges. This is in accordance with the procedure described in Section 2.5. The non-uniform distribution of the extra vertices due to splitting according to orientation change can be clearly observed. It can also be seen that a boundary edge in the lower right quadrant crosses another boundary patch, a problem that was discussed in Section 2.4.4.

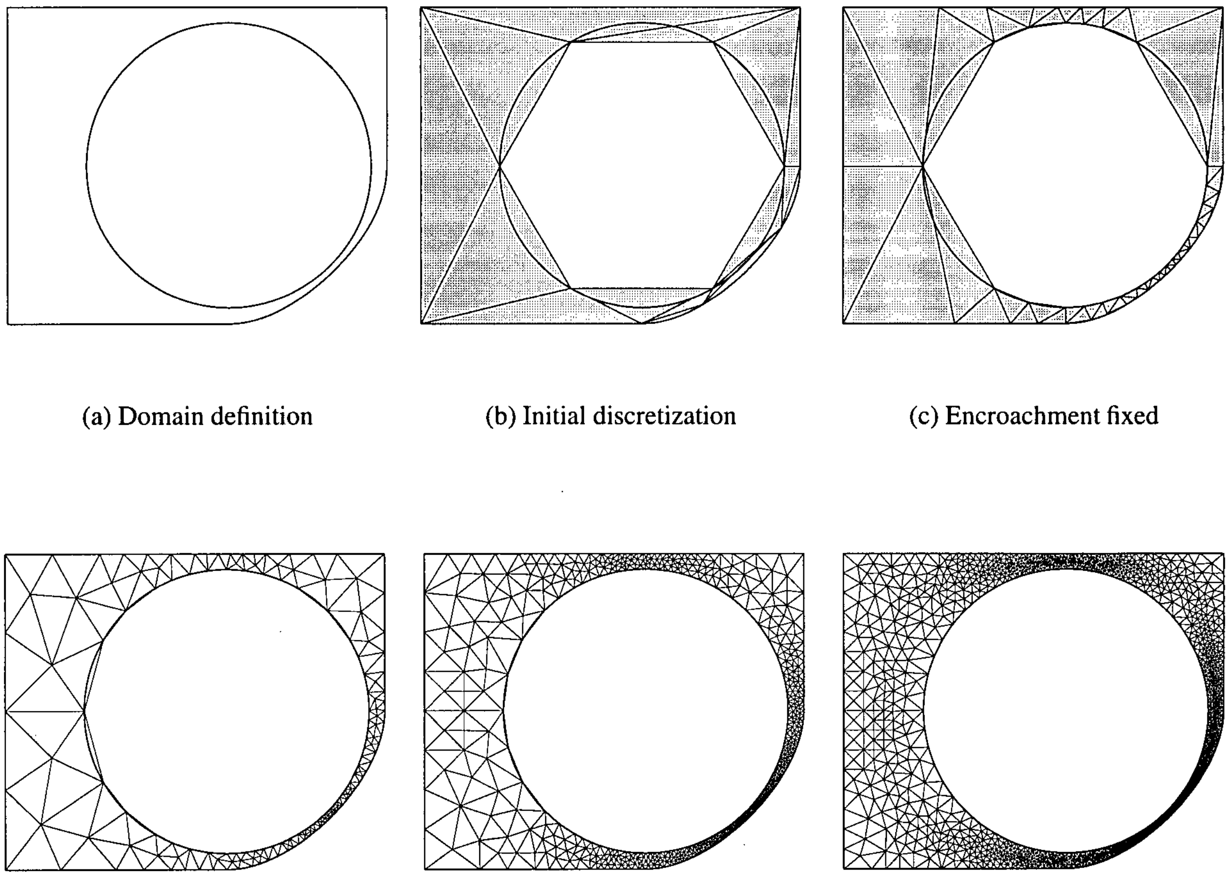
Even though the mesh in Figure 2.21b is a valid constrained Delaunay triangulation, encroached boundary edges must be split before Ruppert's algorithm can be started, as described in Section 2.3.1. Figure 2.21c is the result of the encroachment fix step, and this is the triangulation that Ruppert's algorithm is started with. Note that the circle is now discretized much more precisely in its lower-right quadrant than elsewhere because of its proximity to the Bézier curve.

The final result of Ruppert's algorithm is shown in Figure 2.21d. All of the angles in this mesh, and the following ones, are equal to or larger than 30°. To demonstrate how the generic boundaries adapt to a change in required resolution, meshes with two and four times the resolution of the mesh in Figure 2.21d have been generated. These are shown in Figures 2.21e and 2.21f, respectively.

Figures 2.22 and 2.23 show that the algorithm can easily handle complex geometries with generic boundaries. They were both created using $R = G = 4$. In order to demonstrate more practical uses, the mesh of the region surrounding a 4-element airfoil, shown in Figure 2.24, has also been included. The boundary geometry is defined by a circle and four interpolated splines. This relatively coarse mesh was created using $R = G = 1$ for clarity. The immediate surroundings of the airfoil are shown magnified in Figure 2.25, with 2.25a detailing the state of the mesh after initial discretization, and 2.25b the final result from Figure 2.24.

Table 2.1 summarizes the quality of the three previous meshes. The size and grading parameters used, the minimum and maximum angle in the mesh, as well as the ratio of the actual edge length to the "theoretical" edge length (from the average of the LS at its vertices) are listed. These meshes were all generated with an imposed minimum angle bound of 30°. With higher values of R and G, the major constraint is the cell size, not its shape. This explains why the angle bounds as well as the edge length ratios are better for these cases. For case with lower values of R and G, the angle bound is harder to reach than the size constraint. This results in smaller cells in some regions, which affects the edge length ratios.

The use of the generic boundary interface did have a small impact on the time required to insert



(a) Domain definition (b) Initial discretization (c) Encroachment fixed

(d) Final mesh (R=G=1) (e) R=G=2 (f) R=G=4

Figure 2.21: Drawings a, b, and c show the steps required to obtain meshes d, e, and f.

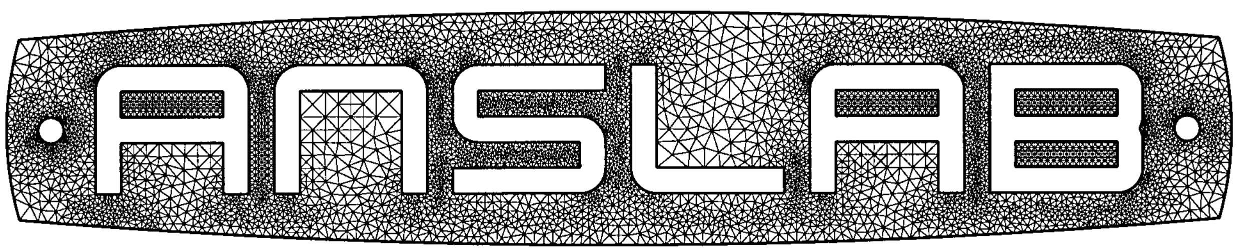


Figure 2.22: Mesh including lines, circles, and arcs as boundary patches. All angles in the mesh are above 30°.

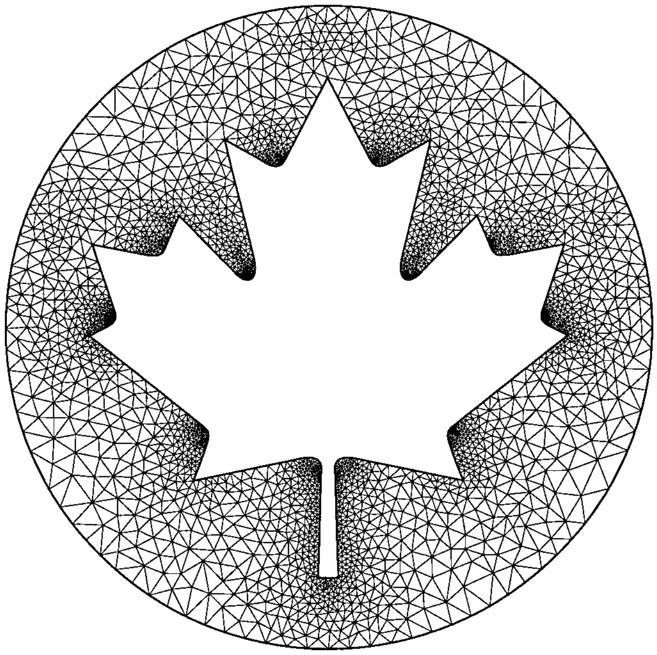


Figure 2.23: Mesh with a boundary made up of Bézier curves, lines, and a circle.

	Parameters		Angles (in degrees)		Edge length ratios		
Figure number	R	G	Min	Max	Min	Avg	Max
2.21	1	1	30.01	104.08	0.2500	0.8819	1.8598
	2	2	35.11	104.42	0.4362	0.8937	2.001
	4	4	32.56	114.88	0.4547	0.9173	1.8015
2.22	4	4	33.85	108.15	0.3811	0.9223	1.7552
2.23	4	4	33.31	110.06	0.1257	0.9003	1.9664
2.24	1	1	30.89	111.21	0.0669	0.3686	3.4170

Table 2.1: Quality measures

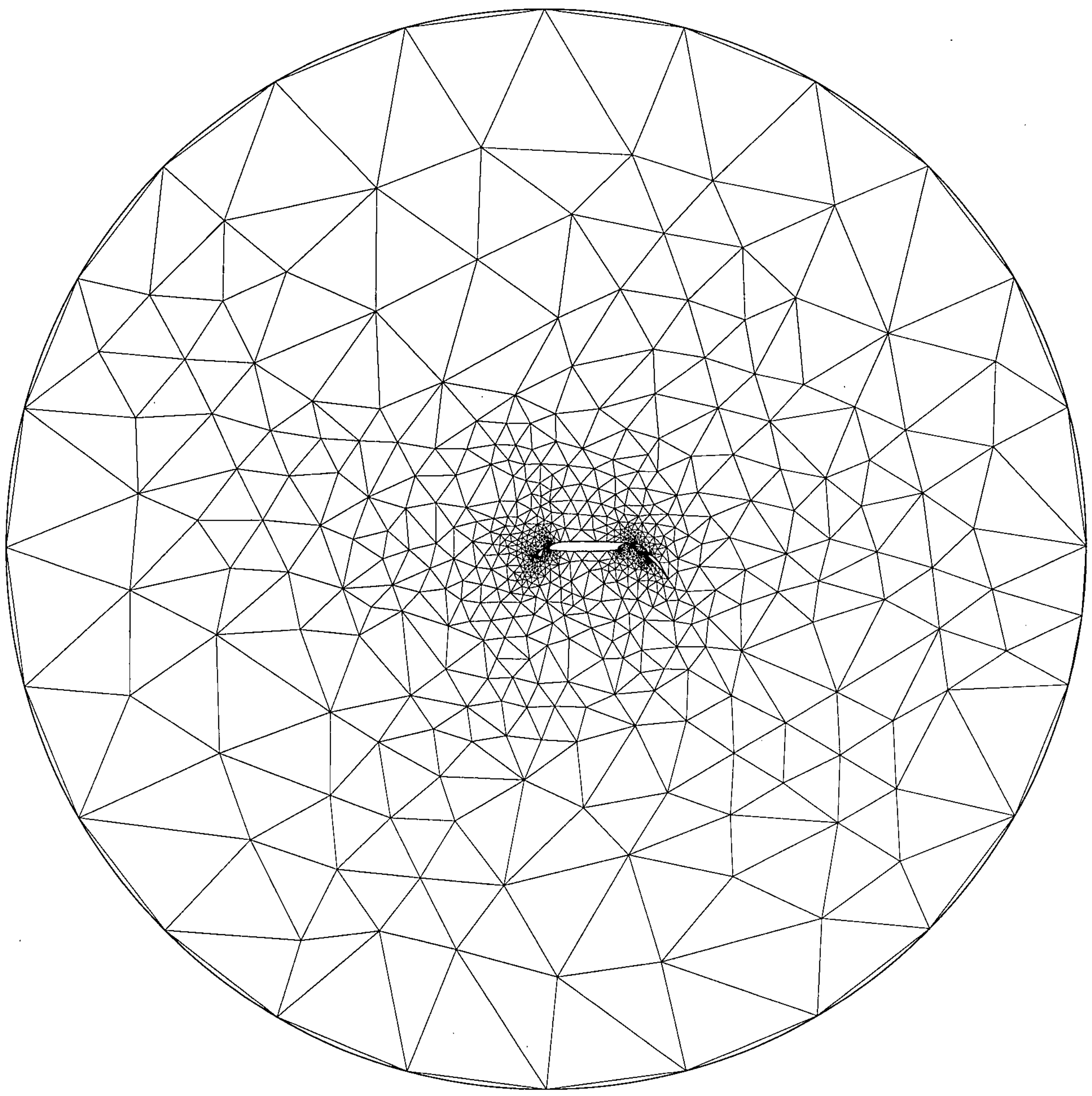
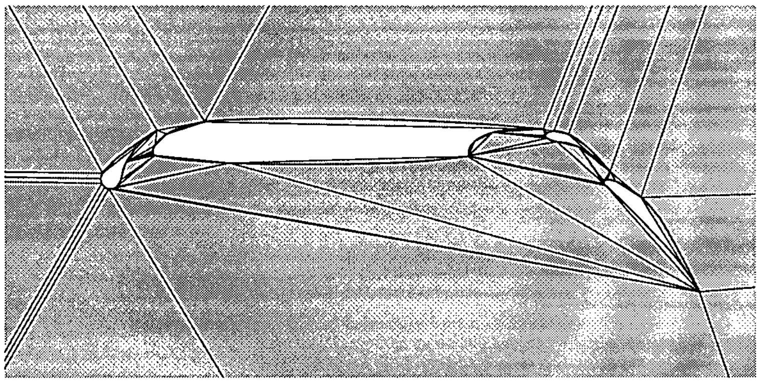
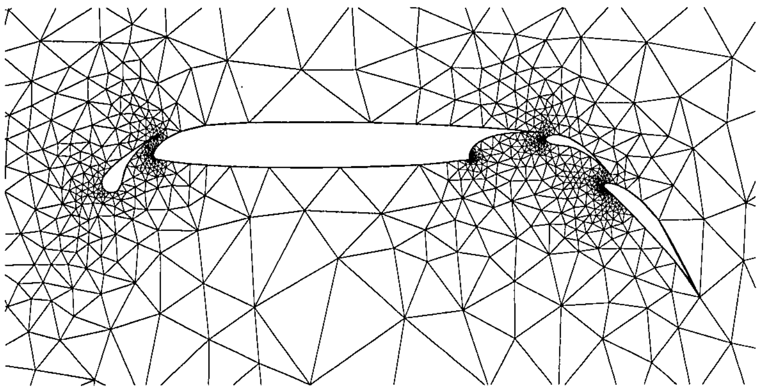


Figure 2.24: 4-element airfoil mesh.



(a) After initial discretization



(b) Final mesh, R=G=1

Figure 2.25: Magnified sections of the 4-element airfoil

points on the boundary. The boundary point insertion routines, which include the calls to determine the location of the new midpoint, take about 5% longer than before, on average. However, considering that, on a typical mesh, the time spent on boundary insertion accounts for less than 1% of the total time, the overall impact on performance is negligible.

2.7 Conclusions

A new framework allowing the use of curved boundaries with a guaranteed-quality Delaunay refinement algorithm has been presented. The boundary data has been separated from the meshing algorithm, removing all assumptions about the shape of the boundary from the meshing code.

The use of curved boundaries demanded a new way of splitting boundary edges, to ensure regions with higher curvature were discretized with a greater number of edges. The midpoints are now computed using the total variation of the tangent angle, $TV(\theta)$. Whenever $TV(\theta)$ is negligible over a given boundary edge, the arclength is used to compute the midpoint.

The introduction of curved boundaries also demanded a new initial discretization strategy. Curved patches are first discretized with as few segments as possible. The minimum number of segments required is determined by the total variation of the tangent angle of the patch. One must also make sure that the curved patch is always protected by the diametral lenses of its boundary edges. Some recovery problems associated with this rather coarse initial representation of the boundary were found. A new strategy for edge recovery was developed and presented in this document.

Several patch types have been implemented and tested successfully. New boundary types can be added to the generic boundary interface by implementing responses for all the generic queries used by the meshing algorithm.

Finally, examples demonstrating the successful use of curved boundary patches were shown. These meshes all showed excellent quality, with a minimum angle exceeding 30° in all of them. Their resolution and grading were easily controlled using parameters R and G. It was also observed that the generic boundary interface had a negligible impact on the time required to mesh a domain.

Chapter 3

Generic Finite-Volume Solver

In this chapter, the details of transforming a finite-volume solver into a generic solver will be explained. A finite-volume solver requires the domain be decomposed into a finite number of control volumes. The more control volumes in the solution, the more accurate the numerical solution. In this project, the original solver used an unstructured domain decomposition into triangular control volumes, or *cells*. The discretized equations for the problem, introduced in Section 3.1, are then solved for each of the control volumes. The finite-volume method solves for the control-volume averaged value of the unknowns, so in order to get a smooth solution (for accuracy purposes), the solver used in this research performs a *reconstruction* of the solution. Reconstruction is detailed in Section 3.2. Boundary conditions can be enforced in two ways: with a constraint on the reconstruction, or using a boundary flux. Both methods are covered in Section 3.3. The task of advancing the solution in time (or towards a steady-state) is presented in Section 3.4. Section 3.5 summarizes the steps required to solve a finite-volume problem.

The concept of a generic solver and the modifications it requires are introduced in Section 3.6. A standard interface in the form of a class was created to access the physics of the problem in a generic fashion; this interface, the `Physics` class, is described in Section 3.6.1. The interface for boundary conditions is detailed in Section 3.6.2. Classes were also used to generalize the types of meshes that can be used with the solver; the `Mesh` class is briefly covered in Section 3.6.3. Finally, a generic interface was created to access the data at the control volume boundaries in a transparent manner regardless of the type of mesh used, and this interface is known as the `Recon` class. Details are given in Section 3.6.4.

3.1 General finite-volume formulation

To derive the finite-volume formulation of physical problems, it is useful to start with the generic form of partial differential equations. In two dimensions, these equations are:

$$\frac{\partial U}{\partial t} + \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} = S \tag{3.1}$$

In this equation, U is a vector containing the unknowns of the problem. U will be referred to as the solution vector, or flux variables vector. F_x is the x-component of the flux vector, and F_y its y-component. The source term vector is represented by S. Integrating over the control volume area[1], we obtain:

$$\int_A \frac{\partial U}{\partial t} dA + \int_A \frac{\partial F_x}{\partial x}\, dA + \int_A \frac{\partial F_y}{\partial y}\, dA = \int_A S\, dA$$

$$\int_A \frac{\partial U}{\partial t}\, dA + \int_A \left(\frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} \right) dA = \int_A S\, dA \tag{3.2}$$

By letting $\vec{F} = F_x\vec{\imath} + F_y\vec{\jmath}$, Equation 3.2 can be rewritten as:

$$\int_A \frac{\partial U}{\partial t}\, dA + \int_A \nabla \cdot \vec{F}\, dA = \int_A S\, dA \tag{3.3}$$

Applying Gauss's theorem to the second term of Equation 3.3 results in:

$$\int_A \frac{\partial U}{\partial t}\, dA + \oint_{\partial A} \vec{F} \cdot \vec{n}\, ds = \int_A S\, dA \tag{3.4}$$

The contour integral is evaluated along the boundaries of the control volume. In this case, $\vec{F}$ is the flux vector across control volume boundaries.

So far, the equations have only been rewritten in a different way; they remain mathematically the same. The finite-volume approximation is now introduced in the equations. The finite-volume method only solves for the control-volume average of the variables in vector U. The control-volume average of variable X over control volume i is defined as:

[1] In two dimensions. For three-dimensional problems, the equations are integrated over the volume.

$$\bar{X}_i = \frac{1}{A_i} \int_{A_i} X \, dA_i \tag{3.5}$$

Applying the averaging equations to the solution vector and the source term, and re-arranging terms, we obtain the finite-volume approximation of the original PDE:

$$\frac{\partial \overline{U_i}}{\partial t} = -\frac{1}{A_i} \oint_{\partial A_i} \vec{F} \cdot \vec{n} \, ds + \frac{1}{A_i} \int_{A_i} S \, dA = FI_i \tag{3.6}$$

We define FI_i as the flux integral for control volume i (even though it also includes the source term integral). Equation 3.6 is solved on each of the control volumes. The integration is carried out using Gauss quadrature. This method does not introduce any arbitrary elements to the equations, such as basis or test functions. Since the average over each control volume is preserved, the variables in the vector of unknowns U are automatically conserved on the domain.[2] This makes the finite-volume method particularly attractive for computational fluid mechanics problems, where the conservation of quantities such as mass, momentum, and energy is critical.

3.2 Reconstruction of the control-volume averaged data

As mentioned previously, a finite-volume solver only computes cell-averaged data, meaning that only a constant average value for the flux variables is available for a given cell. When evaluating the fluxes at the boundaries, or *faces*, of the control volume, using these averaged values is inadequate as it leads to a first-order accurate solution only. This is explained by the fact that the difference in control-volume averages between two neighbor control volumes is first-order with respect to the mesh spacing.

The solution is therefore *reconstructed* over each control volume for improved accuracy [6]. This reconstruction generates a smooth polynomial valid over the entire control volume. This polynomial can be linear, quadratic or cubic, depending on the order of the reconstruction. The polynomial function is chosen to preserve the control-volume average, and attempts to predict the average values of neighbor control volumes as well as possible. The function is obtained by solving a constrained least-squares problem, where the main constraint is the average value of the function over the cell.

[2]This is dependent on the proper accumulation of fluxes in control volumes neighbor to a given face; basically, one must ensure that flux leaving a given control volume at a face is the same as the flux that enters the control volume on the other side of that face.

Once such a function has been computed, it can be used to obtain the flux variables values at the control volume boundary. Figure 3.1 displays the difference between the averaged solution (where the solution is constant over a given cell) and a second-order reconstruction of the same solution on an unstructured mesh (where the solution varies linearly over the cell). It is clear from this figure that the reconstructed solution will yield a much more accurate value of the flux variables at the control volume boundaries.

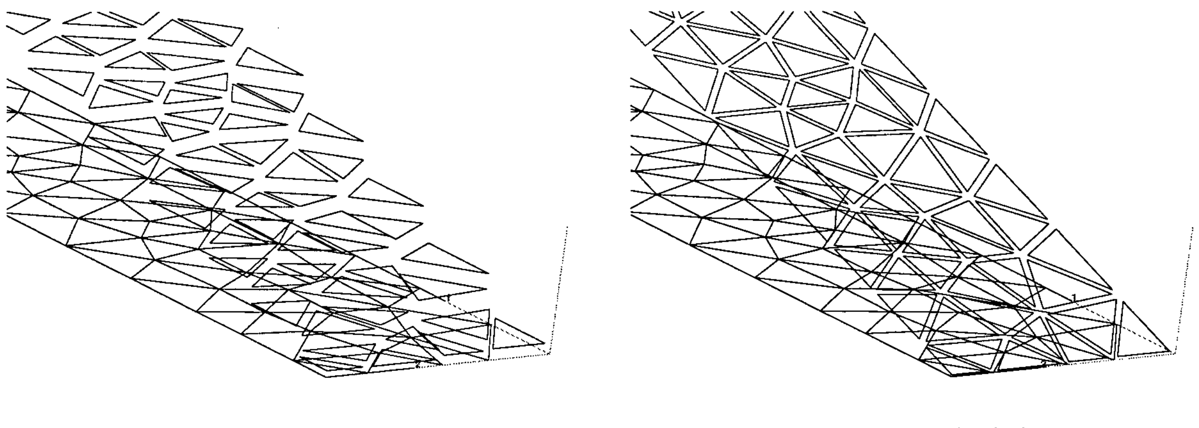

(a) Control-volume averaged solution (b) Reconstructed solution

Figure 3.1: Difference between averaged and reconstructed solution. The height of the cells represents their value.

This extrapolation is only valid over a given control volume. The extrapolations from two neighbor control volumes will yield different values at the common boundary. Both of these values are available for flux computation, and it is up to the user to decide whether one, the other, or a combination of both values is to be used. In some cases, the physics of the problem will dictate that choice, as in problems requiring upwind formulation of the fluxes.

3.3 Boundary conditions

Some of the control volume boundaries will also be domain boundaries. The fluxes evaluated at these locations are used to implement the boundary conditions of the problem.

Boundary conditions can be enforced in two different forms. The first form, known as the *weak form* of the boundary condition, imposes a specific flux at the boundary. When such a boundary condition is encountered, the solver does not use the flux $\vec{F}$ but rather the flux $\vec{F_b}$ specified by the user for that boundary.

The *strong form* of the boundary condition imposes a specific value of the flux variables (or any of their derivatives) at the boundary control volume. This is done by adding constraints on the reconstruction of the solution over the boundary cell [36, 41]. The polynomial computed by the reconstruction will match the value of the flux variables specified at the boundary. The flux $\vec{F}$ can then often be used for the boundary flux; since it will be evaluated with the values obtained by the reconstruction, it will account for the constraints of the boundary condition.

One can then distinguish between two fluxes in the solver: the *interior flux*, used on interior faces, is *always* the flux $\vec{F}$ as described in Section 3.1. The *boundary flux* is used on faces adjacent to the domain boundaries, and can be the flux $\vec{F}$ or a specific boundary flux $\vec{F_b}$ specified by the user, depending on the form of the boundary condition used. It is also possible to use a constraint on the reconstruction as well as a specific boundary flux for a given boundary condition, if needed.

3.4 Time-advance

The solver in this research advances the solution in time (or towards steady-state) using an explicit multistage scheme. This family of schemes uses intermediate solutions and evaluates the flux integral at each of these intermediate steps in order to obtain the solution at the next timestep. These methods are self-starting, i.e. they do not require data at any previous timestep, and can be made to be high-order accurate simply with an appropriate choice of coefficients. Schemes up to fourth-order accurate were used in this research, and are presented below.

3.4.1 Evaluating the timestep

The value of the timestep Δt used to advance the solution in time can either be determined by the user or evaluated by the solver. A value for a stable timestep is computed for each control volume in the domain. These values can then be used to perform global timestepping, where the smallest timestep in the whole domain is used to advance the solution in time, or local timestepping, where solution is advanced using the local value of the timestep over each control volume. The latter method can only be used to obtain steady-state solutions.

The value of the timestep for a given control volume is determined using integration of wavespeeds around the control volume, an approach suggested by Barth [5]:

$$\Delta t_{max_i} = \frac{V_i}{\oint_{CV_i} C_{max}\, ds} \tag{3.7}$$

where V_i is the volume of the control volume and C_{max} is the fastest wave entering the control volume. This wavespeed is determined by the flux function, and sent to the solver in parallel to the flux computations.

For diffusive types of problems, a "pseudo" wavespeed needs to be computed so that the solver can estimate the value of the timestep. This pseudo wavespeed was determined using the one-dimensional heat conduction problem as an example. With explicit time-advance on a structured grid, this type of problem requires $\Delta t_{max} \propto \Delta x^2$. A "wavespeed" of $c = 1/\Delta x$ is therefore required. Such a value is easily computed on structured meshes, but for unstructured meshes, an equivalent inverse distance measure is required. In ANSLib, $1/\Delta x$ is replaced by:

$$\frac{(\vec{x_j} - \vec{x_i}) \cdot \hat{n}_{ij}}{|\vec{x_j} - \vec{x_i}|^2} .$$

where $\vec{x_j} - \vec{x_i}$ is the vector from the control volume reference location in control volume i to that in control volume j and $\hat{n}_{ij}$ is the unit normal vector to the interface between control volumes i and j, pointing into control volume j. This inverse distance measure is always available to the flux functions.

Furthermore, the user can also provide a value of the CFL number, which will be used to modify how the timestep that was computed is used:

$$CFL \equiv \frac{\Delta t}{\Delta t_{max}} \tag{3.8}$$

3.4.2 First-order

The first-order scheme used is known as the Explicit Euler scheme:

$$U_{n+1} = U_n + h\, \mathrm{FI}_n \tag{3.9}$$

where FI_n is the value of the flux integral for cell i at timestep n, and h is the value of the timestep taken multiplied by the CFL number, i.e. $h = \mathrm{CFL} \times \Delta t$. Note that the cell subscripts i are omitted here for clarity.

3.4.3 Second-order

The second-order scheme used is the second-order Runge-Kutta scheme:

$$\begin{aligned} U_{n+\frac{1}{2}} &= U_n + \tfrac{1}{2} h \, \mathrm{FI}_n \\ U_{n+1} &= U_n + h \, \mathrm{FI}_{n+\frac{1}{2}} \end{aligned} \tag{3.10}$$

3.4.4 Third-order

First- and second-order accurate schemes only require the solution and flux integral from the previous (intermediate) time step to get an updated solution. This is not the case for higher order schemes, as they need the flux integrals for up to 4 previous intermediate timesteps to be available. These schemes therefore require extra storage space compared to the first- and second-order schemes. The third-order scheme used is a third-order Runge-Kutta scheme[3]:

$$\begin{aligned} U_{n+\frac{1}{4}} &= U_n + \tfrac{1}{4} h \, \mathrm{FI}_n \\ U_{n+\frac{1}{2}} &= U_n + \tfrac{h}{10} \left(9 \, \mathrm{FI}_n - 4 \, \mathrm{FI}_{n+\frac{1}{4}} \right) \\ U_{n+1} &= U_n - \tfrac{h}{3} \left(8 \, \mathrm{FI}_n - 16 \, \mathrm{FI}_{n+\frac{1}{4}} + 5 \, \mathrm{FI}_{n+\frac{1}{2}} \right) \end{aligned} \tag{3.11}$$

3.4.5 Fourth-order

A fourth-order Runge-Kutta scheme can be found using the same methodology as the third-order scheme:

$$\begin{aligned} \widehat{U}_{n+\frac{1}{2}} &= U_n + \tfrac{1}{2} h \, \mathrm{FI}_n \\ \widetilde{U}_{n+\frac{1}{2}} &= U_n + \tfrac{1}{2} h \, \widehat{\mathrm{FI}}_{n+\frac{1}{2}} \\ \overline{U}_{n+1} &= U_n + h \, \widetilde{\mathrm{FI}}_{n+\frac{1}{2}} \\ U_{n+1} &= U_n + \tfrac{1}{6} h \left[\mathrm{FI}_n + 2 \left(\widehat{\mathrm{FI}}_{n+\frac{1}{2}} + \widetilde{\mathrm{FI}}_{n+\frac{1}{2}} \right) + \overline{\mathrm{FI}}_{n+1} \right] \end{aligned} \tag{3.12}$$

Details on how the coefficients for these schemes were determined can be found in [28].

[3]The equations governing the value of the different coefficients for a given Runge-Kutta scheme are underdetermined; there is therefore a certain freedom in the choice of coefficients, leading to a variety of Runge-Kutta schemes of the same order.

3.5 Implementation of a finite-volume solver

In summary, a reconstruction-based finite-volume solver repeatedly performs the following steps to get a numerical solution:

1. Setup boundary constraints on the reconstruction, if necessary
2. Perform reconstruction over all control volumes
3. Evaluate the fluxes along all control volume boundaries
4. Accumulate the fluxes in the proper control volumes
5. Evaluate the source term integral over each control volume
6. Perform next time-advance step

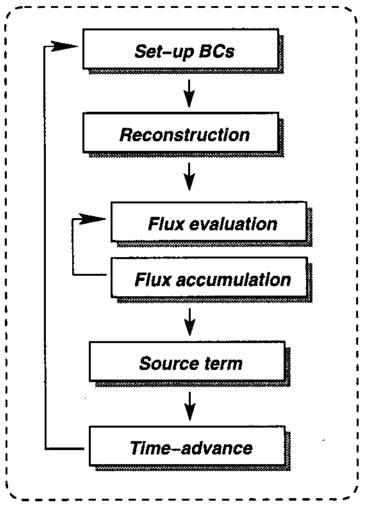


Figure 3.2: Overview of the process involved in solving a numerical problem using the finite-volume method

A schematic of the steps above is shown in Figure 3.2.

3.6 Modifications needed by a generic solver

The concept behind a generic finite-volume solver is quite simple. Of all the steps outlined in Section 3.5, the physics of the problem to solve only appear in steps 1, 3, and 5. Boundary constraints (step 1) are related to the boundary conditions, which are highly physics-dependent. The fluxes of the problem (step 3) depend on the physics of the problem being solved, and finally, the source term (step 5) is also related to the physics of the problem. Steps 2, 4, and 6 are all strictly numerical methods, and no knowledge of the type of problem being solved is necessary to perform these steps properly. Figure 3.3 illustrates the physical information required for each step in the process.

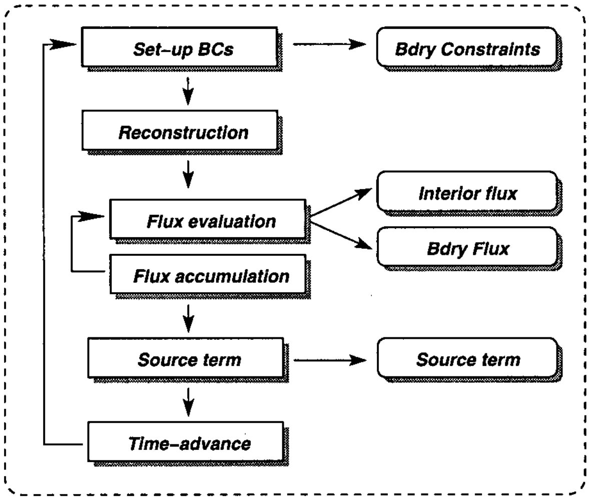


Figure 3.3: Detail of the physical information needed in the finite-volume method

The essential idea for a generic solver is to have all the numerical parts of the problem handled by a finite-volume toolkit. This toolkit then makes calls to external physical packages to receive the information relevant to the physics of the problem being solved. By making all physical packages look the same to the numerical toolkit, *any* physical problem can be solved using the generic solver.

This section describes how the separation of the numerical and physical aspects of the solver was accomplished. The technique considered takes full advantage of the object-oriented approach to programming. Object-oriented programming allows the creation of standard interfaces to parts of code. These interfaces are known as *classes*. The generic solver interacts mainly with the `Physics` class.

3.6.1 The `Physics` class

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

Number of flux variables

The first information the `Physics` class provides to the solver is the number of flux variables in the problem. The number of variables will determine the size of the storage arrays needed later in the simulation. It also sets the number of items in the flux vector.

Interior flux

Perhaps the most important physical aspect of a problem, the interior flux is the flux that crosses the control volume boundaries located in the interior of the domain. This corresponds to the flux $\vec{F} = F_x\vec{\imath} + F_y\vec{\jmath}$ introduced in Section 3.1. The solver will call the interior flux function at every location it needs to be evaluated, i.e. at every Gauss point on each cell boundary.

The interior flux function returns the result of $\vec{F} \cdot \vec{n}$. Information about the location of the Gauss point, as well as the normal vector are available to this function. Furthermore, the values of flux variables (as well as their derivatives) from each side of the boundary are available. These values are known as the *left* and *right* values.

Source term

The source term of the equation, S (from Equation 3.4), must also be provided to the solver. In some cases, a `Physics` class may not have a source term. However, if one is defined, the solver will compute the value of the source term at each of the Gauss points in the control volume. The number of Gauss points varies with the order of accuracy used for the simulation.

The source term function returns the value of S. Once again, the values of the flux variables and their derivatives are available to the function, as well as the location of the Gauss point. No normal vector is provided since the source term is not a vector quantity. Since the source term is evaluated in the interior of the cell, there is only one value for each variable (no left and right values).

Related quantities

The `Physics` class is able to 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.

Initial solution

Finally, a `Physics` class must be able to provide a generic initial solution that the solver can use in case the user does not specify one.

3.6.2 Boundary condition types

The boundary conditions are also defined in the `Physics` class. Each boundary face in a mesh has a boundary condition number associated with it. This number will then be associated with a boundary condition *type*, specific to the `Physics` class. This boundary condition type can be used for any problem where a particular physical phenomenon occurs, so it must not rely on a specific boundary condition number, or mesh.

The boundary condition types can be defined using either a specific boundary flux, or by specifying constraints on the boundary reconstruction. It is possible to assign several boundary types to a boundary condition number. 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.

The boundary flux function returns the value of the flux in a way similar to the interior flux function. The location, the normal vector, as well as the values for the flux variables are available. The only difference from the interior flux function is that the data only comes from one side, since the boundary face only has one neighbor cell; the left and right values are therefore identical.

Boundary constraints also have the location, as well as the normal vector available. Additionally, the value of the constraint itself, typically set by the user, is also available.

Figure 3.4 shows schematically how the solver has been modified to make calls to an external `Physics` class whenever physical information is needed. The `Physics` class is shown with two boundary condition types for illustration purposes.

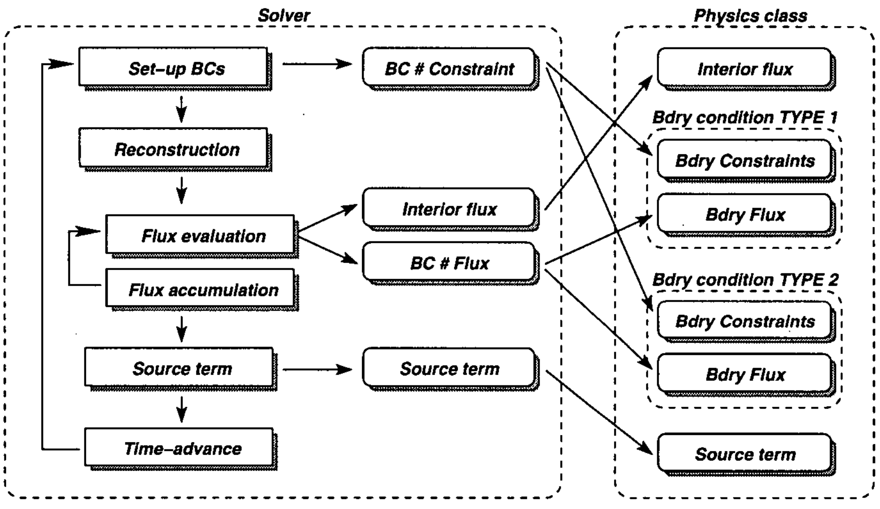


Figure 3.4: Solver making calls to external `Physics` class

3.6.3 Separate `Mesh` classes

Another exciting feature made possible by the use of classes is to make the solver use a generic mesh structure. This allows the solver to be used with *any* type of mesh. Not only can the solver potentially use both structured and unstructured meshes, it can also use both two-dimensional *and* three-dimensional meshes, all within the same solver. This done by having the solver interact with the mesh through a standard interface called a `Mesh` class. The base `Mesh` class defines the interface that allows access to information such as the number of control volumes, the dimension of the mesh, as well as the interface for flux integration on the mesh. Each specialized mesh class then implements these functions.

Two main branches of the `Mesh` class have been written already. The first one, the `Unstructured-Mesh` class, implements the various containers necessary for unstructured mesh representation, as well as the functions needed to access them. This class is then further specialized into 2-D (triangular) and 3-D (tetrahedral) classes, and both of these classes also have implementations for both cell- and vertex-centered meshes.[4] The other main branch for the `Mesh` class is the `StructuredMesh` class. This class is also divided in both 2-D and 3-D classes, although only the 2-D class has been implemented so far. The 2-D structured mesh class uses quadrilateral-shaped control volumes.

[4]Unless specified, the unstructured meshes used in this research are always of the cell-centered type, where the control volume exactly matches the cell in the mesh.

3.6.4 The `Recon` class

The `Recon` interface has been created to facilitate access to reconstructed solution data at Gauss points. The interface for the functions that compute the value of the flux variables and their derivatives is defined here. The class also implements how the constraints on the reconstruction can be defined.

The reconstruction method introduced in Section 3.2 is used with unstructured meshes. The code for this method was moved into a specialized version of the reconstruction class for unstructured meshes. Reconstruction could be used along with structured meshes as well, but the code would have to be modified in order to take into account the difference in mesh topology. It would be possible to write a structured version of the `Recon` class that would implement this modification.

However, with structured meshes, it is possible to take advantage of the predictable topology and compute the values and gradients much more efficiently using Taylor series expansions. These expansions eliminate the costly step of solving a least-squares problem for every control volume in the mesh. This step in simply ignored when using the structured specialization of the `Recon` class. For the solver, however, there is no difference, since the interface for the unstructured and the structured `Recon` classes are exactly the same.

The unstructured reconstruction method implements first-, second-, third-, and fourth-order accurate schemes. The structured version only implements a second-order scheme at the time of writing.

Figure 3.5 is a schematic of how the solver and the various classes highlighted in this section interact. The solver needs access to the `Recon` class whenever it needs access to reconstructed data: when computing interior fluxes, source terms, and boundary conditions. In turn, the `Recon` class needs access to the topology of the mesh in order to perform the reconstruction (unstructured) or compute the Taylor series (structured). Furthermore, the solver needs to know the location of the Gauss points for flux computation, so it also interacts with the `Mesh` object.

3.7 Sample problems

To demonstrate the effectiveness of the generic solver, three different physical problems were solved. The physical phenomena investigated were: heat conduction, solid mechanics, and incompressible fluid flow. The mesh remained the same for the three problems, and is illustrated in

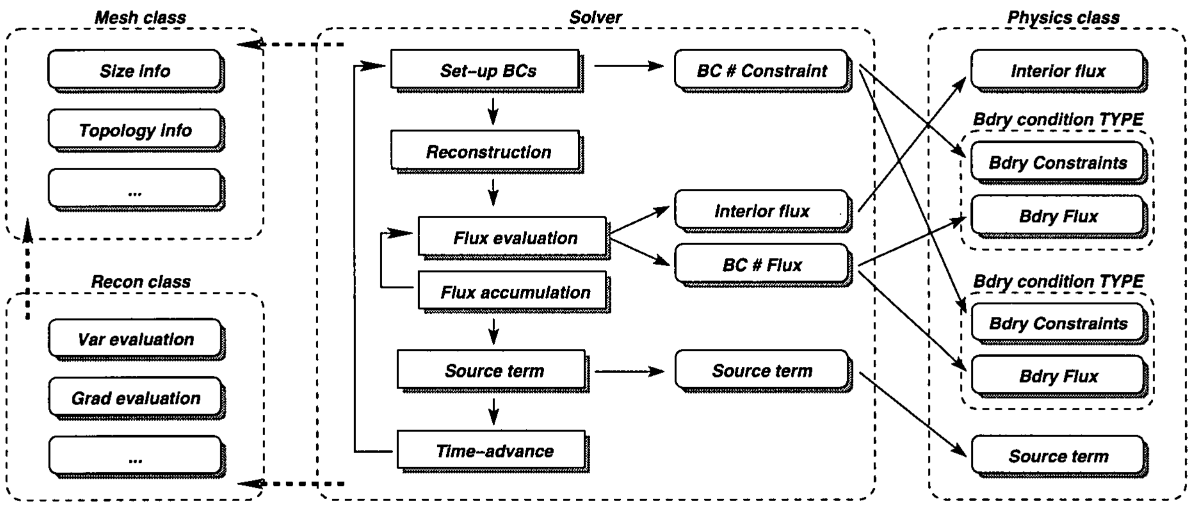


Figure 3.5: Schematic of the various standard interfaces for the physics, the mesh, and the reconstruction interacting with the generic solver

Figure 3.6.[5] The domain has a length $L = 1.0$ and a height $H = 0.2$. All the simulations were performed using a cell-centered scheme.

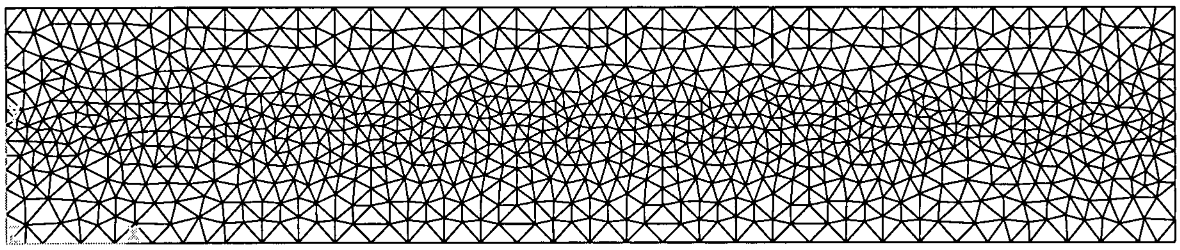


Figure 3.6: Mesh used for the sample problems presented in this chapter

The following problems can easily be solved analytically; they are used to demonstrate the flexibility and the accuracy of the generic solver. The numerical solutions were compared to their analytical counterpart.

[5]Keeping the same mesh is, of course, not necessary. In fact, some of these problems could be solved more efficiently on a different mesh. However, for illustration purposes, the same mesh is used for all three problems.

3.7.1 Heat conduction

Physical description

Heat conduction is governed by the following differential equation:

$$\frac{\rho c_p}{k}\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\dot{q}}{k} \tag{3.13}$$

Where T is the temperature, $\dot{q}$ is the energy generated per unit volume, k is the thermal conductivity, ρ is the density and c_p the specific heat of the material. Letting Equation 3.13 take the form of Equation 3.1 results in:

$$\frac{\partial T}{\partial t} - \frac{\partial}{\partial x}\left(\alpha\frac{\partial T}{\partial x}\right) - \frac{\partial}{\partial y}\left(\alpha\frac{\partial T}{\partial y}\right) = \frac{\dot{q}}{\rho c_p} \tag{3.14}$$

where $\alpha = \frac{k}{\rho c_p}$ is the heat diffusivity of the material. Comparing Equation 3.14 to Equation 3.1, it can be seen that:

$$U = T \tag{3.15}$$

This translates into the following fluxes:

$$F_x = -\alpha\frac{\partial T}{\partial x} \tag{3.16}$$

$$F_y = -\alpha\frac{\partial T}{\partial y} \tag{3.17}$$

and the following source term:

$$S = \frac{\dot{q}}{\rho c_p} \tag{3.18}$$

Boundary condition type: Imposed temperature

There are two boundary condition types for the heat conduction problem. The first one imposes a temperature at the wall. This is done by setting a constraint for the temperature value at the wall. The value of the constraint is set by the user. The boundary flux is simply $\vec{F_b} = F_x \cdot n_x + F_y \cdot n_y$.

Boundary condition type: Imposed heat flux

The second boundary condition type imposes a specific heat flux at the wall. This boundary condition is defined by specifying a boundary flux. The normal heat flux at the wall $q_n = k\frac{\partial T}{\partial n}$ is specified by the user. The boundary flux then becomes:

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

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

Problem definition

The domain was subjected to a temperature boundary conditions on all boundaries. The left boundary had an imposed temperature of $T_L = \frac{10y}{H}$; the right boundary had a similar temperature boundary condition, but with a decreasing temperature in y: $T_R = 10 - \frac{10y}{H}$. The top and bottom surfaces have imposed temperatures varying with x, with $T_B = \frac{10x}{L}$ and $T_T = 10 - \frac{10x}{L}$.

The analytical steady-state solution for this problem is a bilinear temperature distribution:

$$T(x,y) = \frac{10x}{L} + \frac{10y}{H} - \frac{20xy}{LH}$$

Looking along the diagonal joining the top-left corner to the bottom-right corner of the domain, the temperature profile is:

$$T(x)\big|_{y=H\left(1-\frac{x}{L}\right)} = 20\frac{x}{L}\left(\frac{x}{L} - 1\right) + 10$$

This is the temperature profile that will be used for comparisons.

Since they do not affect the steady-state solution, k, ρ, and c_p were set to 1. A second-order accurate scheme was used.

Results

A second-order accurate reconstruction scheme recovers the gradients of the interior flux exactly, so it is no surprise to see in Figure 3.7 that the numerical solution matches the analytical solution well.

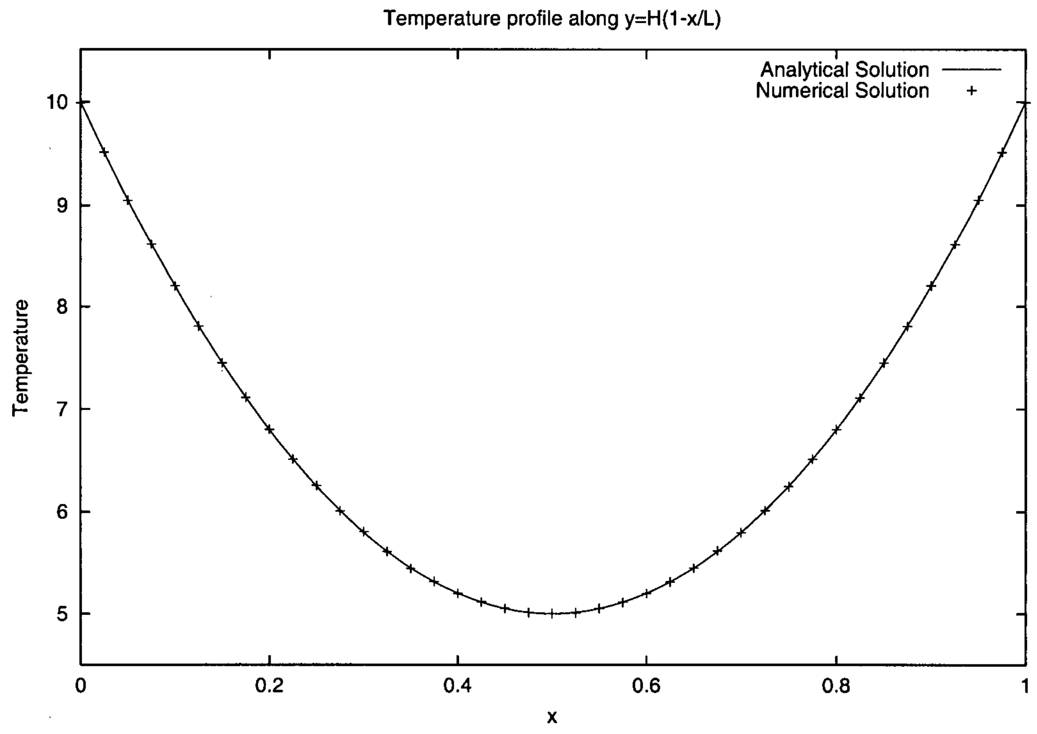


Figure 3.7: Temperature profile along $y = H(1 - \frac{x}{L})$

3.7.2 Solid mechanics: plane stress

The second test problem is most often solved using the Finite Element Method. However, it is possible to solve plane-stress problems with ANSLib; one only has to provide the necessary description of the physics.

Physical description

The differential equations of motion of a deformable solid are (from [10]):

$$\frac{\partial \sigma_{ij}}{\partial x_j} + B_i = 0 \tag{3.19}$$

where B_i are the body forces in the i direction. Using small-displacement theory, the strains for an isotropic material can be expressed using the following relationship:

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{3.20}$$

In this equation, u_i is the displacement of the material. The plane stress assumption dictates that:

$$\begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{pmatrix} = \frac{E}{1-\nu^2}\begin{pmatrix} \epsilon_{xx} + \nu\epsilon_{yy} \\ \epsilon_{yy} + \nu\epsilon_{xx} \\ \left(\frac{1-\nu}{2}\right)\epsilon_{xy} \end{pmatrix} \tag{3.21}$$

where E is the modulus of elasticity and ν is Poisson's ratio. Combining Equations 3.19, 3.20, and 3.21 yields:

$$-\frac{\partial}{\partial x}\begin{pmatrix} \frac{E}{1-\nu^2}\left(\epsilon_{xx} + \nu\epsilon_{yy}\right) \\ \frac{E}{2(1+\nu)}\epsilon_{xy} \end{pmatrix} - \frac{\partial}{\partial y}\begin{pmatrix} \frac{E}{2(1+\nu)}\epsilon_{xy} \\ \frac{E}{1-\nu^2}\left(\epsilon_{yy} + \nu\epsilon_{xx}\right) \end{pmatrix} = \begin{pmatrix} -B_x \\ -B_y \end{pmatrix} \tag{3.22}$$

Using displacements u and v, the following are obtained:

$$U = \begin{pmatrix} u \\ v \end{pmatrix} \tag{3.23}$$

$$F_x = -\begin{pmatrix} \frac{E}{1-\nu^2}\left(\frac{\partial u}{\partial x} + \nu\frac{\partial v}{\partial y}\right) \\ \frac{E}{4(1+\nu)}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \end{pmatrix} \tag{3.24}$$

$$F_y = -\begin{pmatrix} \frac{E}{4(1+\nu)}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \\ \frac{E}{1-\nu^2}\left(\frac{\partial v}{\partial y} + \nu\frac{\partial u}{\partial x}\right) \end{pmatrix} \tag{3.25}$$

$$S = -\begin{pmatrix} B_x \\ B_y \end{pmatrix} \tag{3.26}$$

Boundary condition type: 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 $\vec{F}$.

Boundary condition type: Imposed y-displacement

This boundary condition type imposes a value for displacement v at the boundary. The boundary condition is enforced in the same way as the x-displacement boundary condition. 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.

Boundary condition type: 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).

Boundary condition type: Symmetry

This boundary condition type is used whenever there is a symmetry in the problem. Taking advantage of symmetries in the geometry improves the performance of the simulation by reducing the number of computational nodes needed. The symmetry boundary condition imposes two things: zero displacement in the direction normal to the boundary, and also imposes a value of 0 to the 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 $\frac{\partial v}{\partial x} = 0$.

Problem definition

We are solving the problem of a beam under tension in the x-direction. Only a quarter of the beam is modeled, because of symmetry in both axes. Symmetry boundary conditions are imposed on the left and bottom boundary faces. The top boundary is free to move, and the right boundary face has a normal stress of $\sigma_b = 1$ imposed to it.The modulus of elasticity E was set to 7000 and Poisson's ratio ν to 0.2.

The displacements at the right boundary are used to assess the accuracy of the simulation. The theory of elasticity states that the displacments in the direction of the applied stress are:

$$u(y)|_{x=L} = \frac{\sigma_b}{E}L$$

The displacements in the other axis are governed by Poisson's ratio:

$$v(y)|_{x=L} = -\frac{\nu\sigma_b}{E}Ly$$

A third-order accurate scheme was used.

Results

The numerical results matched the analytical solution accurately, as can be seen in Figure 3.8.

3.7.3 Laminar incompressible Navier-Stokes

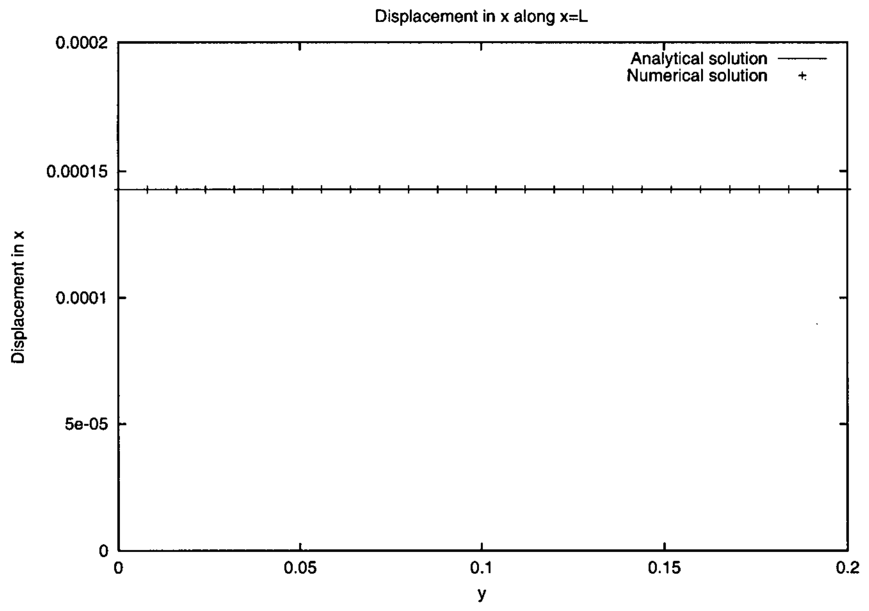
Physical description

The incompressible Navier-Stokes equations can be put in the form of Equation 3.6 by adding a time-dependent pressure term in the continuity equation, as described in [22]. The continuity equation now looks like this:

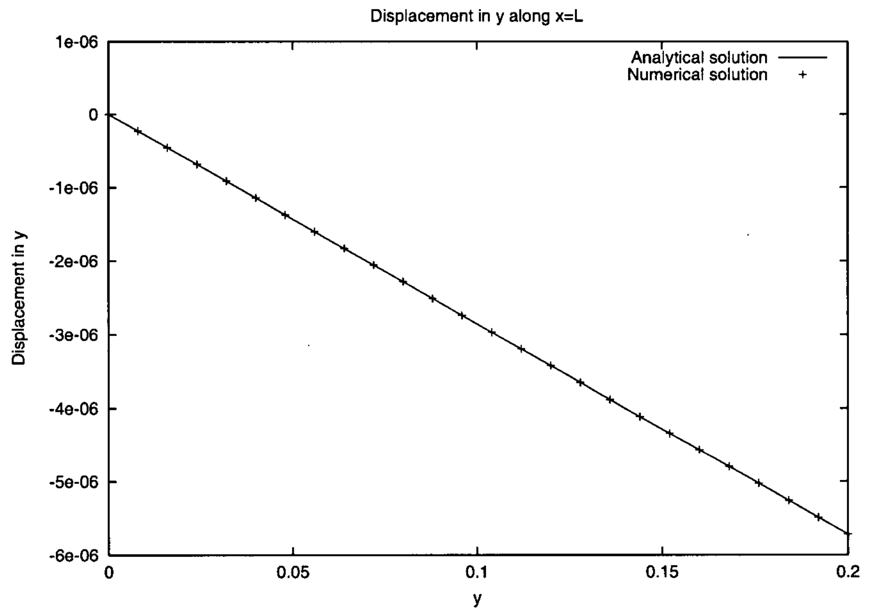
$$\frac{\partial P}{\partial t} + \frac{1}{\beta}\frac{\partial u_i}{\partial x_i} = 0 \tag{3.27}$$

where β is an artificial compressibility parameter. The smaller the value of β, the more "incompressible" the equations, although very small values of β make the equations stiff numerically. Of course, the addition of this time dependent term to the continuity equations means that the real incompressible equations are no longer being solved. However, when the solution reaches steady-state, the time derivative is zero and Equation 3.27 becomes the continuity equation. This method therefore can only be used to obtain steady-state solutions.

The Navier-Stokes momentum equations remain the same:



(a) x-displacement



(b) y-displacement

Figure 3.8: Displacements along $x = L$

$$\frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} = -\frac{\partial P}{\partial x_i} + \frac{1}{\mathrm{Re}} \frac{\partial^2 u_i}{\partial x_k \partial x_k} \tag{3.28}$$

where u_i are fluid velocities and Re is the Reynolds number of the flow (from [43]).

If one converts these equation to the form given by Equation 3.6, the following relationships are obtained for a two-dimensional problem:

$$U = \begin{pmatrix} P \\ u \\ v \end{pmatrix} \tag{3.29}$$

$$F_x = \begin{pmatrix} \frac{u}{\beta} \\ u^2 + P - \frac{1}{Re}\frac{\partial u}{\partial x} \\ uv - \frac{1}{Re}\frac{\partial v}{\partial x} \end{pmatrix} \tag{3.30}$$

$$F_y = \begin{pmatrix} \frac{v}{\beta} \\ uv - \frac{1}{Re}\frac{\partial u}{\partial y} \\ v^2 + P - \frac{1}{Re}\frac{\partial v}{\partial y} \end{pmatrix} \tag{3.31}$$

$$S = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{3.32}$$

In this case, it is practical to use the flux vector $\vec{F} = F_x \cdot \hat{n}_x + F_y \cdot \hat{n}_y$ since some simplification is possible. The flux vector is:

$$\vec{F} = \begin{pmatrix} \frac{V}{\beta} \\ Vu + \left(P - \frac{1}{Re}\frac{\partial u}{\partial x}\right)\hat{n}_x + \left(\quad - \frac{1}{Re}\frac{\partial u}{\partial y}\right)\hat{n}_y \\ Vv + \left(\quad - \frac{1}{Re}\frac{\partial v}{\partial x}\right)\hat{n}_x + \left(P - \frac{1}{Re}\frac{\partial v}{\partial y}\right)\hat{n}_y \end{pmatrix} \tag{3.33}$$

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

Boundary condition type: Inflow

This boundary condition type imposes a fully-developed normal velocity profile at the boundary. The fully-developed condition imposes that $\frac{\partial V_b}{\partial n} = 0$. The tangential velocity is set to zero. The pressure boundary condition is determined using the normal momentum equation and the fully-developed condition, and results in:

$$\frac{\partial P}{\partial n} = \frac{1}{Re} \frac{\partial^2 V_b}{\partial c^2} \tag{3.34}$$

where c represents the cross-stream direction. Since V_b 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.

Boundary condition type: Stationary wall

At the wall, velocities u and v are zero, due to the no-slip condition. Furthermore, $\frac{\partial P}{\partial n}$ is also set to zero, from Equation 3.34. These three conditions are enforced using boundary constraints, and the interior flux is used on that boundary face.

Boundary condition type: 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.

Problem definition

We are solving a simple channel flow problem. An inflow boundary condition is imposed at the left boundary of the domain, an outflow at the right face, and both the top and bottom surfaces are considered stationary walls. The constants used were $Re = 50$ and $\beta = 5$. The velocity profile at the inflow is constant:

$$V_b = 1$$

The profile at the outlet will be used to verify the accuracy of the simulation. The exit profile should be the laminar fully-developed parabolic profile:

$$u|_{x=L} = 4U_{\max}\frac{y}{H}(1 - \frac{y}{H})$$

Using conservation of mass, this profile should have a maximum velocity of $U_{\max} = 1.5$. A third-order accurate scheme was used.

Results

The velocity profile obtained matches the fully-developed profile, as can be seen in Figure 3.9. The error on the maximum velocity was of the order of 3×10^{-4}, or about 0.02%. This error is due to the fact that the flow rate entering the system is not exactly V_bH. This discrepancy is caused by the conflicting boundary conditions for the control volumes located in the left corners of the domain. Reconstruction tries to set $u = 0$ at the top and bottom walls, but is also trying to satisfy the inlet boundary condition, where $u \neq 0$. In fact, the velocity profile at the outlet matches the fully-developed profile for the real flow rate entering the domain.

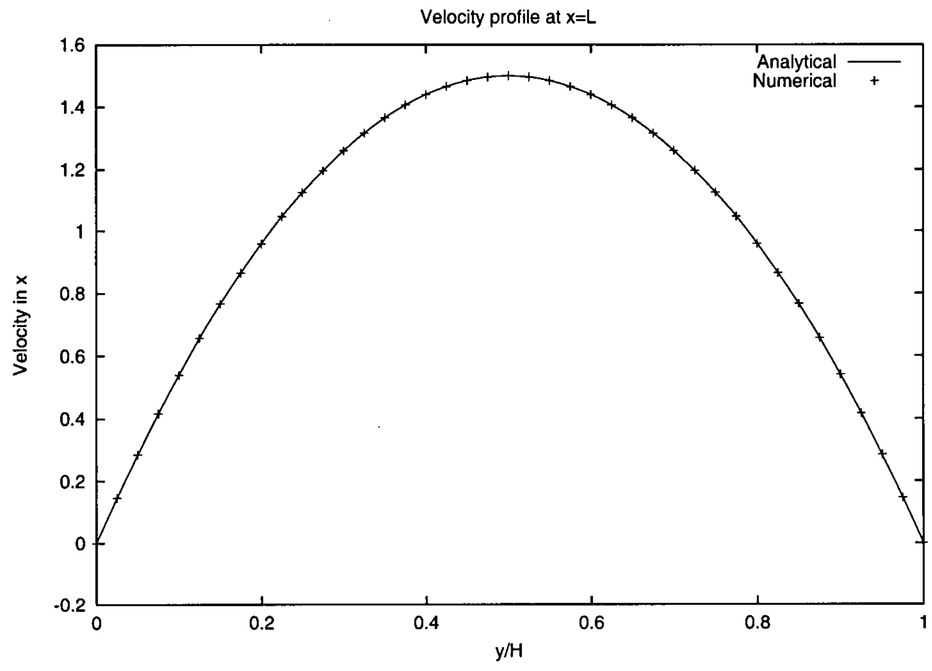


Figure 3.9: Velocity profile at $x = L$

Summary

In this chapter, the details of the generic numerical toolkit used in this research have been presented. The finite-volume method has been described in Section 3.1. Section 3.2 covered reconstruction, the method used by the toolkit to achieve high-order accuracy on unstructured meshes. Boundary conditions and time-advance methods were then covered next in Sections 3.3 and 3.4.

The generic solver concept was introduced in Section 3.6; the different classes needed by a generic solver are presented in this section. Finally, in Section 3.7, three different physical problems were solved on the same mesh to demonstrate the effectiveness of the generic nature of the solver.

Chapter 4

Generic Multiphysics Solver

Multiphysics problems are problems where 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* [21].

4.1 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.

4.2 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 problem is far more complex. The fluid region dictates the stresses in the solid region at the boundary: the normal stress must be equal to the pressure in the fluid, and the shear stress must be equal to the wall shear stress in the fluid. The solid region, in turn, influences the fluid region 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).

4.3 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 [7, 8]. In fact, single-physics problems are now treated as special cases of multiphysics problems.

One of the obvious changes needed was that the solver must now interact with multiple `Physics` classes. Another change was to allow multiple regions, or subdomains, in a problem, each with their own set of `Physics` classes. But perhaps the most crucial modification was allowing the exchange of information between different `Physics` classes, whether on the same domain, or across an interface. This requirement had a serious impact on how the interface for the physics of a problem was modified, and is therefore introduced first in Section 4.4. The following sections, Section 4.5 and Section 4.7, then introduce new classes that were created to simplify the task of managing multiple physical phenomena, and multiple subdomains in a problem, respectively. Results are presented in Sections 4.6 and 4.8.

4.4 Data exchange

The first requirement for data exchange is that the solver must know what information each physical package can exchange with another. This is accomplished by the use of another class, 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] There are different types

[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.

of variables needed by the `Physics` class, and the types are outlined in Section 4.4.1.

Once all the `Physics` classes have listed their variable requirements, it is then possible to couple the classes together. The coupling is accomplished through *variable association*, and this process is covered in Section 4.4.2.

Additionally, some computations can only take place when the values for all dependent variables are available. This requirement is fulfilled by the use of dependency trees. Details are presented in Sections 4.4.3 and 4.4.4. Finally, a summary of all the modifications made to the `Physics` class to make use of `PhysVars` is given in Section 4.4.5

4.4.1 Variable types

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; each is given an identification number, a name, a symbol, units, and a variable type. 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 variable 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. 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 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).

Computed variables

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.

A computed variable is another quantity that the `Physics` class can compute, besides fluxes and source terms. This type of variable is particularly useful 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.

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 uses the identification number of the variable to know which quantity to compute, and returns the value to the solver.

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.

4.4.2 Variable association

Now that the `Physics` class can inform the solver on all the variables it needs and provides, coupling the different physical packages is straightforward conceptually. For this task, *variable association* between a required and a provided variable is used. The association is done by having the solver link the two variables together using a pointer. This way, whenever the solver needs the value of a given required variable, it automatically uses the value of the provided variable associated with it.

4.4.3 Variable dependencies

In certain cases, the value for some variables can not be fetched until the value of other variables are available. For example, computed variables typically need a list of other variables values to be available before they can be computed and used by the solver. 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 4.4.1). 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.

4.4.4 Dependency trees

It was mentioned above that 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. The only way a variable value is available is if the value for all of its dependencies was 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, shown in Figure 4.1. 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. Figures 4.1a 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 4.1b. Figure 4.1c 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 4.1d 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 4.1e. Finally, Figure 4.1f 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

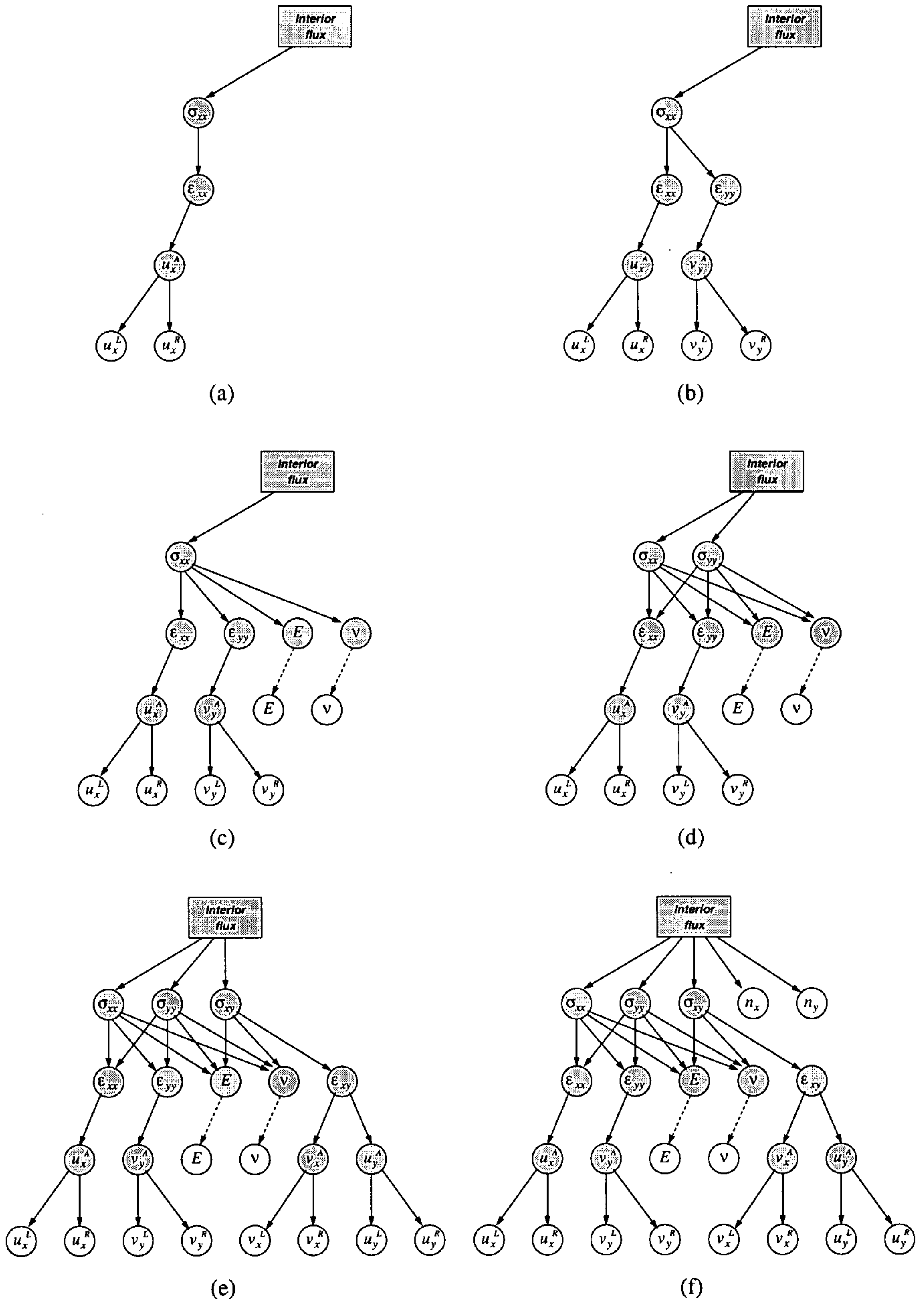


Figure 4.1: Dependency tree for the interior flux computation of the solid mechanics package

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 value 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 4.2.

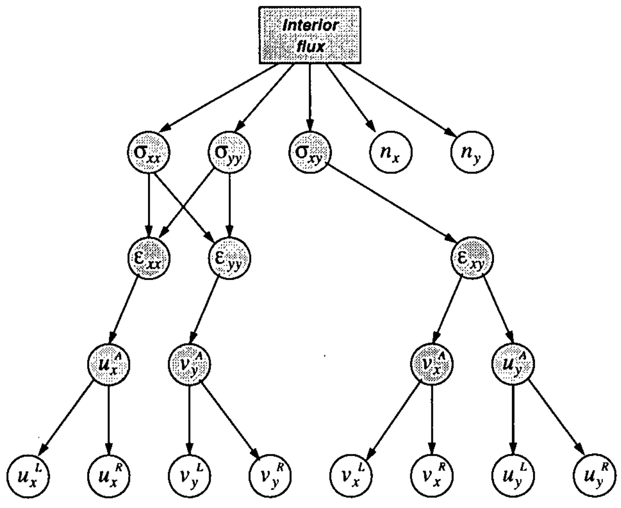


Figure 4.2: Optimized dependency tree for the solid mechanics package with constants removed

4.4.5 Modifications to the `Physics` class

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

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 (the `vecValues[]` array) that is used to store the values of each of the `PhysVars`.

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.

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.

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.

4.5 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 shy away from 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 proper super-`Physics` class would cover 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 4.3.

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.

4.5.1 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 `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

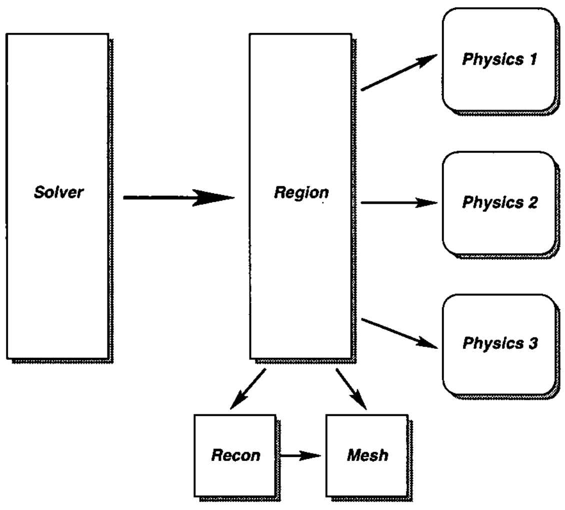


Figure 4.3: Schematic of the generic multiphysics framework for field coupling

association of boundary condition number to boundary condition type in the `Physics` classes is correct.

4.5.2 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 also 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 also stores the dependency trees needed for interior flux, boundary flux, boundary constraint and source term evaluation. 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.

4.5.3 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 Recon 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.

4.6 Field coupling results

The field coupling infrastructure introduced in this section allowed the solution of the problems presented in this section. These problems both use two Physics classes that were introduced in the previous chapter.

4.6.1 Solid mechanics and heat conduction

The problem being solved is that of a bar subjected to a vertical temperature gradient. The solid mechanics package was modified to also account for thermal strains. The mechanical strains for the problem are now:

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \delta_{ij}\alpha_T(T - T_{ref}) \tag{4.1}$$

where α_T is the coefficient of thermal expansion, T_{ref} is the zero-strain temperature, and T is the temperature of the solid. The fluxes were modified accordingly. The temperature distribution is computed by the heat conduction package. The schematic of the problem is shown in Figure 4.4.

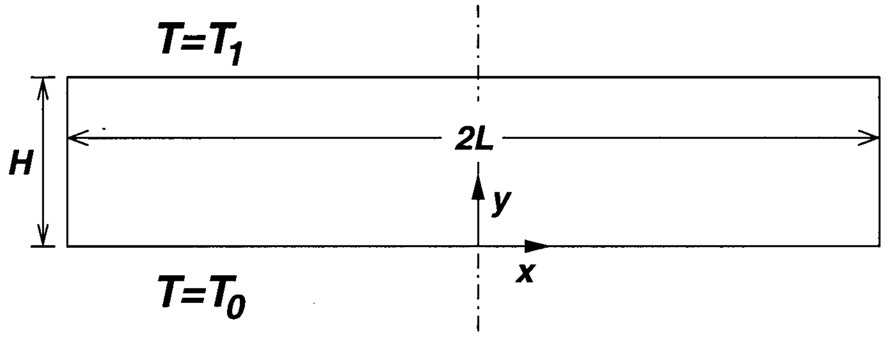


Figure 4.4: Sample solid mechanics with thermal strains problem

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. 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 considered fixed at $(0, 0)$.

The temperature distribution will be linear:

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

From this temperature distribution, the bar will deform according to the following values:

$$\begin{aligned} u(x,y) &= \frac{\alpha_T(T_1 - T_0)xy}{H} \\ v(x,y) &= \frac{\alpha_T(T_1 - T_0)}{2H}\left(y^2 - x^2\right) \end{aligned}$$

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 4.5.

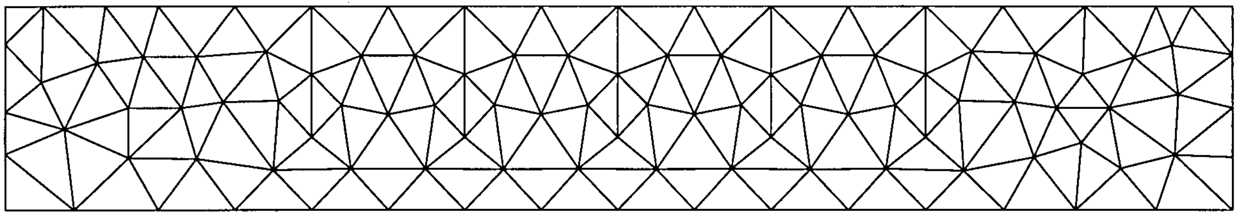


Figure 4.5: Mesh used for the heat conduction and solid mechanics simulation

Results

The displacements in the beam matched the exact solution well, as can be seen in Figures 4.6 and 4.7. This indicates that the temperature results were also accurate, since the displacements are caused by the temperature field. This was expected, since a third-order accurate scheme recovers the fluxes, and the quadratic nature of the solution exactly.

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}.

4.6.2 Incompressible Navier-Stokes and the energy equation

The incompressible energy equation will be solved in conjunction with the Navier-Stokes package introduced in the previous chapter. The energy equation will have its own `Physics` package. The partial differential equation is:

$$\frac{\partial T}{\partial t} + \frac{\partial (uT)}{\partial x} + \frac{\partial (vT)}{\partial y} = \frac{1}{Re \cdot Pr}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) + \frac{Ec}{Re}\Phi \tag{4.2}$$

where Φ is the two-dimensional dissipation term, a source term in the equation:

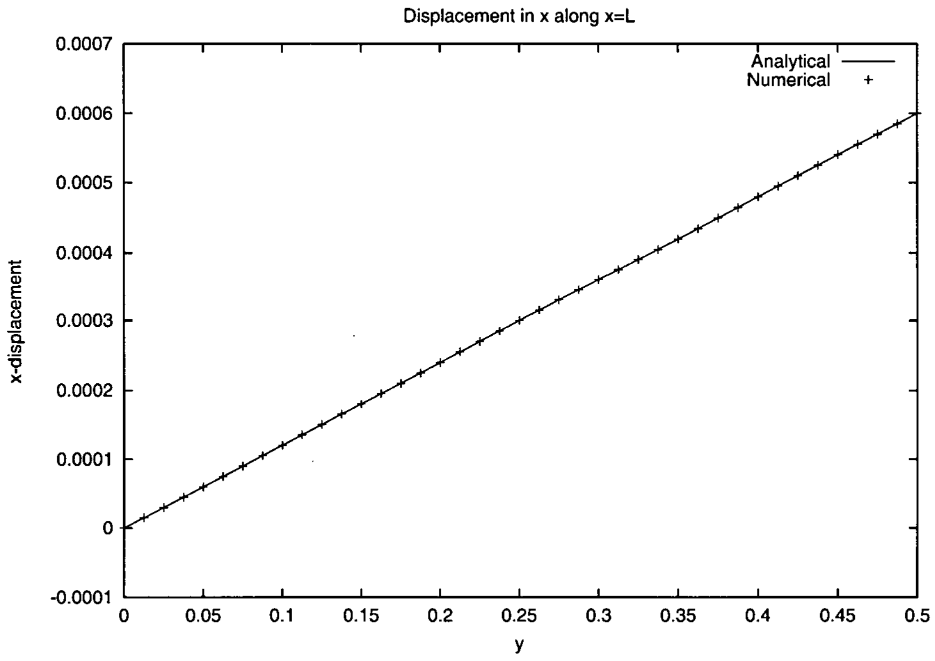


Figure 4.6: Displacement in x along $x = L$

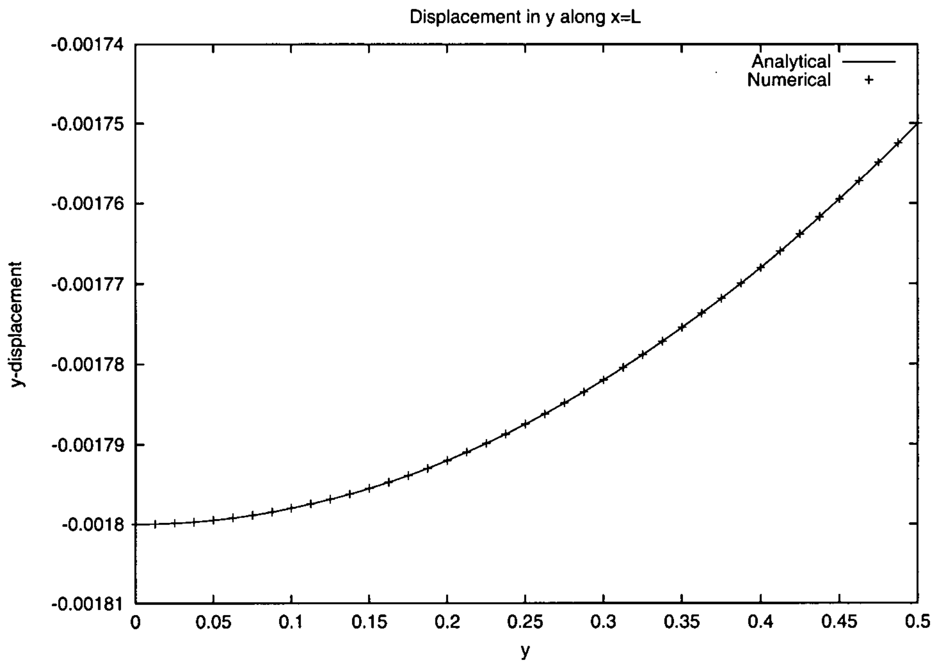


Figure 4.7: Displacement in y along $x = L$

$$\Phi = 2\left(\frac{\partial u}{\partial x}\right)^2 + 2\left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)^2 \tag{4.3}$$

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:

$$T = T_H\left(\frac{y}{H}\right) + T_0\left(1 - \frac{y}{H}\right) + 6\mathrm{Pr}\cdot\mathrm{Ec}\left(\left(\frac{y}{H}\right) - 3\left(\frac{y}{H}\right)^2 + 4\left(\frac{y}{H}\right)^3 - 2\left(\frac{y}{H}\right)^4\right) \tag{4.4}$$

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

$$T = y + 6\mathrm{Pr}\cdot\mathrm{Ec}\left(y - 3y^2 + 4y^3 - 2y^4\right)$$

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 $T = y + 5.5\mathrm{Pr}\cdot\mathrm{Ec}(y - 3y^2 + 4y^3 - 2y^4)$ 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 nature of the temperature profile. The mesh used in this simulation is shown in Figure 4.8.

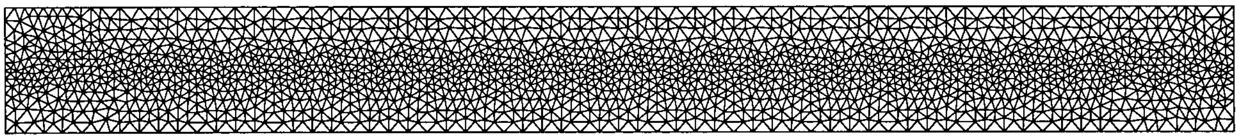


Figure 4.8: Mesh used for the Navier-Stokes and energy equation simulation

Results

The results shown in Figure 4.9 were taken along $x = 9$, where the temperature had reached its fully-developed profile.

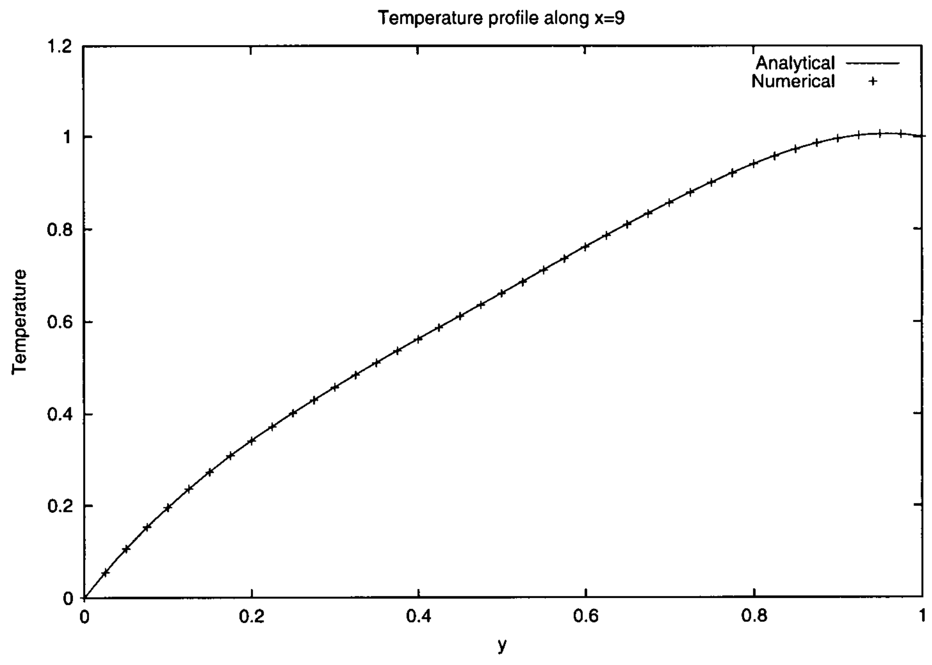


Figure 4.9: Temperature profile along $x = 9$ for the Navier-Stokes and energy equation simulation

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.

4.7 The `Domain` class

The `Domain` class is used to handle the additional requirements of interface coupling. One of these requirements is the ability to handle multiple subdomains, on which field coupling might still occur. Another requirement, and perhaps the most important, is to allow coupling between `Physics` classes located on different subdomains, through a common interface.

The `Domain` class is an example of the high leverage possible with the object-oriented approach. It was mentioned above 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 4.10. The interface coupling is handled through boundary conditions. More details are given in the section below.

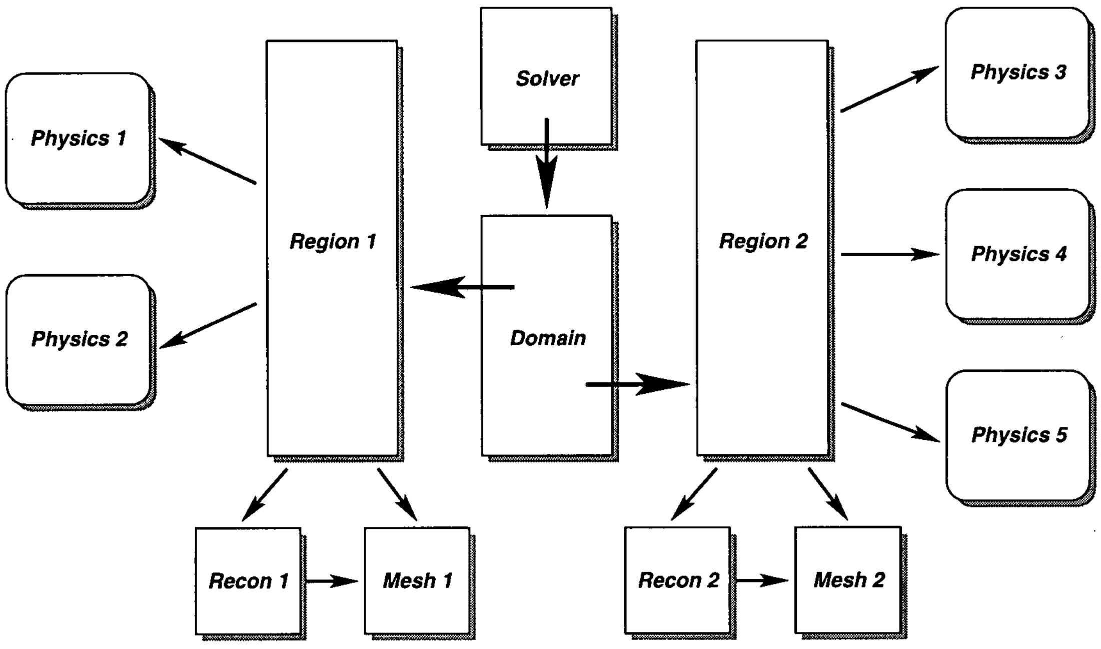


Figure 4.10: Schematic of a problem with both field and interface coupling

4.7.1 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, or order. 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, it 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.

4.7.2 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, and some other provided variable. Since the `Domain` class has access to all the `Region` objects, it can access the master list of variables for each, and assign pointers to a required variable in a `Region` class to a provided variable in another subdomain. 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.

4.7.3 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. In the heat transfer case, one side

[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.

of the interface enforces matching temperatures (by using a temperature boundary condition with the value determined by the temperature from the other side), and the other side enforces matching heat fluxes (using a heat flux boundary condition, with the value of the heat flux computed on the other side as well). 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 were Neumann boundary conditions are 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 construction, 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 4.8.

4.8 Interface coupling results

The problem solved in this section will make use of the same `Physics` classes introduced in previous chapters. The problem is a conjugate heat transfer problem. The domain, along with the `Physics` classes being used in each subdomain, is represented in Figure 4.11. Regions are identified using roman numerals.

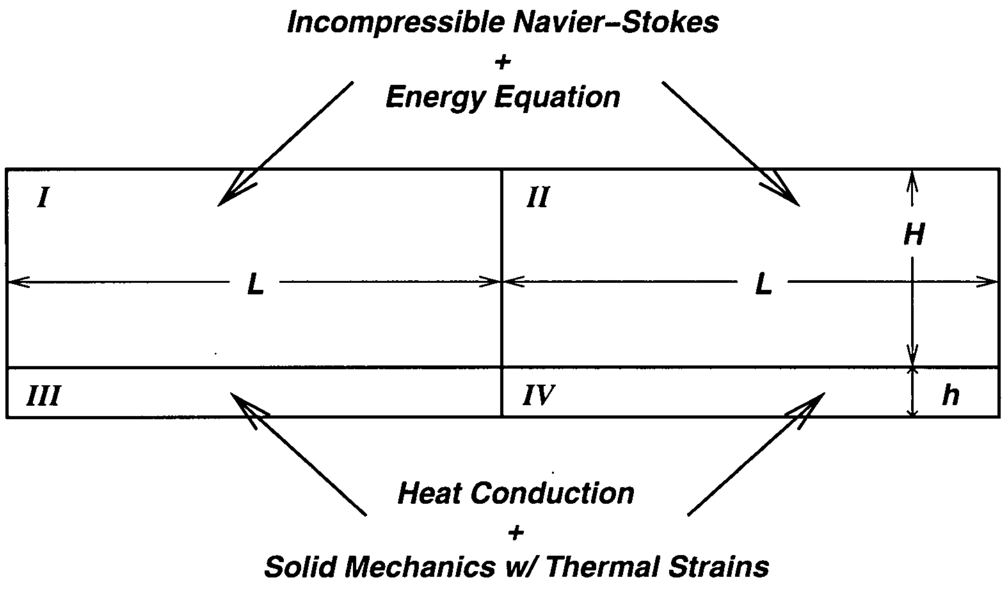

Incompressible Navier-Stokes
+
Energy Equation
I
II
III
IV
L
L
H
h
Heat Conduction
+
Solid Mechanics w/ Thermal Strains

Figure 4.11: 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 4.6.2. 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 4.6.1.

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 4.7.3. 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 $L = 2.5$, $H = 1$,*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 determined 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 the parabolic fully-developed profile, as presented in Section 3.7.3. For this reason, the flow velocity results will not be repeated here.

The temperature profile at $x = 4.7$ is used to determined 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 meh 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 4.1 lists the number of cells in all the meshes used.

Refinement level	Mesh I	Mesh II	Mesh III	Mesh IV
1	507	235	25	25
2	507	348	100	100
3	507	1117	225	225
4	507	1374	400	400

Table 4.1: Number of cells used in the meshes

The temperature profiles at $x = 4.7$ for the different refinement levels are shown in Figure 4.12. 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.

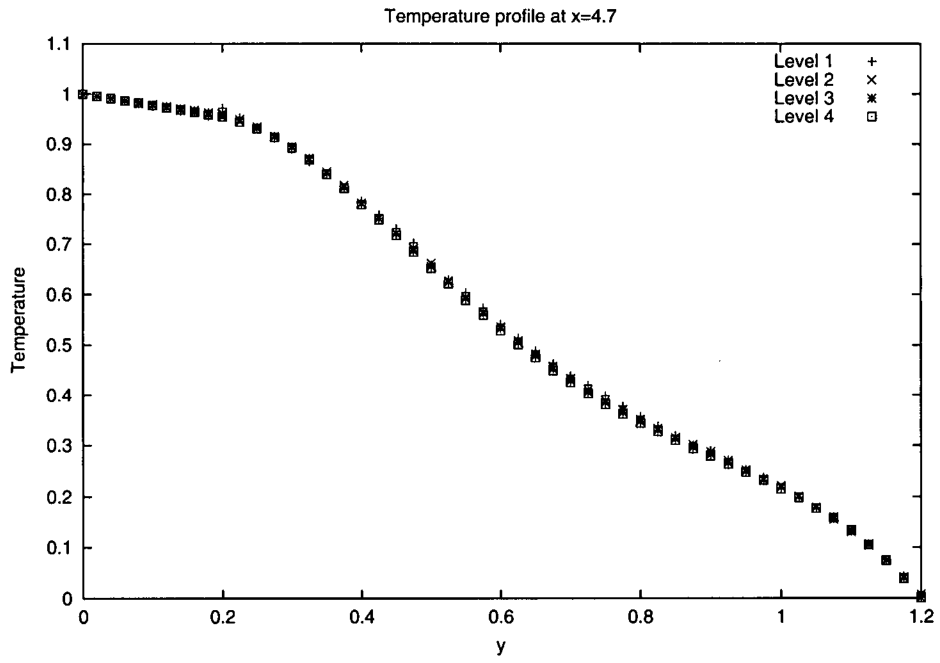


Figure 4.12: Temperature profiles at $x = 4.7$ for the interface coupling problem for various refinement levels

The temperature field is plotted in Figure 4.13. It can be seen that the temperature is continuous across the fluid-solid interface, and that the dissipation term in the energy equation causes the temperature gradient to be non-uniform across the fluid channel.

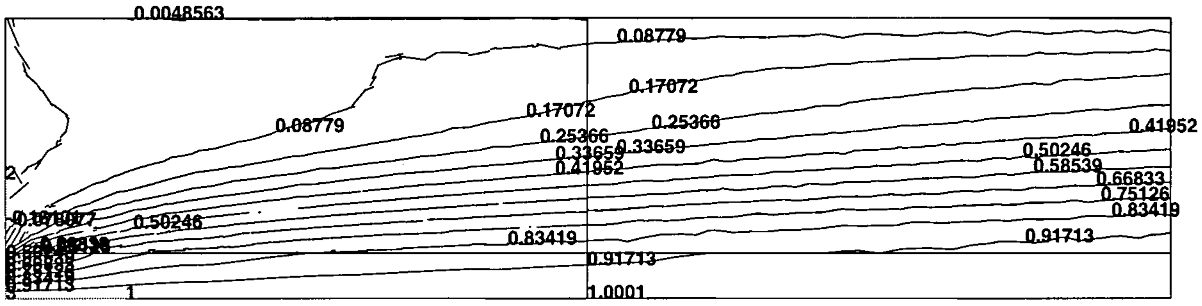


Figure 4.13: Temperature field for the interface coupling problem

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 4.14 displays the temperature along $y = 0.2$ at the interface between regions II and IV, and Figure 4.15 shows the normal heat transfer along the same interface.

Figure 4.14 shows that the temperature profiles from both regions match well at the interface. The results for the heat flux, from Figure 4.15, 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. It should also be noted that the energy in the problem is conserved even with these discrepancies: the heat flux in the solid region is determined by the heat flux computed in the fluid region. Therefore, the fluxes that are used for the energy term are the same in both regions at the interface. However, the heat flux that is *computed* in the solid region is different (as can be seen in Figure 4.15), but this is not the flux used to update the solution in the boundary control volumes; it has no effect on the conservation of energy in the problem.

Thermal strains in the solid region

The displaced geometry in the solid region are shown in Figure 4.16; only the displacements from region IV are shown here for clarity. The displacements are magnified by a factor of 1000 in x

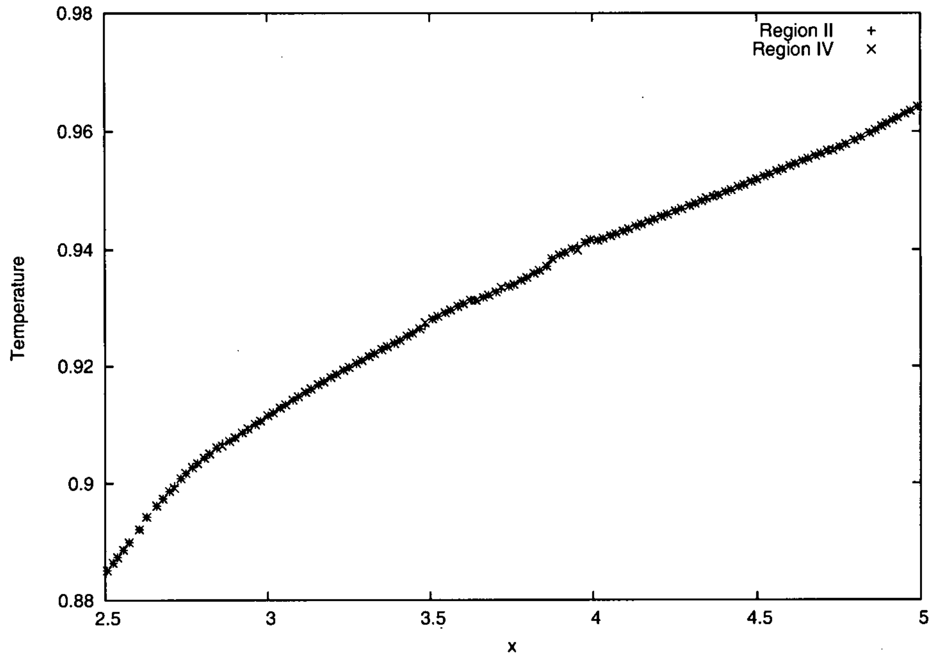


Figure 4.14: Temperature along the interface at $y = 0.2$

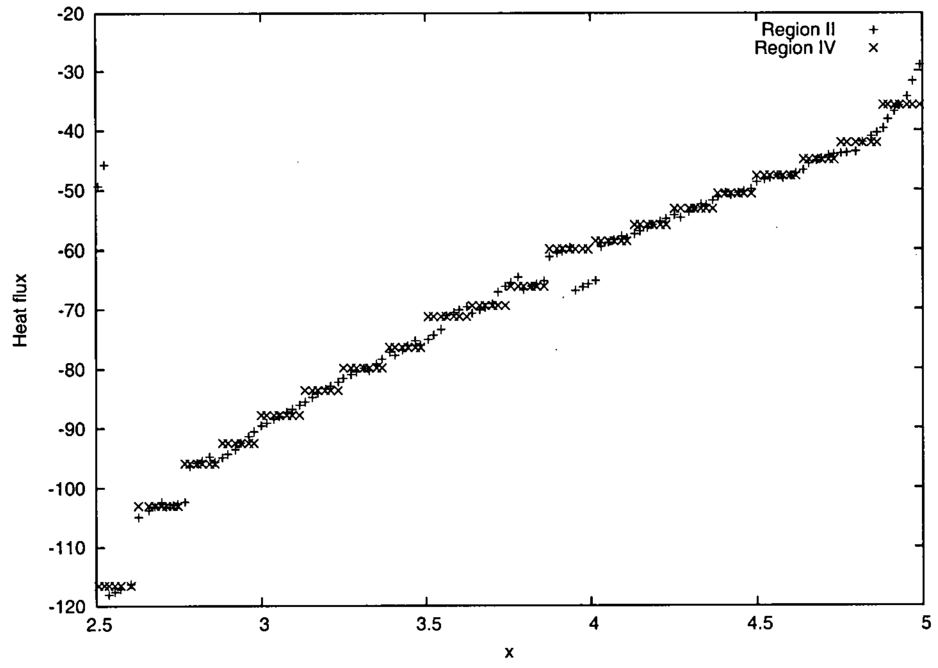


Figure 4.15: Normal heat flux along the interface at $y = 0.2$

and a factor of 10000 in y. The maximum displacements observed were $u = 9.17 \times 10^{-4}$ and $v = -3.93 \times 10^{-5}$.

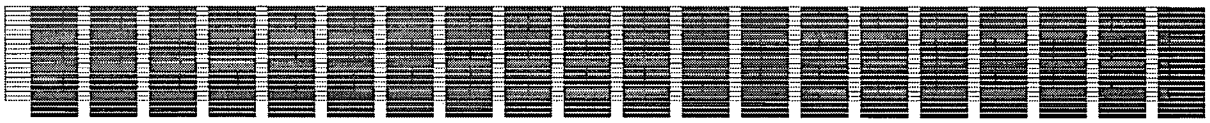


Figure 4.16: Displaced geometry in region IV for the interface coupling problem

The maximum displacements observed are reasonable, considering that a solid region with a uniform temperature of 1 would have observed 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.

Summary

In this chapter, the features that were added to the generic numerical toolkit to solve generic multiphysics problems were presented. A discussion of the different types of coupling present in multiphysics problems was given in Sections 4.1 and 4.2. Data exchange between the different physical phenomena was the most crucial part of this research; Section 4.4 presented how this was implemented in the solver.

Section 4.5 introduced the `Region` class, which was added to the toolkit as a way to manage multiple `Physics` classes. Field coupling problems could then be solved, and results of such problems were shown in Section 4.6. The `Domain` class was also created to allow multiple subdomains to communicate with each other; details were presented in Section 4.7. Interface coupling problems were then solved in Section 4.8.

Chapter 5

Additional Features

The features presented in this chapter were not covered previously, because they did not play a large role in the design of the multiphysics framework. Nonetheless, they are worth mentioning since they address specific problems encountered when trying to solve generic multiphysics problems.

Section 5.1 introduces the concept of source term exchange, which can be useful for solving combustion problems. Additionally, a parsing tool written to help the creation of new `Physics` classes is presented in Section 5.2.

5.1 Source term exchange

Source term exchange refers to the possibility of a `Physics` class computing source terms for flux variables in other `Physics` classes. This feature was added for future use with combustion problems in mind, but can also be useful for other types of problems.

5.1.1 Overview

In field coupling problems, some `Physics` packages have source terms that affect flux variables from other `Physics` classes. A chemistry package for combustion is an example of such package. This `Physics` class contains the equations that keep track of chemical interactions needed for the simulation of combustion. It would typically be used in conjuction with the Navier-Stokes and energy equation packages. The chemical interactions, among other things, generate or absorb heat in the system; this can be represented as source terms in the energy equation. The energy equation package would have to be modified in order to account for such source terms. However, a change in the type of chemistry being used would also require a change in the energy equation package; an undesirable effect, at best.

To keep `Physics` classes independent from each other as much as possible, the source term function from the chemistry `Physics` class would need access to the fluxes from the energy equation package. This is impossible to do using the framework defined so far; `Physics` class functions (such as interior and boundary flux, or source term functions) only have access to the fluxes for their own flux variables.

5.1.2 Changes made to the source term framework

An exception was made for the source term function to allow packages such as combustion packages to be used efficiently with other `Physics` classes without any modification. The solution is rather simple: the source term function is given access to the flux vector for *all* flux variables in the `Region`, not just its own flux variables. It is now the responsibility of the user to assign a source term to the proper flux variable in the `Region`, as the `Physics` class does not know which other flux variables are present in the simulation.

Each source term is given a description to help the user map the source terms to flux variables. The `Physics` class can have any number of source terms, and each of them must be assigned to a flux variable. When computing source terms, the `Region` class asks each `Physics` class in turn to compute its own source terms. The vector being passed to a `Physics` class is sized according to the total number of flux variables in the `Region`. Each source term is stored in the proper location by the `Physics` class, according to the mapping determined by the user. The values in the vector are then added to the residuals for each control volume before the `Region` asks the next `Physics` class to compute its source terms.

The only inconvience to this method is that source terms must always be mapped to flux variables by the user, even in cases where the source terms only affect the flux variables computed by the same `Physics` class. However, this inconvenience is outweighed by the ability to solve problems with complex interactions, such as those present in combustion problems.

5.2 **Physics** class definition syntax

A simple syntax, based on XML (eXtensible Markup Language), has been created to simplify the task of writing new Physics classes. The XML format describes the class using human-readable keywords. The XML file is automatically converted to C++ by a parsing program written specifically for that purpose. The governing principles of XML syntax, as well as the specific keywords used to define a Physics class, are described in this section.

5.2.1 XML principles

XML is an extensible markup language used to describe an object using a tree model. XML is extensible because any keyword can be used. The XML document forms a tree, and each node in the tree has a value. Each node can also have multiple *child nodes* below it.

XML nodes are defined using *tags*. A node, and its associated value, are defined in the following way: **<NodeName>**NodeValue**</NodeName>**. Every XML document begins with a special tag that identifies the XML version used. For our use, this tag is the following: **<?xml version="1.0" ?>**.

The Physics definition file uses a single main node, with a tag name of **<Physics>**. All other nodes in the document are child nodes of the **<Physics>** node. An overview of a Physics class definition file looks like this:

```
<?xml version="1.0" ?>
<Physics>
     child nodes are defined here
</Physics>
```

5.2.2 Class description

These tags are used to identify the Physics class. Two tags are used. The first one is the **<ClassID>** tag, and its value will be used by the solver to call the Physics class. This value must be a single word and unique, i.e. no other Physics class can use the same value.

The other tag is used to provide the user with a human-readable description of the Physics class. The tag used is **<ClassDesc>**. Here is an example of these two tags from the heat conduction package:

```
    <ClassID>Heat2D</ClassID>
    <ClassDesc>2D Heat conduction package</ClassDesc>
```

5.2.3 Variables

The variables used in the Physics class must be defined in the file, each as a child node of the **<Physics>** node. It is recommended to define these variable immediately following the class description tags (although the order of child nodes in the document is irrelevant), as they are used in flux definitions later on.

Only flux, required, computed and constant variables are defined in the XML file. The definitions for the reconstruction variables needed by the `Physics` class are created automatically by the parsing program. The special keywords for reconstruction variables are described below.

All variables must be given an unique identifier string, using the **<ID>** tag.[1] This ID string is used to refer to the variable in the definition file. The **<Name>** tag is used to describe the variable to the user, and the **<Symbol>** tag is used for short form representation. Finally, the **<Units>** tag is used to describe the units of the variable, by providing the power of each dimension (length, mass, time or temperature). The **<Units>** node contains child nodes **<M>** for mass, **<L>** for length, **<T>** for time and **<K>** for temperature.

As a reminder, here is a system of partial differential equations, with the notation used in the solver:

$$\frac{\partial}{\partial t}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix} f_{x_1} \\ f_{x_2} \\ \vdots \\ f_{x_n} \end{pmatrix} + \frac{\partial}{\partial y}\begin{pmatrix} f_{y_1} \\ f_{y_2} \\ \vdots \\ f_{y_n} \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \tag{5.1}$$

In this equation, $U = (u_1, u_2, ...u_n)^T$ is the vector of flux variables, also referred to as the unknowns of the problem, $F_x = (f_{x_1}, f_{x_2}, ...f_{x_n})^T$ is the flux vector in the x-direction, $F_y = (f_{y_1}, f_{y_2}, ...f_{x_n})^T$ is the flux vector in the y-direction, and $S = (s_1, s_2, ...s_n)^T$ is the source term vector.

[1]The ID string only needs to be unique within the `Physics` class.

Flux variables

Flux variables, i.e. each variable in vector U, are defined using the **<FluxVar>** tag. For the heat conduction package, there is only one flux variable: temperature. It is defined as:

```
<FluxVar>
  <ID>eFlux_T</ID>
  <Name>Temperature</Name>
  <Symbol>T</Symbol>
  <Units>
    <L>0</L>
    <M>0</M>
    <T>0</T>
    <K>1</K>
  </Units>
</FluxVar>
```

Required variables

Required variables are variables needed by the `Physics` class, such as material properties, etc. They are defined using the **<RequiredVar>** tag. The heat conduction package requires k, ρ, and c_p. The conductivity variable is defined as follows:

```
<RequiredVar>
  <ID>eConduct</ID>
  <Name>Conductivity</Name>
  <Symbol>k</Symbol>
  <Units> <L>1</L> <M>1</M> <T>-3</T> <K>-1</K> </Units>
</RequiredVar>
```

Computed variables

Computed variables are variables that the `Physics` class can provide to other classes. They can be used for boundary conditions, post-processing, or to simplify the definition of fluxes. For the heat conduction package, the heat diffusivity is required to compute the interior flux. Heat diffusivity is defined as $\alpha = \frac{k}{\rho c_p}$. Computed variables have an extra tag, **<Formula>**, to specify how they are computed. The formula must be given using known variable identifiers and standard C language mathematical rules.

```
<ComputedVar>
  <ID>eHeatDiff</ID>
  <Name>Heat Diffusivity</Name>
  <Symbol>a</Symbol>
  <Units> <L>2</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  <Formula>eConduct / (eDens * eSpecHeat)</Formula>
</ComputedVar>
```

The known identifiers in the **`<Formula>`** tag will be expanded in the C++ file. In this example, `eConduct`, `eDens` and `eSpecHeat` are the identifier strings for the k, ρ and c_p variables, respectively. The C++ output for the formula looks like this:

```
dValue = vecValues[eConduct] / (vecValues[eDens] * vecValues[eSpecHeat]);
```

Constant variables

Constant variables are also provided variables, but their values never change. The extra tag **`<Value>`** is used to specify the value of the constant. The heat conduction package does not have any constant variables, but here is an example of a constant variable defined in a "database" `Physics` class.

```
<ConstantVar>
  <ID>eTempB</ID>
  <Name>Steady Temp B</Name>
  <Symbol>TbB</Symbol>
  <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
  <Value>10.0</Value>
</ConstantVar>
```

Reconstruction variables

Reconstruction variables provide access to the data computed by the solver. This data includes the value of the flux variables and their gradients, as well as the location and the normal vectors. These variables do not have to be defined using tags as other variables. Instead, they all have specific identifiers. All these identifiers require a parameter given between `{` and `}`. The reconstruction variables identifiers are:

- **`Loc{}`**: Provides a value of the location vector. Accepted parameters are: `X`, `Y`, and `Z`. Example: `Loc{X}`.
- **`Norm{}`**: Provides a value of the normal vector. Accepted parameters are: `X`, `Y`, and `Z`. Example: `Norm{X}`.
- **`LSoln{}`**: Provides the *left* value of a flux variable. Accepted parameters are identifiers of flux variables defined using **`<FluxVar>`** tags. Example: `LSoln{eFlux_T}`.
- **`RSoln{}`**: Same as above, except it provides the *right* value of a flux variable.
- **`Soln{}`**: Provides the *average* of the left and right values of a flux variable. Accepted parameters are identifiers of flux variables defined using **`<FluxVar>`** tags. A computed variable returning the average of the left and right values of flux variable will automatically be created in the C++ file. Example: `Soln{eFlux_T}`.
- **`LGradX{}`, `LGradY{}`, `LGradZ{}`**: Provides the *left* value of $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, or $\frac{\partial}{\partial z}$ of a flux variable. Accepted parameters are identifiers of flux variables defined using **`<FluxVar>`** tags. Example: `LGradY{eFlux_T}`.
- **`RGradX{}`, `RGradY{}`, `RGradZ{}`**: Same as above, except it provides the *right* value of a flux variable.
- **`GradX{}`, `GradY{}`, `GradZ{}`**: Provides the *average* of the left and right values of the gradient of a flux variable. See **`Soln{}`.**

5.2.4 Interior flux

The interior flux is defined within the **`<InteriorFlux>`** tag. The flux for each flux variable must be specified using a **`<FluxData>`** tag. For our partial differential equations, one should define

$\vec{F_1} = f_{x_1}\vec{n}_x + f_{y_1}\vec{n}_y$, $\vec{F_2} = f_{x_2}\vec{n}_x + f_{y_2}\vec{n}_y$, etc. The **<FluxData>** tag requires two child nodes: a **<Var>** node that specifies for which flux variable this interior flux is to be used, and a **<Formula>** node that gives the value of the interior flux itself. As in computed variables, known identifiers in the **<Formula>** tag will be expanded.

The interior flux for the heat conduction equation is the following:

$$\vec{F} = -\alpha(\frac{\partial T}{\partial x}\vec{n}_x + \frac{\partial T}{\partial y}\vec{n}_y)$$

This is defined in the XML file as:

```
<InteriorFlux>
  <FluxData>
    <Var>eFlux_T</Var>
    <Formula>- eHeatDiff * (GradX{eFlux_T}*Norm{X} +
                            GradY{eFlux_T}*Norm{Y}) </Formula>
  </FluxData>
</InteriorFlux>
```

In this example, the average value of the temperature gradients are used. Other `Physics` classes would have as many **<FluxData>** tags as there are flux variables.

5.2.5 Wave Speeds

ANSLib can solve steady-state problems using local time-stepping techniques. This maximizes the timestep locally, which has the advantage of speeding up convergence. However, since the timestep is not the same everywhere in the domain, this technique is not time-accurate and hence should only be used for steady-state solutions.

The local time-stepping approach uses minimum and maximum "wavespeeds" information from the `Physics` classes to determine the maximum timestep possible. For problems with convective terms, the wavespeeds can be computed from the Jacobian of the problem. For other problems, like heat conduction, the reciprocal of a distance measure between the reference points of adjacent control volumes can be used instead. This provides a "fake" wavespeed that can be used by the solver. This value is available using the specialized keyword **eInvDist.**

The wavespeeds are defined within the **<WaveSpeeds>** tag. Two nodes, a **<Minimum>** and a **<Maximum>** value, are expected. Keywords inside the minimum and maximum values are expanded similarly to **<Formula>** nodes. Following is the example from the heat conduction package.

```
<WaveSpeeds>
  <Minimum>-eInvDist * 0.3</Minimum>
  <Maximum> eInvDist * 0.3</Maximum>
</WaveSpeeds>
```

5.2.6 Source term

The source term vector is defined within the **<SourceTerm>** tag. It contains as many **<SourceData>** tags as there are source terms defined in the `Physics` class. The **<SourceData>** tags are similar to the **<FluxData>** tags. The difference is that, as seen in Section 5.1, the source terms can act on any variable defined on the same `Region`. Therefore, the **<Var>** tag is not needed anymore. Instead, the **<SourceData>** node is treated in a similar way to computed variables: it is given an identifier **<ID>**, a description **<Desc>** as well as **<Units>**, so the user can assign the source terms to the proper flux variables. However, it keeps its **<Formula>** tag that specifies how the source term should be computed.

There are no source terms defined in the heat conduction package. The following is therefore an example of the source term definition for the energy equation package introduced in Section 4.6.2.

```
<SourceTerm>
  <SourceData>
    <ID>eDissFunc</ID>
    <Desc>Adds dissipation effects to the energy equation</Desc>
    <Units> <L>0</L> <M>0</M> <T>-2</T> <K>0</K> </Units>
    <Formula>(-eEckert/eReynolds)*
             (2.0*eStrainXX*eStrainXX + 2.0*eStrainYY*eStrainYY +
              4.0*eStrainXY*eStrainXY)</Formula>
  </SourceData>
</SourceTerm>
```

5.2.7 Boundary condition types

The `Physics` file must also define boundary condition types. Boundary condition types are defined within the **`<BCType>`** tag. There is no limit on the number of **`<BCType>`** tags in a `Physics` definition file.

Boundary conditions often require values to be passed to them. These values are either set by the user, or used for coupling purposes. The boundary condition values are stored using variables. A special variable type, the Boundary Condition variable, is used for that purpose. Boundary condition variables are defined within the <BCVar> tag, and use the same syntax as other variables, as described in Section 5.2.3, i.e. they require **`<ID>`**, **`<Name>`**, **`<Symbol>`** and **`<Units>`** tags.

Boundary conditions can be defined using either a specific boundary flux $\vec{F}_b$, by using constraints on the reconstruction, or by using a combination of both. If constraints are used, they are specified within the **`<Constraint>`** tag, and there is no limit to the number of constraints that can be specified for a given boundary condition type. The <Constraint> tag expects three child nodes: a **`<Type>`** node, a **`<Var>`** node, and a **`<Formula>`** node. The **`<Type>`** nodes indicates what type of constraint is needed. The possible values are:

- **`Solution`**: The constraint is on a flux variable itself.
- **`XGradient`**, **`YGradient`**, **`ZGradient`**: The constraint is on the x-, y-, or z-derivative of a flux variable.
- **`NGradient`**, **`TGradient`**, **`CGradient`**: The constraint is on the normal, tangential, or cross-flow derivative of a flux variable.

The value of the **`<Var>`** node determines on which flux variable the constraint acts. Finally, the **`<Formula>`** value determines the value of the constraint itself. This value is expanded for known identifiers, including the identifiers for boundary condition variables defined within the same **`<BCType>`**.

Boundary fluxes must be specified for all boundary condition types, even if the flux is the same as the interior flux. The flux is defined within the **`<BdryFlux>`** tag using **`<FluxData>`** tags. The **`<FluxData>`** tags work exactly the same way as for the interior flux definition; see Section 5.2.4.

As an example, the definition of a temperature boundary condition for heat conduction is following.

```
<BCType>
  <Desc>This BC imposes a temperature at the wall</Desc>
  <ID>eBC_Temp</ID>
  <BCVar>
    <ID>eTempBCVar</ID>
    <Name>Boundary temperature</Name>
    <Symbol>Tb</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
  </BCVar>
  <Constraint>
    <Type>Solution</Type>
    <Var>eFlux_T</Var>
    <Formula>eTempBCVar</Formula>
  </Constraint>
  <BdryFlux>
    <FluxData>
      <Var>eFlux_T</Var>
      <Formula>- eHeatDiff * (RGradX{eFlux_T} * Norm{X}
                          + RGradY{eFlux_T} * Norm{Y})</Formula>
    </FluxData>
  </BdryFlux>
</BCType>
```

This boundary condition type requires one boundary condition variable, `eTempBCVar`, which will store the value of the temperature imposed on the boundary. The boundary condition is imposed using a constraint on the reconstruction. Finally, the boundary flux is the same as the interior flux shown in Section 5.2.4. The only difference here is that the right-sided values are used for the flux computation. Since a boundary face only has one neighbor control volume, both the left- and right-sided values point to the values from the same control volume. It is a waste of computational resources to compute the average of the two on the boundary.

5.2.8 Complex flux functions

In some situations, the flux (and wavespeeds) computations are more complex than the **`<FluxData>`** constructs could handle. For these cases, it is possible to use the **`<UserCode>`** tag instead of the

<FluxData> tag. The **<UserCode>** tag must be used in conjunction with the **<UserFunction>** tag.

The <UserFunction> tag

This tag allows the user to define a complete C++ function inside the Physics class. No identifier expansion is performed in this section, so the code written must only rely on data passed to it through its parameters. This technique can be used to write flux functions that can compute both the interior and boundary fluxes by using different parameters.

For example, the interior flux for the solid mechanics package can be written as:

$$\vec{F} = -\begin{pmatrix} \frac{E}{1-\nu^2}\left(\epsilon_{xx} + \nu\epsilon_{yy}\right) \\ \frac{E}{2(1-\nu)}\epsilon_{xy} \end{pmatrix}\vec{n}_x - \begin{pmatrix} \frac{E}{2(1-\nu)}\epsilon_{xy} \\ \frac{E}{1-\nu^2}\left(\epsilon_{yy} + \nu\epsilon_{xx}\right) \end{pmatrix}\vec{n}_y$$

A <UserFunction> definition for this flux would look like this:

```
<UserFunction>
  void vGenericFlux(const double ddudx, const double ddudy,
                    const double ddvdx, const double ddvdy,
                    const double dnu, const double dE,
                    const double dnx, const double dny,
                    double adFlux[2]) const {
    // Generic flux for CSM problem
    double adFluxX[2];
    double adFluxY[2];
    double dConst1 = dE / (1.0 - dnu*dnu);
    double dConst2 = dE / (2.0 * (1.0 - dnu));
    adFluxX[0] = dConst1 * (ddudx + dnu*ddvdy);
    adFluxX[1] = 0.5*dConst2 * (ddudy + ddvdx);
    adFluxY[0] = adFluxX[1];
    adFluxY[1] = dConst1 * (ddvdy + dnu*ddudx);

    adFlux[0] = -adFluxX[0]*dnx - adFluxY[0]*dny;
    adFlux[1] = -adFluxX[1]*dnx - adFluxY[1]*dny;
  }
</UserFunction>
```

The function returns the flux in the array of doubles `adFlux`. This function can then be called from within code defined in the **`<UserCode>`** tag.

The `<UserCode>` tag

This tag allows the user to input C++ code within the flux definition function. It is used in place of the **`<FluxData>`** tag. The user can assume that an array of doubles `adFlux` with the proper size is available to the piece of code. The code expects the flux values to be in this variable once the **`<UserCode>`** snippet is complete. Furthermore, known identifiers found in this section of the code will be expanded to access the proper variables. The interior flux definition for the solid mechanics package from the previous example now looks like this:

```
<InteriorFlux>
  <UserCode>
    vGenericFlux(GradX{eFlux_U}, GradY{eFlux_U},
                 GradX{eFlux_V}, GradY{eFlux_V},
                 ePoisson, eElastMod, Norm{X}, Norm{Y},
                 adFlux);
  </UserCode>
</InteriorFlux>
```

5.2.9 Complete file descriptions

The complete file descriptions for all the `Physics` classes used in this research can be found in Appendix B. The C++ code generated by the parser for the heat conduction package is also included, for reference purposes. Note how the XML file for heat conduction is easier to read, and about 35% the size of the corresponding C++ file.

Summary

Section 5.1 described how the solver was modified to allow the source terms from one physical package to affect the flux variables from another physical package. This is particularly helpful for combustion problems. Additionally, Section 5.2 described the XML syntax for the definition of Physics classes. This syntax was created to further simplify the task of developing new `Physics` classes.

Chapter 6

Discussion and future work

This chapter is separated in two main sections. Section 6.1 discusses the accuracy of the solver, and possible ways to further improve it in the future. In particular, improved reconstruction along boundary faces, deformable geometries, periodic boundary conditions, and time-accurate simulations are covered. Section 6.2 discusses several ways the efficiency of ANSLib could be improved, such as implicit and multigrid methods, parallel computing and adaptive mesh refinement.

6.1 Accuracy

The accuracy of the original generic solver has already been demonstrated in [34, 33]. The same level of accuracy was expected, and observed, for the multiphysics solver developed in this research, as it uses the same numerical toolkit.

The ability to use third- or fourth-order reconstruction schemes certainly was useful in achieving these excellent results. A third-order reconstruction scheme uses a quadratic polynomial to approximate the solution over a given control volume. This scheme therefore recovers the quadratic velocity profile of problems like channel flow exactly, requiring few control volumes for a good level of accuracy. A second-order scheme, with its linear approximation over each control volume, would require a greater number of cells to reach the same results.

6.1.1 Reconstruction along boundary faces

The reconstruction scheme must take into account the boundary constraints when computing the reconstruction polynomial for control volumes located next to the boundary faces. These constraints are enforced at the Gauss points on the control volume faces. For boundary constraints, only control volumes with faces adjacent to the boundary are affected. In a cell-centered mesh, this has the effect that the reconstruction for cells with only one vertex on the boundary (the light-shaded cells in Figure 6.1) is not constrained.

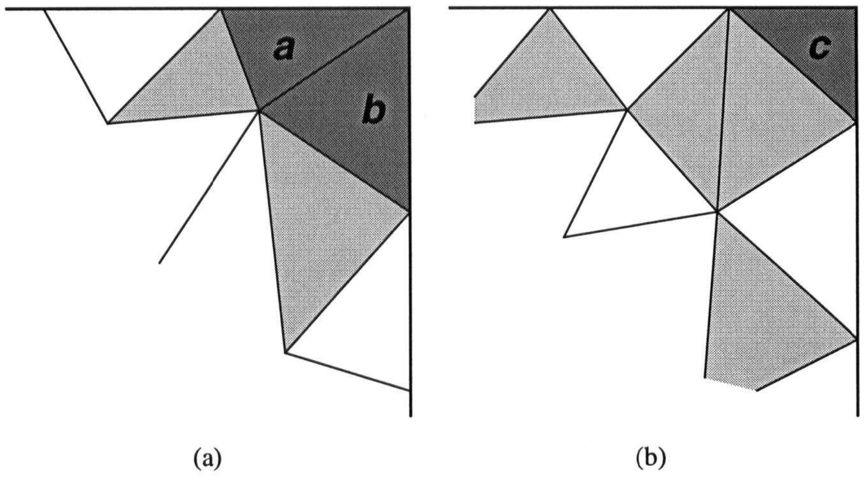


Figure 6.1: Reconstruction along boundary faces.

For corner control volumes (the dark-shaded cells in Figure 6.1), two scenarios are possible. If multiple cells share the same corner, as cells a and b in Figure 6.1a, each cell will lack the constraints from the boundary face it does not touch. For corners with only one cell (cell c), the constraints from both boundary faces will be enforced; the reconstruction will be accurate for that corner.

The rest of the shaded cells will have a reconstruction polynomial that does not account for all boundary constraints. This can lead to some errors in the solution. For example, in a simulation with the Navier-Stokes and the energy equation packages, the outflow boundary condition simply lets the flux at that boundary face exit the domain; no constraint is imposed. Having this boundary condition type with a mesh similar to the one shown in Figure 6.1a would result in cell b left completely unconstrained. This can result in a large discrepancy with the expected results, and is the reason the temperature profile shown for this simulation in Chapter 4 was taken slightly upstream from the edge of the domain.

Fixing this problem requires adding reconstruction constraints at the vertices of control volumes adjacent to a boundary face. A simple way to achieve this would be to create additional Gauss points at the boundary vertices; these Gauss points would ensure that the constraints at the boundary are enforced. By setting the weight of these vertex Gauss points to zero, they would not affect the flux integral of the control volume. The vertex Gauss points could also be used in situations where a boundary condition at a given point, rather than a boundary condition imposed on a face, is needed. This would be particularly useful in solid mechanics problems, were displacements in both directions must be fixed at some point in order for a problem to be well-posed.

6.1.2 Deformable geometries and moving 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. Support for deformable geometries is a feature that is definitely needed for multiphysics simulations, but was not implemented in the present research due to time constraints.

6.1.3 Periodic boundary conditions

ANSLib currently does not support periodic boundary conditions. Using the concept of variable association, it would be straightforward to assign the inflow of a domain to the values at the outflow. However, currently, it is impossible to associate variables from different locations, which a periodic boundary condition would require. The solver fetches the values for associated variables as it is going through dependency trees. The solver goes through the tree at every Gauss point, and the only location stored is that of the Gauss point itself; all variables in the tree are computed at that location.

To allow periodic boundary conditions, a location vector would have be tagged to every variable in the dependency tree. This would cause further complications such as recognizing that the same variable needed at different locations will require a separate node in the dependency tree, and that it will also require a separate storage location, to ensure that variable value from one location does not get overwritten by the same variable from another location. Another way would be to have a special mapping function that would map the location vector to a different location, only for boundary faces that have been tagged as having periodic boundary conditions.

6.1.4 Time-accurate interface coupling

All problems solved so far in this document have been steady-state problems. As mentioned before, ANSLib can also solve problems in a time-accurate fashion by using constant timesteps, and this extends to the multiphysics solver as well. However, there is one potential problem for time-accurate simulations of problems with interface coupling.

For interface coupling, multiple subdomains are coupled together through the use of boundary conditions. One subdomain is solved using boundary conditions determined by the neighbor subdomain, and then that subdomain is solved using boundary condition values from the first subdomain. With this approach, the boundary condition values are set using data from the previous timestep. For steady-state simulations, this is not a problem, but it definitely prevents the resolution of time-accurate simulations.

A solution to this problem has been found, and implementation is currently underway. Results were no yet available at the time of writing, but the approach is presented here. The solution is to accurately predict the value of the boundary conditions at the interface between two subdomains using Taylor series. Since the timestep is constant, it is possible to use finite-difference approximations to the boundary condition values. All that is needed is to keep track of previous boundary condition values. This is accomplished by having the `Region` class keep the `Recon` objects it uses.

Finite-difference approximations

These expressions are all one-sided finite-difference relations. They yield the value of boundary condition value T_b at time t_{n+1}. The number of `Recon` object from previous timesteps needed depend on the order of accuracy of the simulation.

First-order

The first-order approximation simply uses the value from the previous timestep.

$$T_{b_{n+1}} = T_{b_n} \tag{6.1}$$

Second-order

$$T_{b_{n+1}} = 2T_{b_n} - T_{b_{n-1}} \tag{6.2}$$

Third-order

$$T_{b_{n+1}} = 3T_{b_n} - 3T_{b_{n-1}} + T_{b_{n-2}} \tag{6.3}$$

Fourth-order

$$T_{b_{n+1}} = 4T_{b_n} - 6T_{b_{n-1}} + 4T_{b_{n-2}} - T_{b_{n-3}} \tag{6.4}$$

6.2 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 then 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 also relatively small hope that the situation would improve without a complete overhaul of the multiphysics framework.

There are however several other ways that the overall efficiency of the solver can be improved, with a few of them listed below.

6.2.1 Implicit methods

ANSLib uses explicit time-advance methods. These methods were chosen for the initial development of the toolkit because of the ease with which they can be implemented. The emphasis at this development stage was accuracy. Now that the framework has been proven to solve problems accurately, different methods can be used to solve them more efficiently. With explicit methods, convergence for either steady-state problems (using local timestepping) or time-accurate problems (using global timestepping) is slow. A doctoral project already underway looks at implementing generic implicit time-advance methods in ANSLib. Implicit methods result in having to solve very large linear systems. Since solving these large systems directly is too costly computationally, iterative solvers based on Krylov spaces are being investigated. These iterative solvers also have the benefit that they could solve for the steady-state solution of linear problems directly rather than through timestepping methods. This would be a huge efficiency gain for problems such as linear solid mechanics and heat conduction.

6.2.2 Multigrid methods

Multigrid methods use several meshes of different densities of the same domain to damp out different frequencies of the error of the solution. The high-frequency errors are first smoothed on the finest grid, and the problem is transferred to a next-coarser mesh. This process repeats until it reaches the coarsest mesh, at which point a solution correction is transferred back to the next-finer mesh, and the error is smoothed once more, until it reaches the finest grid to complete an iteration.

Even though each iteration now takes longer, the dramatic reduction in the overall number of iterations needed to reach a given convergence level makes this method very attractive. Both multigrid and implicit methods are being investigated in the same project; further efficiency gains could possibly be made by combining the two together.

6.2.3 Parallel computing

Another way to make the solver run faster is to make good use of the advantages parallel computing provides. By solving a problem on n processors, the solution can theoretically be obtained n times faster. The practical efficiency gains are somewhat lower due to the latency caused by the communication between different processors. Proper implementation of parallel techniques helps reduce these delays to a minimum. A graduate student is already working on turning ANSLib into a parallel solver.

For multiphysics problems, `Region` classes already solve problems on their mesh independently from each other. Spreading each `Region` class on a separate processor seems like a natural division. However, since the size of each mesh, the number of unknowns, and the complexity of the flux functions in each `Region` are different, this separation probably will not yield the optimal efficiency. Each `Region` will likely need to be further divided into separate subdomains to ensure the load among processors is properly balanced. Even in this case, the techniques used for interface coupling might be a great solution for the information exchange between different processors.

6.2.4 Adaptive mesh refinement

Adaptive mesh refinement aims to minimize the number of computational nodes needed for a given accuracy of the solution. This is accomplished by refining the mesh in areas where the error in the solution is large, and coarsening the mesh in areas where the error is small. This is known as h-refinement. Since the number of computational nodes is minimized, the time required to perform each iteration of the simulation is much smaller.

The most critical aspect of an adaptive mesh refinement technique is the error estimator. If the error is deemed too large, nodes must be added; too small, and some nodes are removed. For the finite-volume method, literature documenting second-order accurate estimators for specific problems is readily available. However, for third- or fourth-order methods, very little is known; the second-order error estimators most likely cannot be used for high-order methods.

The high-order reconstruction schemes also open the door to using automatic p-refinement techniques: this consists in solving different regions of the domain using different orders of accuracy. Again, error estimators that can determine when it is advantageous to switch to a higher order of accuracy would have to be created. Since the reconstruction code is already working, asking the solver to change the order of reconstruction over different control volumes would be straightforward.

Chapter 7

Summary

Numerical simulation tools allow scientists to solve problems of increasing complexity, leading to increased efficiency of the design and engineering processes. These tools can however be improved, and this research focused on improving two aspects of the numerical simulation process: mesh generation and numerical computation of multiphysics problems.

7.1 Mesh generation

First, a fully-automatic mesh generator capable of generating guaranteed-quality meshes for complex geometries with curved boundary has been written. A framework allowing two-dimensional curved boundary segments was combined with a guaranteed-quality Delaunay refinement algorithm; details were presented in Chapter 2. The use of curved boundaries demanded that boundary edges be split differently to ensure regions with higher curvature were discretized with a greater number of edges. The midpoints are now computed using the total variation of the tangent angle. The initial discretization of the domain also needed some modifications; the new strategy first discretizes the curved boundaries with as few segments as possible, avoiding the creation of artificial small features in the mesh. Some recovery problems due to the very coarse nature of the initial discretization of curved boundaries were encountered, but solutions to these were incorporated into the new initial discretization strategy. Several boundary patch types were implemented and tested. Furthermore, it is possible to add new boundary types to the generic boundary easily. Examples demonstrating the successful use of curved boundary patches were also presented. These two-dimensional meshes all showed excellent quality, and had a minimum angle exceeding 30°.

Finally, a mathematical proof that guarantees the quality of the final mesh has been extended to include meshes from a domain with curved boundaries. This research was an important step in allowing the generation of guaranteed-quality meshes directly from CAD models, which would considerably cut down on the amount of time necessary for domain preparation.

7.2 Numerical simulation of generic multiphysics problems

The second part of this research addressed the problems domain experts face when trying to perform numerical simulations related to their field of expertise. A generic numerical toolkit was modified, allowing generic multiphysics simulations to be performed. With this new solver, scientists can simply describe each of the physics of their problems — something they understand very well — and obtain solutions to new and complex physical problems within days. There is no need to write a specialized solver tailored to their specific problems, or to wait for a commercial package to be available.

The multiphysics solver is based on a generic numerical toolkit which was described in Chapter 3. The toolkit uses the finite-volume method, which allow the decoupling of the numerical and physical aspects of a simulation easily. By decoupling the two, all physical problems look the same to the numerical toolkit. The physics of a problem are described in a `Physics` class. To demonstrate the effectiveness of this generic approach to numerical simulations, results from three different physical problems solved using the same mesh and solver were presented.

Multiphysics problems presented a number of challenges, as described in Chapter 4. For one, the solver must interact with multiple `Physics` classes at the same time. A layer of code was added to manage these multiple physical packages: the `Region` class. Furthermore, in some cases, multiphysics problems are defined over several subdomains; the `Domain` class was created to handle this. However, the most crucial modification made to the numerical toolkit was allowing the exchange of information between multiple `Physics` classes, whether on the same subdomain or not. This exchange can only take place if the solver knows what information the different `Physics` objects can exchange with each other; the `PhysVar` class ensures this information exchange happens. The coupling between the different physical packages is done using variable association. The association is accomplished by having the solver link a required variable to a provided variable. Variable association can however lead to some dependency problems: dependency trees were implemented to avoid these problems. Results for several multiphysics problems where two or more physical packages were coupled together were presented. The results matched analytical solutions well, when such solutions were available.

In Chapter 5, a way for the source term from one physical package to affect the flux variables from another physical package was described. This is particularly useful for combustion problems. In addition, a simple `Physics` class definition syntax based on XML was created. This file uses human-readable keywords and is automatically converted to C++ by a parsing program specifically written for that purpose. This syntax will make it even easier to write new `Physics` classes for the multiphysics solver. Finally, in Chapter 6, several ways the solver could be further improved were discussed, both in terms of efficiency and accuracy.

7.3 Conclusion

This research helped make the numerical simulation process easier and more accessible to scientists and engineers by improving tools used in both domain preparation and numerical computation. The automatic generation of guaranteed-quality meshes for complex geometries is a huge gain in efficiency for the users of numerical tools. Furthermore, the ability to quickly and easily define and solve new multiphysics problems will surely lead to numerical solvers being used in a wider variety of fields by a greater number of scientists; this can only benefit the whole scientific community.

Bibliography

[1] FEMLAB: An Introductory Course. Available from the FEMLAB website, http://www.femlab.com, 2002.

[2] E. Arge, A. M. Bruaset, P. B. Calvin, J. F. Kanney, H. P. Langtangen, and C. T. Miller. On the numerical efficiency of C++ in scientific computing. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 93–119. Birkhäuser, 1997.

[3] I. Babuska and A. Aziz. On the angle condition in the finite element method. 13:214–226, 1976.

[4] C. Bailey, G. Taylor, S. M. Bounds, G. Moran, and M. Cross. PHYSICA: A multiphysics computational framework and its application to casting simulations. In *Computational Fluid Dynamics in Mineral & Metal Processing and Power Generation*, pages 419–425. CSIRO Division of Minerals, 1997.

[5] T. J. Barth. Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations. In *Unstructured Grid Methods for Advection-Dominated Flows*, pages 6–1–6–61. AGARD, 1992.

[6] T. J. Barth and P. O. Frederickson. Higher order solution of the Euler equations on unstructured grids using quadratic reconstruction. AIAA paper 90-0013, Jan. 1990.

[7] C. Boivin and 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*, pages 49–54, June 2002.

[8] C. Boivin and 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*, pages 55–60, June 2002.

[9] C. Boivin and C. F. Ollivier-Gooch. Guaranteed-quality triangular mesh generation for domains with curved boundaries. *International Journal for Numerical Methods in Engineering*, 55:1185–1213, 2002.

[10] A. P. Boresi, R. J. Schmidt, and O. Sidebottom. *Advanced Mechanics of Materials.* John Wiley and Sons, 5th edition, 1993.

[11] A. M. Bruaset, E. J. Holm, and H. P. Langtangen. Increasing the efficiency and reliability of software development for systems of PDEs. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 247–268. Birkhäuser, 1997.

[12] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations: Diffpack. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 63–92. Birkhäuser, 1997.

[13] L. P. Chew. Guaranteed-quality triangular meshes. Technical Report TR-89-983, Dept. of Computer Science, Cornell University, 1989.

[14] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 274–280. Association for Computing Machinery, May 1993.

[15] S. Dey, R. M. O'Bara, and M. S. Shepard. Curvilinear mesh generation in 3D. In *Proceedings of the Eighth International Meshing Roundtable*, pages 407–417, Oct. 1999.

[16] Y. Dubois-Pèlerin and P. Pegon. Improving modularity in object-oriented finite element programming. *Communications in Numerical Methods in Engineering*, 13:193–198, 1997.

[17] H. Edelsbrunner and D. Guoy. Sink-insertion for mesh improvement. In *Proceedings of the 17th ACM Symposium on Computational Geometry*, pages 115–123, June 2001.

[18] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements: II. A symbolic environment for automatic programming. *Computational Methods in Applied Mechanics and Engineering*, 132:277–304, 1996.

[19] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements: III. Theory and application of automatic programming. *Computational Methods in Applied Mechanics and Engineering*, 154:41–68, 1998.

[20] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements: IV. Symbolic derivations and automatic programming of nonlinear formulations. *Computational Methods in Applied Mechanics and Engineering*, 190:2729–2751, 2001.

[21] C. A. Felippa, K. Park, and C. Farhat. Partitioned analysis of coupled mechanical systems. *Computer Methods in Applied Mechanics and Engineering*, 190:3247–3270, 2001.

[22] J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer-Verlag, 2nd edition, 1999.

[23] P. Fleischmann and S. Selberherr. Three-dimensional Delaunay mesh generation using a modified advancing front approach. In *Proceedings of the Sixth International Meshing Roundtable*, pages 267–278, October 1997.

[24] L. A. Freitag and C. F. Ollivier-Gooch. A cost/benefit analysis of simplicial mesh improvement techniques as measured by solution efficiency. *International Journal for Computational Geometry*, Aug. 2000.

[25] I. Fried. Condition of finite element matrices generated from nonuniform meshes. *AIAA Journal*, 10:219–221, 1972.

[26] H. P. Langtangen. *Computational Partial Differential Equations — Numerical Methods and Diffpack Programming*. Lecture Notes in Computational Science and Engineering. Springer-Verlag, 1999.

[27] P. Laug, H. Borouchaki, and P.-L. George. Maillage de courbes gouverné par une carte de métriques. Technical Report RR-2818, INRIA, 1996.

[28] H. Lomax, T. Pulliam, and D. Zingg. *Fundamentals of Computational Fluid Dynamics*. Springer, 2001.

[29] D. L. Marcum and N. P. Weatherill. Unstructured grid generation using iterative point insertion and local reconstruction. *AIAA Journal*, 33(9):1619–1625, 1995.

[30] D. Mavriplis and S. Pirzadeh. Large-scale parallel unstructured mesh computations for 3D high-lift analysis. *Journal of Aircraft*, 36(6), 1999.

[31] F. Mavriplis. CFD in aerospace in the new millenium. *Canadian Aeronautics and Space Journal*, 46(4):167–176, Dec. 2000.

[32] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. In *Proceedings of the ACM Computational Geometry Conference*, pages 212–221, 1992. Also appeared as Cornell C.S. TR 92-1267.

[33] C. F. Ollivier-Gooch. ANSLib: A scientific computing toolkit supporting rapid numerical solution of practically any PDE. In *Proceedings of the Eighth Annual Conference of the Computational Fluid Dynamics Society of Canada*, pages 21–28, June 2000.

[34] 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*, 192:1147–1175, 2003.

[35] C. F. Ollivier-Gooch and C. Boivin. Guaranteed-quality simplicial mesh generation with cell size and grading control. *Engineering with Computers*, 17:269–286, 2001.

[36] C. F. Ollivier-Gooch and M. Van Altena. A high-order accurate unstructured mesh finite-volume scheme for the advection-diffusion equation. *Journal of Computational Physics*, 181(2):729–752, 2002.

[37] M.-C. Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *International Journal for Numerical Methods in Engineering*, 40:3313–3324, 1997.

[38] J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18:548–585, 1995.

[39] A. Sheffer and A. Ungor. Efficient adaptive meshing of parametric models. In *Proceedings of the 6th ACM Symposium on Solid Modeling and Applications*, pages 59–70, June 2001.

[40] J. R. Shewchuk. *Delaunay Refinement Mesh Generation.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.

[41] M. Van Altena. High-order finite-volume discretisations for solving a modified advection-diffusion problem on unstructured triangular meshes. Master's thesis, Dept. of Mechanical Engineering, University of British Columbia, Oct. 1999.

[42] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Computer Journal*, 24(2):167–172, 1981.

[43] F. M. White. *Viscous Fluid Flow.* McGraw-Hill, 2nd edition, 1991.

[44] T. Zimmermann and D. Eyheramendy. Object-oriented finite elements: I. Principles of symbolic derivations and automatic programming. *Computational Methods in Applied Mechanics and Engineering*, 132:259–276, 1996.

Appendix A

Proof of Mesh Quality in Two Dimensions

A.1 Angle bounds

The proof presented in this appendix follows the same approach used in previous research [35]. Boundary edges are assumed to be protected by diametral lenses and that input angles are greater than 60°, to prevent adjacent edges from encroaching on each other. It is possible show the same bounds for curved boundaries that were previously demonstrated for straight boundaries.

The following lemma will now be proven, which will, among other results, establish an angle bound for finite cell size and hence for algorithm termination. This lemma is deliberately stated in language as similar as possible to Ruppert's Lemma 2, even to exact quotation of much of the phrasing, although details of the proof and the derived constants differ.

Lemma 1. (After Ruppert [38]) For fixed constants C_L, C_T and C_S, determined below, the following statements hold:

1. At initialization, for each input vertex p, the distance to its nearest neighbor vertex is at least $lfs_c(p) \equiv R \cdot LS(p)$.
2. When a point p is chosen as the circumcenter of an overly-large triangle, the distance to the nearest vertex is at least $LS(p)/C_L$. (p may be added to the triangulation, or may be rejected because it encroaches upon some segment.)
3. When a point p is chosen as the circumcenter of a skinny triangle, the distance to the nearest vertex is at least $LS(p)/C_T$. (Again, p may be added to the triangulation, or may be rejected because it encroaches upon some segment.)

4. When a vertex p is added as the midpoint of a split segment, the distance to its nearest neighbor vertex is at least $LS(p)/C_S$.

Proof.

Case 1. Statement 1 of Lemma 1 is true by the definition of the length scale LS from the local feature size lfs_c, provided only that the constant R in Equation 2.1 is ≥ 1.

Having established the truth of Lemma 1 for the initial mesh, it is now possible to proceed by induction to prove that it must be true for all meshes generated by the algorithm. As such, it is assumed that Lemma 1 holds for all points in the mesh and determine the bounds on C_S, C_T, C_L, and G that are required for Lemma 1 to hold for newly inserted points.

Case 2. The case of insertion to split a large triangle, as shown in Figure A.1, is first considered. By definition, the circumradius of $\triangle abc$ is larger than $\frac{\sqrt{2}}{2}$ times the average of the length scales at its vertices. The distance from p to the nearest point is the circumradius of $\triangle abc$, or

$$r \geq \frac{\sqrt{2}}{2}\frac{LS(a)+LS(b)+LS(c)}{3} \tag{A.1}$$

The length scale at p can be bound in terms of this same average using Equation 2.1:

$$\begin{aligned}
LS(p) &\leq LS(a)+\frac{r}{G} \\
LS(p) &\leq LS(b)+\frac{r}{G} \\
LS(p) &\leq LS(c)+\frac{r}{G} \\
LS(p) &\leq \frac{LS(a)+LS(b)+LS(c)}{3}+\frac{r}{G} \qquad \text{(A.2)} \\
\frac{LS(a)+LS(b)+LS(c)}{3} &\geq LS(p)-\frac{r}{G}
\end{aligned}$$

Combining inequalities A.1 and A.2 results in:

$$\begin{aligned}
r &\geq \frac{\sqrt{2}LS(p)}{2}-\frac{r\sqrt{2}}{2G} \\
r &\geq \frac{LS(p)}{\sqrt{2}+\frac{1}{G}} \qquad \text{(A.3)}
\end{aligned}$$

This inequality places a lower bound on point spacing for points inserted to split large triangles, and confirms Statement 2 of Lemma 1, for any

$$C_L \geq \sqrt{2} + \frac{1}{G} \tag{A.4}$$

The lower bound on C_L becomes smaller for large values of G, corresponding to slow change in cell size.

Case 3. The case of insertion to split a badly shaped triangle, as illustrated in Figure A.2, is considered next. Without loss of generality, vertices can be labelled so that a and b are connected by the shortest edge (of length l_{min}), and a was inserted in the mesh after b (or both were input vertices). The radius of the vertex-free circle around a is r'. Four subcases for relating r' to $LS\,(p)$ arise, depending on why a was inserted in the mesh.

Subcase 3a. *a was an input vertex.* Then so was b, Statement 1 of Lemma 1 applies, and the distance $l_{\text{min}} \geq R \cdot LS\,(a)$.

Subcase 3b. *a was inserted to split a large triangle.* The circumradius r' of that triangle is no larger than l_{min}, because vertex b was not inside the circumcircle. Then Statement 2 of Lemma 1 applies, and $l_{\text{min}} \geq r' \geq LS\,(a)\,/C_L$.

Subcase 3c. *a was inserted to split a badly shaped triangle.* By a similar argument and using Statement 3 of Lemma 1, and $l_{\text{min}} \geq r' \geq LS\,(a)\,/C_T$.

Subcase 3d. *a was inserted to split an encroached boundary edge.* It is known that b does not lie inside the diametral lens of the edge a split, because otherwise b would encroach on that edge. Statement 4 of Lemma 1 applies, and $l_{\text{min}} \geq r' \geq LS\,(a)\,/C_S$.

If $C_S \geq C_T = C_L \geq 1$ is satisfied (which will be shown is possible), then the inequality $l_{\text{min}} \geq LS\,(a)\,/C_S$ (subcase 3d) causes the most difficulty in satisfying Statement 3 of Lemma 1 by making the length scale at p larger than that for any other subcase.

The radius r of the circumcircle of $\triangle abc$ is related to its smallest angle. The angle $\angle apb = 2\theta$ by geometry, and trigonometry gives $l_{\text{min}} = 2r\sin\theta$. The definition of length scale gives

$$\begin{aligned} LS\,(p) \leq LS\,(a) + \frac{r}{G} &\leq C_S l_{\text{min}} + \frac{r}{G} \\ &= 2rC_S \sin\theta + \frac{r}{G} \end{aligned} \tag{A.5}$$

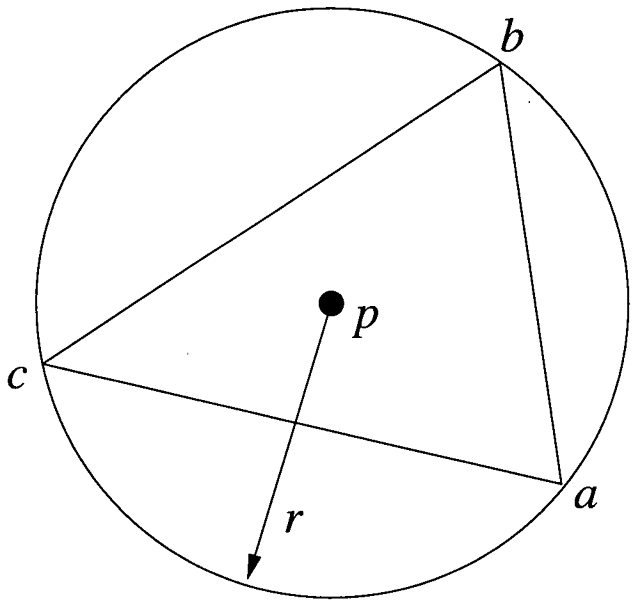


Figure A.1: Lemma 1, Statement 2: p added as circumcenter of large triangle T.

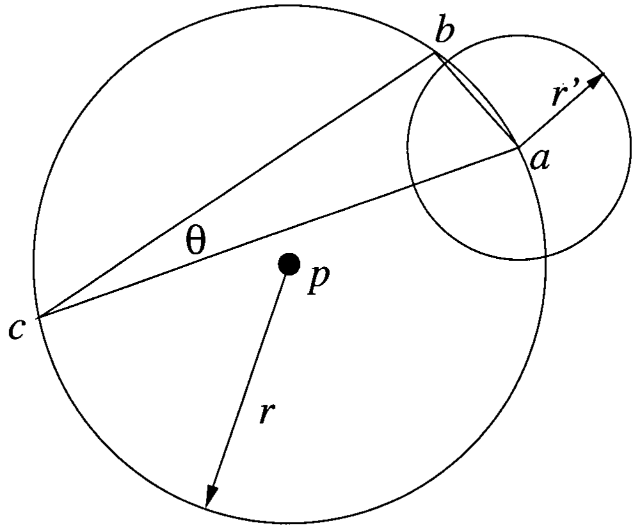


Figure A.2: Lemma 1, Statement 3: p added as circumcenter of badly shaped triangle T.

This triangle is being split because θ is less than the required angle bound α. Inequality A.5 can be strengthened by replacing θ with α, and obtain

$$r \geq \frac{LS\left(p\right)}{\frac{1}{G} + 2C_S \sin\alpha}$$

Lemma 1 states that, for this case, $r \geq LS\left(p\right)/C_T$, so it is required that

$$C_T \geq \frac{1}{G} + 2C_S \sin\alpha \tag{A.6}$$

Case 4. The case in which vertex p is added to the mesh to split a segment s, because some vertex or triangle circumcenter lies inside the diametral lens of s, is covered next. Vertex p is inserted on the patch between b and c, not necessarily at the midpoint of edge bc.This case is illustrated in Figure A.3. There are four subcases.

Subcase 4a. *a lies on a segment t,* which can not share a vertex with s, because it was assumed that input edges are separated by 60°. Therefore, p and a lie on non-adjacent segments, and the length scale at p is $LS\left(p\right) \leq \frac{1}{R}\left|\vec{a}-\vec{p}\right|$. To satisfy Lemma 1 in this subcase, it is therefore required that $C_S \geq \frac{1}{R}$. Because $R \geq 1$, this inequality is always satisfied for $C_S \geq 1$.

Subcase 4b. *a is a point at the circumcenter of a large triangle T.* a has of course been rejected for insertion since it is located inside the diametral lens. The definition of the length scale then gives:

$$LS\left(p\right) \leq LS\left(a\right) + \tfrac{1}{G}\left|\vec{a}-\vec{p}\right|$$

The circumradius r' of T is smaller than the shorter of $\left|\vec{a}-\vec{b}\right|$ and $\left|\vec{a}-\vec{c}\right|$, because T's circumcircle must be point-free. The largest value of r' is obtained when a is at the apex of the diametral lens, so $r' \leq \frac{2}{\sqrt{3}}d$. Also, it is known from this lemma that $LS\left(a\right)/C_L \leq r'$. Furthermore, the largest value of $\left|\vec{a}-\vec{p}\right|$ places a and p at opposite ends of edge $\overline{bc}$, so $\left|\vec{a}-\vec{p}\right| \leq 2d$. The length scale at p now becomes:

$$\begin{aligned} LS\left(p\right) &\leq r'C_L + \tfrac{2d}{G} \\ &\leq \tfrac{2d}{\sqrt{3}}C_L + \tfrac{2d}{G} \\ &\leq d\left(\tfrac{2}{G} + \tfrac{2}{\sqrt{3}}C_L\right) \end{aligned}$$

This inequality satisfies Lemma 1 provided that

$$C_S \geq \frac{2}{G} + \frac{2}{\sqrt{3}} C_L \tag{A.7}$$

Subcase 4c. *a is a point at the circumcenter of a skinny triangle.* The same reasoning can be applied as in the previous subcase, with the result that

$$C_S \geq \frac{2}{G} + \frac{2}{\sqrt{3}} C_T \tag{A.8}$$

Subcase 4d. *The radius of curvature at p is smaller than the local feature size.* In this case, the radius of curvature will define the length scale, and it does not matter whether a comes from a large or a skinny triangle. The definition of length scale will yield:

$$LS\,(p) \quad \leq \tfrac{\rho(p)}{R}$$

This length scale is valid whenever it is smaller than the length scale found in cases $4a$, $4b$, or $4c$. Using the results from case $4b$, for example, this is equivalent to saying that:

$$\frac{\rho}{R} \leq d\left(\frac{2}{G} + \frac{2}{\sqrt{3}} C_L\right)$$

It will be shown that this inequality only holds for a few splits at the boundary, and therefore does not affect the provable angle bound nor the termination of the algorithm. Using $\rho = \frac{1}{|\kappa|}$ and re-arranging terms in the previous inequality results in:

$$\frac{1}{d} \leq \left(\frac{2R}{G} + \frac{2R}{\sqrt{3}} C_L\right) |\kappa|$$

Integrating both sides with respect to arclength over $\overline{bc}$:

$$\begin{aligned} \tfrac{1}{d}\int_b^c ds \quad &\leq \quad \left(\tfrac{2R}{G} + \tfrac{2R}{\sqrt{3}} C_L\right) \int_b^c |\kappa(s)|\; ds \\ \tfrac{s_{bc}}{d} \quad &\leq \quad \left(\tfrac{2R}{G} + \tfrac{2R}{\sqrt{3}} C_L\right) TV(\theta)\,|_b^c \end{aligned} \tag{A.9}$$

The left-hand side integral is the arclength of the patch between b and c, s_{bc}, and the right-hand side integral is $TV(\theta)$ between b and c. As the boundary patch gets split, s_{bc}, d, and $TV(\theta)\,|_b^c$ all decrease. The ratio on the left-hand side is bounded as the curve looks more and more linear, i.e. $\frac{s_{bc}}{d} \geq 2$. The right-hand side term decreases by a factor of two after every split. After enough splits, the inequality invariably no longer holds, and the length scale is not defined by the radius of curvature at point p anymore.

To establish the truth of Lemma 1, one must find values of C_S, C_T, and C_L that simultaneously satisfy Inequalities A.4, A.6, A.7, and A.8. Tight bounds are established on each constant by requiring equality in each case. Using only A.6, A.7, and A.8 results in:

$$\begin{aligned} C_S &= \frac{1}{G}\frac{2\sqrt{3}+2}{\sqrt{3}-4\sin\alpha} \\ C_T = C_L &= \frac{1}{G}\frac{\sqrt{3}+4\sqrt{3}\sin\alpha}{\sqrt{3}-4\sin\alpha} \end{aligned}$$

These values are bounded for any angle bound $\alpha \leq \arcsin\left(\frac{\sqrt{3}}{4}\right)$, just as for Shewchuk's modification to Ruppert's scheme. The constants are nearly identical to those found previously for straight boundaries, with only a slight difference in the numerators.

Equation A.4 can be treated as establishing a lower bound on the grading rate $G \geq \frac{1}{C_L-\sqrt{2}}$. So long as G is finite, the mesh will be non-uniform. Relating G to the angle bound α:

$$G \leq \frac{2\sqrt{2}\sin\alpha\left(1+\sqrt{3}\right)}{\sqrt{3}-4\sin\alpha}$$

The minimum theoretical grading rate G remains finite up to the previously established angle bound. If the higher bound for G is used, the following constants are found:

$$\begin{aligned} C_S &= \frac{1}{\sqrt{2}\sin\alpha} \\ C_T = C_L &= \frac{\sqrt{3}\left(1+4\sin\alpha\right)}{2\sqrt{2}\sin\alpha\left(1+\sqrt{3}\right)} \end{aligned}$$

In summary, it has been established that, for $R \geq 1$, meshes can be generated with the same angle bounds as Shewchuk's modification to Ruppert's scheme. In the process, bounds have been placed on the grading rate G and on the length of the shortest edge in the mesh relative to the local length scale (the constants C_S, C_T, and C_L give this information).

A.2 Termination and Size Optimality

The quality lemmas can be used to prove the following theorem about finite mesh size and mesh size optimality.

Theorem 1. Given a vertex p in the output triangular mesh, its nearest neighbor vertex q is at a distance at least $LS(p) / (C_S + 1/G)$. This implies mesh size optimality.

Proof.

Lemma 1 handles the case where p is inserted after q. If q is inserted last, then the lemma is applied to q:

$$|\vec{q} - \vec{p}| \geq \frac{LS(q)}{C_S}$$

But $LS(p) \geq LS(q) + \frac{|\vec{q}-\vec{p}|}{G}$, so

$$|\vec{q} - \vec{p}| \geq \frac{LS(p) - \frac{|\vec{q}-\vec{p}|}{G}}{C_S}$$

and the theorem follows, with only minor algebra.

Because the shortest edge in the mesh must be longer than $\frac{LS_{\text{min}}}{C_S+\frac{1}{G}}$, each cell has finite size and only a finite number of them will be required.

Futhermore, because the shortest possible edge is within a constant factor of the length scale locally, the smallest possible triangle is within the square of that same factor of the size of a triangle whose edges all match the length scale. This implies that the size of the mesh must be within a constant factor of the size of the smallest possible mesh whose cells meet the quality bound and whose edges have length within a constant factor of the length scale locally.

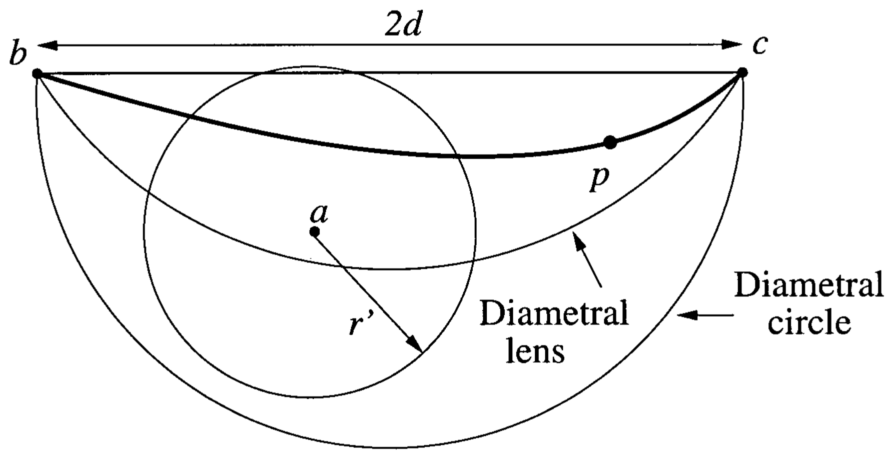


Figure A.3: Lemma 1, Statement 4: p added to split an encroached boundary edge.

Appendix B

Physics classes definitions

B.1 Heat conduction XML file

```
<?xml version="1.0" ?>
<Physics>
  <ClassID>Heat2D</ClassID>
  <ClassDesc>2D Heat conduction package</ClassDesc>
    <FluxVar>
      <ID>eFlux_T</ID>
      <Name>Temperature</Name>
      <Symbol>T</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    </FluxVar>
    <RequiredVar>
      <ID>eConduct</ID>
      <Name>Conductivity</Name>
      <Symbol>k</Symbol>
      <Units> <L>1</L> <M>1</M> <T>-3</T> <K>-1</K> </Units>
    </RequiredVar>
    <RequiredVar>
      <ID>eDens</ID>
      <Name>Density</Name>
      <Symbol>rho</Symbol>
      <Units> <L>-3</L> <M>1</M> <T>0</T> <K>0</K> </Units>
    </RequiredVar>
    <RequiredVar>
      <ID>eSpecHeat</ID>
      <Name>Specific Heat</Name>
    <Symbol>cp</Symbol>
    <Units> <L>2</L> <M>0</M> <T>-2</T> <K>-1</K> </Units>
  </RequiredVar>
  <ComputedVar>
    <ID>eHeatDiff</ID>
    <Name>Heat Diffusivity</Name>
    <Symbol>a</Symbol>
    <Units> <L>2</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>eConduct / (eDens * eSpecHeat)</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eHeatFlux</ID>
    <Name>Heat Flux</Name>
    <Symbol>q</Symbol>
    <Units> <L>0</L> <M>1</M> <T>-3</T> <K>0</K> </Units>
    <Formula>eConduct * ( GradX{eFlux_T}*Norm{X} +
                          GradY{eFlux_T}*Norm{Y} )</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eAveTemp</ID>
    <Name>Temperature</Name>
    <Symbol>T</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    <Formula>0.5 * (LSoln{eFlux_T} + RSoln{eFlux_T})</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryDTDX</ID>
    <Name>Bdry dT/dx</Name>
    <Symbol>Tx</Symbol>
    <Units> <L>-1</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    <Formula>RGradX{eFlux_T}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryDTDY</ID>
    <Name>Bdry dT/dy</Name>
    <Symbol>Ty</Symbol>
    <Units> <L>-1</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    <Formula>RGradY{eFlux_T}</Formula>
  </ComputedVar>
<InteriorFlux>
  <FluxData>
```

```
        <Var>eFlux_T</Var>
        <Formula>- eHeatDiff * (GradX{eFlux_T}*Norm{X} +
                                  GradY{eFlux_T}*Norm{Y})</Formula>
      </FluxData>
    </InteriorFlux>
    <WaveSpeeds>
      <Minimum>-eInvDist * 0.3</Minimum>
      <Maximum> eInvDist * 0.3</Maximum>
    </WaveSpeeds>
    <SourceTerm>
    </SourceTerm>
    <BCType>
      <Desc>This BC imposes an insulated wall</Desc>
      <ID>eBC_Ins</ID>
      <BdryFlux>
        <FluxData>
          <Var>eFlux_T</Var>
          <Formula>0</Formula>
        </FluxData>
      </BdryFlux>
      <Constraint>
        <Type>NGradient</Type>
        <Var>eFlux_T</Var>
        <Formula>0</Formula>
      </Constraint>
    </BCType>
    <BCType>
      <Desc>This BC imposes a temperature at the wall</Desc>
      <ID>eBC_Temp</ID>
      <BCVar>
        <ID>eTempBCVar</ID>
        <Name>Boundary temperature</Name>
        <Symbol>Tb</Symbol>
        <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
      </BCVar>
      <Constraint>
        <Type>Solution</Type>
        <Var>eFlux_T</Var>
        <Formula>eTempBCVar</Formula>
      </Constraint>
      <BdryFlux>
      <FluxData>
        <Var>eFlux_T</Var>
        <Formula>- eHeatDiff * (RGradX{eFlux_T} * Norm{X} +
                                RGradY{eFlux_T} * Norm{Y})</Formula>
      </FluxData>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes a heat flux at the wall</Desc>
    <ID>eBC_Flux</ID>
    <BCVar>
      <ID>eHeatBCVar</ID>
      <Name>Boundary heat flux</Name>
      <Symbol>qb</Symbol>
      <Units> <L>0</L> <M>1</M> <T>-3</T> <K>0</K> </Units>
    </BCVar>
    <BdryFlux>
      <FluxData>
        <Var>eFlux_T</Var>
        <Formula>- eHeatBCVar / (eDens * eSpecHeat)</Formula>
      </FluxData>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes an internal (block) boundary</Desc>
    <ID>eBC_IntBdry</ID>
    <BCVar>
      <ID>eIntBdryDTDX</ID>
      <Name>Boundary dT/dx</Name>
      <Symbol>Txb</Symbol>
      <Units> <L>-1</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    </BCVar>
    <BCVar>
      <ID>eIntBdryDTDY</ID>
      <Name>Boundary dT/dy</Name>
      <Symbol>Tyb</Symbol>
      <Units> <L>-1</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    </BCVar>
    <BdryFlux>
      <FluxData>
        <Var>eFlux_T</Var>
        <Formula>- 0.5 * eHeatDiff *
                   ( (RGradX{eFlux_T} + eIntBdryDTDX)*Norm{X}
                   + (RGradY{eFlux_T} + eIntBdryDTDY)*Norm{Y} )</Formula>
      </FluxData>
    </BdryFlux>
  </BCType>
</Physics>
```

B.2 Heat conduction C++ file

As an example, here is the C++ file generated by the parser from the XML file above. Notice how reconstruction and various computed variables were created automatically.

```
#include <math.h>

#define VAR_U_NORM -1
#define VAR_U_TANG -2
#define VAR_U_CROSS -3

#ifndef VAR_VEC_X
#define VAR_VEC_X -100  /* Deliberately illegal value */
#endif
#ifndef VAR_VEC_Y
#define VAR_VEC_Y -100  /* Deliberately illegal value */
#endif
#ifndef VAR_VEC_Z
#define VAR_VEC_Z -100  /* Deliberately illegal value */
#endif

#include "ANS_NewPhysics.h"
#include "ANS_PhysVar.h"
#include "ANS_Taylor.h"

class Heat2D_Physics : public NewPhysics {

public:

  enum eVars {
         eFlux_T, eConduct, eDens, eSpecHeat, eRGradX_eFlux_T,
         eLGradX_eFlux_T, eAGradX_eFlux_T, eNormX, eRGradY_eFlux_T,
         eLGradY_eFlux_T, eAGradY_eFlux_T, eNormY, eLSoln_eFlux_T,
        eRSoln_eFlux_T, eHeatDiff, eHeatFlux, eAveTemp, eLastVar
      };

enum eBCTypes {
        eBC_Ins, eBC_Temp, eBC_Flux, eLastBCType
      };

enum e_eBC_Temp_Vars {
        eTempBCVar, eBC_Temp_LastVar
      };

enum e_eBC_Flux_Vars {
        eHeatBCVar, eBC_Flux_LastVar
      };

Heat2D_Physics() : NewPhysics() {
  // Class description
  sWordDescription = "2D Heat conduction package";
  // variable number
  vSetNumberOfVariables(eLastVar);

  // Now define variables

  // eFlux_T
  vAddPhysVariable(eFlux_T, eStateProvided, eTypeFlux, eWhereField,
                   0, 0, 0, 1, "T", "Temperature");
  // eConduct
  vAddPhysVariable(eConduct, eStateRequired, eTypeVariable, eWhereField,
                   1, 1, -3, -1, "k", "Conductivity");
  // eDens
  vAddPhysVariable(eDens, eStateRequired, eTypeVariable, eWhereField,
                   -3, 1, 0, 0, "rho", "Density");
  // eSpecHeat
  vAddPhysVariable(eSpecHeat, eStateRequired, eTypeVariable, eWhereField,
                   2, 0, -2, -1, "cp", "Specific Heat");
  // eAGradX_eFlux_T
  vAddPhysVariable(eAGradX_eFlux_T, eStateProvided, eTypeComputed,
                   eWhereField, 0, 0, 0, 0, "",
                   "Average eTypeGrad eFlux_T");
  // eAGradY_eFlux_T
  vAddPhysVariable(eAGradY_eFlux_T, eStateProvided, eTypeComputed,
                    eWhereField, 0, 0, 0, 0, "",
                    "Average eTypeGrad eFlux_T");
// eHeatDiff
vAddPhysVariable(eHeatDiff, eStateProvided, eTypeComputed, eWhereField,
                    2, 0, -1, 0, "a", "Heat Diffusivity");
// eHeatFlux
vAddPhysVariable(eHeatFlux, eStateProvided, eTypeComputed, eWhereField,
                    0, 1, -3, 0, "q", "Heat Flux");
// eAveTemp
vAddPhysVariable(eAveTemp, eStateProvided, eTypeComputed, eWhereField,
                    0, 0, 0, 1, "T", "Temperature");
// eRGradX_eFlux_T
vAddReconVariable(eRGradX_eFlux_T, eTypeGrad, eWhereRight, XDIR, eFlux_T);
// eLGradX_eFlux_T
vAddReconVariable(eLGradX_eFlux_T, eTypeGrad, eWhereLeft, XDIR, eFlux_T);
// eNormX
vAddReconVariable(eNormX, eTypeNorm, eWhereField, XDIR, -1);
// eRGradY_eFlux_T
vAddReconVariable(eRGradY_eFlux_T, eTypeGrad, eWhereRight, YDIR, eFlux_T);
// eLGradY_eFlux_T
vAddReconVariable(eLGradY_eFlux_T, eTypeGrad, eWhereLeft, YDIR, eFlux_T);
// eNormY
vAddReconVariable(eNormY, eTypeNorm, eWhereField, YDIR, -1);
// eLSoln_eFlux_T
vAddReconVariable(eLSoln_eFlux_T, eTypeSoln, eWhereLeft, -1, eFlux_T);
// eRSoln_eFlux_T
vAddReconVariable(eRSoln_eFlux_T, eTypeSoln, eWhereRight, -1, eFlux_T);

// Now define BCs

// number of BCs
vSetNumberOfBCTypes(eLastBCType);

// BC Types

vAddBCType(eBC_Ins,  "This BC imposes an insulated wall");
vAddBCType(eBC_Temp, "This BC imposes a temperature at the wall");
vAddBCType(eBC_Flux, "This BC imposes a heat flux at the wall");

// Variables for eBC_Temp
vSetNumberOfBCVariables(eBC_Temp, 1);
```

```
    vAddBCVariable(eTempBCVar, eBC_Temp, 0, 0, 0, 1, "Tb", "Boundary temperature");

    // Variables for eBC_Flux
    vSetNumberOfBCVariables(eBC_Flux, 1);
    vAddBCVariable(eHeatBCVar, eBC_Flux, 0, 1, -3, 0, "qb", "Boundary heat flux");

  //
  };

  // Destructor
  virtual ~Heat2D_Physics() {
    // Typically no need to have anything here, but it's available in any case

  };

  // Interior Flux dependencies
  virtual void vGetInteriorFluxDependencies(std::vector<bool> * vecAllVars) {
    // Set the appropriate flags to true

    (*vecAllVars)[eAGradX_eFlux_T] = true;
    (*vecAllVars)[eNormX] = true;
    (*vecAllVars)[eAGradY_eFlux_T] = true;
    (*vecAllVars)[eNormY] = true;
    (*vecAllVars)[eHeatDiff] = true;
  //
  };

  // Interior Flux function
  virtual void vInteriorFlux(FluxOutput& FO) const {
    // Define the interior flux here

    double adFlux[1];

    adFlux[eFlux_T] =
      - vecValues[eHeatDiff] * (vecValues[eAGradX_eFlux_T]*vecValues[eNormX]
                              + vecValues[eAGradY_eFlux_T]*vecValues[eNormY]);
    //
    FO.vSetFlux(adFlux);
    //
    if (FO.qWantWaveSpeeds()) {
      // Set min speed
      double dMinWaveSpeed = -vecValues[iTotalNumberVariables-1] * 0.3;
      // Set max speed
      double dMaxWaveSpeed =  vecValues[iTotalNumberVariables-1] * 0.3;
      // store them..
      FO.vSetWaveSpeeds(dMinWaveSpeed, dMaxWaveSpeed);
    //
    }
  };

  // Computed variables dependencies
  virtual void vGetComputedVarDependencies(
    const int iWhich,
    std::vector<bool> * vecAllVars) {
    // Set the appropriate variables to true

    switch(iWhich) {
      case eAGradX_eFlux_T: {
        (*vecAllVars)[eRGradX_eFlux_T] = true;
        (*vecAllVars)[eLGradX_eFlux_T] = true;
      }
      break;
      case eAGradY_eFlux_T: {
        (*vecAllVars)[eRGradY_eFlux_T] = true;
        (*vecAllVars)[eLGradY_eFlux_T] = true;
      }
      break;
      case eHeatDiff: {
        (*vecAllVars)[eConduct] = true;
        (*vecAllVars)[eDens] = true;
        (*vecAllVars)[eSpecHeat] = true;
      }
      break;
      case eHeatFlux: {
        (*vecAllVars)[eConduct] = true;
        (*vecAllVars)[eAGradX_eFlux_T] = true;
        (*vecAllVars)[eNormX] = true;
        (*vecAllVars)[eAGradY_eFlux_T] = true;
        (*vecAllVars)[eNormY] = true;
      }
      break;
      case eAveTemp: {
        (*vecAllVars)[eLSoln_eFlux_T] = true;
        (*vecAllVars)[eRSoln_eFlux_T] = true;
      }
      break;
    } // end of switch on computed variable
  //
  };

  // Computed variables are... well... computed... here
  virtual double dComputeVarValue(const int iWhich) {
    // Compute the requested variable...

    double dValue;

    switch(iWhich) {
      case eAGradX_eFlux_T: {
        dValue =
          0.5 * (vecValues[eRGradX_eFlux_T] + vecValues[eLGradX_eFlux_T]);
      }
      break;
      case eAGradY_eFlux_T: {
        dValue =
          0.5 * (vecValues[eRGradY_eFlux_T] + vecValues[eLGradY_eFlux_T]);
      }
      break;
      case eHeatDiff: {
        dValue =
          vecValues[eConduct] / (vecValues[eDens] * vecValues[eSpecHeat]);
      }
      break;
      case eHeatFlux: {
        dValue =
          vecValues[eConduct] * ( vecValues[eAGradX_eFlux_T]*vecValues[eNormX]
                                + vecValues[eAGradY_eFlux_T]*vecValues[eNormY]
);
      }
      break;
      case eAveTemp: {
        dValue =
          0.5 * (vecValues[eLSoln_eFlux_T] + vecValues[eRSoln_eFlux_T]);
      }
      break;
    } // end of switch on computed variable
    // Return the value
    return dValue;
  //
  };

  // Source term dependencies
  virtual void vGetSourceTermDependencies(std::vector<bool> * vecAllVars) {
    // Set the appropriate flags to true

  //
  };

  // Source term mapping variables


  // Speeds up the mapping..
  virtual void vSetSourceTermMapping() {
  //
  };


  // Source term function itself..
  virtual void vSourceTerm(FluxOutput& FO) const {
    // Source term is defined here

  //
  };

  // Boundary flux dependencies
  virtual void vGetBoundaryTypeFluxDependencies(
    const int iBCType,
    std::vector<bool> * vecAllVars) {
    // Set the appropriate variables to true... BC variables are automatically
added

    switch(iBCType) {
      case eBC_Ins: {
      }
      break;
      case eBC_Temp: {
        (*vecAllVars)[eRGradX_eFlux_T] = true;
        (*vecAllVars)[eNormX] = true;
        (*vecAllVars)[eRGradY_eFlux_T] = true;
        (*vecAllVars)[eNormY] = true;
        (*vecAllVars)[eHeatDiff] = true;
      }
      break;
      case eBC_Flux: {
        (*vecAllVars)[eDens] = true;
        (*vecAllVars)[eSpecHeat] = true;
      }
      break;
    }; // end of switch on bdry flux variables
  //
  };

  // Boundary flux...
  virtual void vBoundaryTypeFlux(
    const int iBCType,
    const int iBCVarBaseIndex,
    FluxOutput& FO) const {
    // Compute the boundary flux...

    double adFlux[1];

    switch(iBCType) {
      case eBC_Ins: {
        adFlux[eFlux_T] =
          0;
        //
        if (FO.qWantWaveSpeeds()) {
          // Set min speed
          double dMinWaveSpeed = -vecValues[iTotalNumberVariables-1] * 0.3;
          // Set max speed
          double dMaxWaveSpeed =  vecValues[iTotalNumberVariables-1] * 0.3;
          // store them..
          FO.vSetWaveSpeeds(dMinWaveSpeed, dMaxWaveSpeed);
        //
        }
      }
      break;
      case eBC_Temp: {
        adFlux[eFlux_T] =
          - vecValues[eHeatDiff] *
              (vecValues[eRGradX_eFlux_T] * vecValues[eNormX] +
               vecValues[eRGradY_eFlux_T] * vecValues[eNormY]);
        //
        if (FO.qWantWaveSpeeds()) {
          // Set min speed
          double dMinWaveSpeed = -vecValues[iTotalNumberVariables-1] * 0.3;
          // Set max speed
          double dMaxWaveSpeed =  vecValues[iTotalNumberVariables-1] * 0.3;
          // store them..
          FO.vSetWaveSpeeds(dMinWaveSpeed, dMaxWaveSpeed);
        //
        }
      }
      break;
      case eBC_Flux: {
        adFlux[eFlux_T] =
          - vecValues[iBCVarBaseIndex + eHeatBCVar] /
            (vecValues[eDens] * vecValues[eSpecHeat]);
        //
        if (FO.qWantWaveSpeeds()) {
          // Set min speed
          double dMinWaveSpeed = -vecValues[iTotalNumberVariables-1] * 0.3;
          // Set max speed
          double dMaxWaveSpeed =  vecValues[iTotalNumberVariables-1] * 0.3;
          // store them..
          FO.vSetWaveSpeeds(dMinWaveSpeed, dMaxWaveSpeed);
        //
        }
      }
      break;
    } // end of switch on computed variable
    // Return the flux values
    FO.vSetFlux(adFlux);
  //
  };

  // Boundary constraints dependencies
  virtual void vGetBoundaryTypeConstraintDependencies(
    const int iBCType,
    std::vector<bool> * vecAllVars) {
    // Set the appropriate variables to true... BC variables are automatically
added

    switch(iBCType) {
      case eBC_Ins: {
      }
      break;
      case eBC_Temp: {
      }
      break;
      case eBC_Flux: {
      }
      break;
    }; // end of switch on bdry cons variables
  //
  };

  // Boundary constraints
  virtual void vSetupTypeConstraints(
    const int iBCType, const int iBCVarBaseIndex, const int iRegionVarIndex,
    const int iNumUnknowns, double *a2dReconArray[], double *a2dReconRHS[1],
    int &iRow, const int iDim, const double adLoc[],
    const double adRelLoc[], const double adNorm[], const int iBC) const {
    switch(iBCType) {
      case eBC_Ins: {
        ((Taylor(iRegionVarIndex + eFlux_T, iDim, adNorm,
          iNumUnknowns)).TNDeriv()).vTranscribeRow(a2dReconArray[iRow],
          adRelLoc);
        a2dReconRHS[iRow][0] = 0;
        iRow++;

      }
      break;
      case eBC_Temp: {
        ((Taylor(iRegionVarIndex + eFlux_T, iDim, adNorm, iNumUnknowns))).
          vTranscribeRow(a2dReconArray[iRow], adRelLoc);
        a2dReconRHS[iRow][0] = vecValues[iBCVarBaseIndex + eTempBCVar];
        iRow++;
      }
      break;
    }; // end of switch on bdry type
  //
  };

  // Boundary constraints
  virtual void vComputeBdryTypeData(
    const int iBCType, const int iBCVarBaseIndex, const int iRegionVarIndex,
    const FieldQuant& FQSoln, const int i, const int j,
    const enum eCVFace eWhichFace, const double adLoc[],
    const double adNorm[], double *a2dBC1[], double *a2dBC2[],
    double adRHS[]) const {

    bool qUseCartesian = false;
    bool qUseRotated = false;
    const int iInvalidVar = -200;
    int iVar, iVarInUse = iInvalidVar;
    switch(iBCType) {
      case eBC_Ins: {
        iVar = iRegionVarIndex + eFlux_T;
        // // Check for a clash between Cartesian and boundary coordinates
        if (iVar == VAR_U_NORM ||
            iVar == VAR_U_TANG ||
            iVar == VAR_U_CROSS) {
          assert(!qUseCartesian);
          qUseRotated = true;
        }
        else if (iVar == iRegionVarIndex + VAR_VEC_X ||
            iVar == iRegionVarIndex + VAR_VEC_Y ||
            iVar == iRegionVarIndex + VAR_VEC_Z) {
          assert(!qUseRotated);
          qUseCartesian = true;
        }
        // //MapVariable
        switch (iVar) {
          case VAR_U_NORM:  iVar = iRegionVarIndex + VAR_VEC_X; break;
          case VAR_U_TANG:  iVar = iRegionVarIndex + VAR_VEC_Y; break;
          case VAR_U_CROSS: iVar = iRegionVarIndex + VAR_VEC_Z; break;
          default:          iVar = iVar; break;
      }
      if (iVarInUse == iInvalidVar) {
        iVarInUse = iVar;
        if (qUseRotated) {
          double dTemp1 = adRHS[iRegionVarIndex + VAR_VEC_X]*adNorm[XDIR] +
              adRHS[iRegionVarIndex + VAR_VEC_Y]*adNorm[YDIR];
          double dTemp2 = adRHS[iRegionVarIndex + VAR_VEC_Y]*adNorm[XDIR] -
              adRHS[iRegionVarIndex + VAR_VEC_X]*adNorm[YDIR];
          adRHS[iRegionVarIndex + VAR_VEC_X] = dTemp1;
          adRHS[iRegionVarIndex + VAR_VEC_Y] = dTemp2;
        }
        // // Zero this constraint to avoid contamination
        a2dBC1[iVar][iVar] = 0;
        a2dBC2[iVar][iVar] = 0;
        adRHS[iVar] = 0;
      }
      else {
        assert(iVarInUse == iVar);
      }
      a2dBC1[iVarInUse][iVarInUse] = 1;
      adRHS[iVarInUse] =  0;
      iVarInUse = iInvalidVar;

    }
    break;
    case eBC_Temp: {
      iVar = iRegionVarIndex + eFlux_T;
      // // Check for a clash between Cartesian and boundary coordinates
      if (iVar == VAR_U_NORM ||
          iVar == VAR_U_TANG ||
          iVar == VAR_U_CROSS) {
        assert(!qUseCartesian);
        qUseRotated = true;
      }
      else if (iVar == iRegionVarIndex + VAR_VEC_X ||
          iVar == iRegionVarIndex + VAR_VEC_Y ||
          iVar == iRegionVarIndex + VAR_VEC_Z) {
        assert(!qUseRotated);
        qUseCartesian = true;
      }
      // //MapVariable
```

```
        switch (iVar) {
          case VAR_U_NORM:  iVar = iRegionVarIndex + VAR_VEC_X; break;
          case VAR_U_TANG:  iVar = iRegionVarIndex + VAR_VEC_Y; break;
          case VAR_U_CROSS: iVar = iRegionVarIndex + VAR_VEC_Z; break;
          default:          iVar = iVar; break;
        }
        if (iVarInUse == iInvalidVar) {
          iVarInUse = iVar;
          if (qUseRotated) {
            double dTemp1 = adRHS[iRegionVarIndex + VAR_VEC_X]*adNorm[XDIR] +
                adRHS[iRegionVarIndex + VAR_VEC_Y]*adNorm[YDIR];
            double dTemp2 = adRHS[iRegionVarIndex + VAR_VEC_Y]*adNorm[XDIR] -
                adRHS[iRegionVarIndex + VAR_VEC_X]*adNorm[YDIR];
            adRHS[iRegionVarIndex + VAR_VEC_X] = dTemp1;
            adRHS[iRegionVarIndex + VAR_VEC_Y] = dTemp2;
          }
          // // Zero this constraint to avoid contamination
          a2dBC1[iVar][iVar] = 0;
          a2dBC2[iVar][iVar] = 0;
          adRHS[iVar] = 0;
        }
        else {
          assert(iVarInUse == iVar);
        }
        a2dBC2[iVarInUse][iVarInUse] = 1;
        adRHS[iVarInUse] =  vecValues[iBCVarBaseIndex + eTempBCVar];
        iVarInUse = iInvalidVar;

      }
      break;
    }; // end of switch on bdry type
  //
  };

  // This needs to be static since it will be called as a function variable
  static void vInitialSolution(const double adLoc[3], double adResult[]) {
    adResult[eFlux_T] = 0.;
  };

  virtual void vInitializeSolution(
    const ANS_Mesh* const pM, FieldQuant& FQSoln,
    const bool qDoTranslation = true) const {
    // Simple call here
    vComputeCVAverages(pM,
      (void (*)(const double *, double *))(vInitialSolution),
      6, FQSoln, 1);
  };

};
```

B.3 Solver file for heat conduction simulation

Here is the C++ file for the main program that performs a heat conduction simulation.

```
#include <math.h>
#include <unistd.h>

#include "Heat2D_Physics.cxx"
#include "FP_Physics.cxx"

#include "ANS_Domain.h"
#include "ANS_MeshCell2D.h"
#include "ANS_MeshVertex2D.h"
#include "ANS_Multistage.h"
#include "ANS_Region.h"
#include "ANS_PhysVar.h"
#include <sstream>
#include <fstream>

int main() {
  char strMeshFile[128];
  int iOrder = 2;

  // Initialize ANSLib
  vANSLibInit();

  // Create mesh
  sprintf(strMeshFile, "../../meshes/channel-CSM-r8g4");
  ANS_Mesh *pUM2D1 = new MeshCell2D(strMeshFile, iOrder);
  std::vector< ANS_Mesh* > vecUM2D;

  vecUM2D.clear();
  vecUM2D.push_back(pUM2D1);

  // Create domain
  Domain *pDomain = new Domain(vecUM2D);
  Region *pR0 = pDomain->pRegion(0);

  NewPhysics *pnpFP1, *pnpH1;

  // Function provider
  pnpFP1  = new FP_Physics();
  // Heat conduction class
  pnpH1 = new Heat2D_Physics();

  pR0->vAddNewPhysics(pnpH1);
  pR0->vAddNewPhysics(pnpFP1);

  // Assign boundary conditions

//          4
//     ----------
// 2  |          |
//    |          |
//    |          |  3
//    |          |
//    |__________|
//
//           1

  // For the Heat conduction
  pnpH1->vSetBC(1, Heat2D_Physics::eBC_Temp);
  pnpH1->vSetBC(2, Heat2D_Physics::eBC_Temp);
  pnpH1->vSetBC(3, Heat2D_Physics::eBC_Temp);
  pnpH1->vSetBC(4, Heat2D_Physics::eBC_Temp);

  pDomain->vCopyVariables();


#define PHYS_H  0
```

```
#define PHYS_FP   1

  // Variable association

  // k
  pDomain->vAssociateVariables(0, PHYS_H, Heat2D_Physics::eConduct,
    0, PHYS_FP, FP_Physics::eOne);
  // rho
  pDomain->vAssociateVariables(0, PHYS_H, Heat2D_Physics::eDens,
    0, PHYS_FP, FP_Physics::eOne);
  // cp
  pDomain->vAssociateVariables(0, PHYS_H, Heat2D_Physics::eSpecHeat,
    0, PHYS_FP, FP_Physics::eOne);

  // Boundary condition variables

  // Inlet temp
  pDomain->vAssociateVariables(0, PHYS_H, 2, 0,
    0, PHYS_FP, FP_Physics::eTempIncY);
  // Outlet temp
  pDomain->vAssociateVariables(0, PHYS_H, 3, 0,
    0, PHYS_FP, FP_Physics::eTempDecY);
  // Bottom temp
  pDomain->vAssociateVariables(0, PHYS_H, 1, 0,
    0, PHYS_FP, FP_Physics::eTempIncX);
  // Top temp
  pDomain->vAssociateVariables(0, PHYS_H, 4, 0,
    0, PHYS_FP, FP_Physics::eTempDecX);

  pDomain->vSetupVariables();

  Multistage MS;
  pDomain->vInitializeSolution();

  double dObjective = 1e-13;

  MS.vSetCFL(0.3);
  MS.vSetOverrelaxationParam(3.95);
  // Start computing
  MS.vRelaxAbsolute(pDomain, iOrder, dObjective);
  // Write solution
  pDomain->vSetupSolutionFile();
  // Add the vars you want to see in the solution file
  pDomain->vAddSolutionFileVariable(0, PHYS_H, Heat2D_Physics::eFlux_T);

  sFile = "HeatConduction.solution.pie";
  pDomain->vWriteSolutionFile(sFile, iOrder);

  // Clean up before exit
  delete pDomain;

  return 0;
}
```

B.4 Solid mechanics XML file

```
<?xml version="1.0" ?>
<Physics>
  <ClassID>CSM2D</ClassID>
  <ClassDesc>2D Computational solid mechanics package</ClassDesc>
    <FluxVar>
      <ID>eFlux_U</ID>
      <Name>Displacement in x</Name>
      <Symbol>u</Symbol>
      <Units> <L>1</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    </FluxVar>
    <FluxVar>
      <ID>eFlux_V</ID>
      <Name>Displacement in y</Name>
      <Symbol>v</Symbol>
      <Units> <L>1</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    </FluxVar>
    <ComputedVar>
      <ID>eAveDispX</ID>
      <Name>Displacement in X</Name>
      <Symbol>uave</Symbol>
      <Units> <L>1</L> <M>0</M> <T>0</T> <K>0</K> </Units>
      <Formula>Soln{eFlux_U}</Formula>
    </ComputedVar>
```

```
<ComputedVar>
  <ID>eAveDispY</ID>
  <Name>Displacement in Y</Name>
  <Symbol>vave</Symbol>
  <Units> <L>1</L> <M>0</M> <T>0</T> <K>0</K> </Units>
  <Formula>Soln{eFlux_U}</Formula>
</ComputedVar>
<RequiredVar>
  <ID>ePoisson</ID>
  <Name>Poisson's ratio</Name>
  <Symbol>nu</Symbol>
  <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
</RequiredVar>
<RequiredVar>
  <ID>eElastMod</ID>
  <Name>ElasticModulus</Name>
  <Symbol>E</Symbol>
  <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
</RequiredVar>
<RequiredVar>
  <ID>eExpCoeff</ID>
  <Name>Expansion coefficient</Name>
  <Symbol>at</Symbol>
  <Units> <L>1</L> <M>0</M> <T>0</T> <K>-1</K> </Units>
</RequiredVar>
<RequiredVar>
  <ID>eRefTemp</ID>
  <Name>Reference Temperature</Name>
  <Symbol>Tref</Symbol>
  <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
</RequiredVar>
<RequiredVar>
  <ID>eTemp</ID>
  <Name>Temperature</Name>
  <Symbol>T</Symbol>
  <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
</RequiredVar>
<ComputedVar>
  <ID>eStrainXX</ID>
  <Name>Strain in x</Name>
  <Symbol>exx</Symbol>
```

```
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    <Formula>GradX{eFlux_U} - eExpCoeff*(eTemp - eRefTemp)</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStrainYY</ID>
    <Name>Strain in y</Name>
    <Symbol>eyy</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    <Formula>GradY{eFlux_V} - eExpCoeff*(eTemp - eRefTemp)</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStrainXY</ID>
    <Name>Shear Strain (x-y)</Name>
    <Symbol>exy</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    <Formula>0.5 * (GradX{eFlux_V} + GradY{eFlux_U})</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStressXX</ID>
    <Name>Stress in x</Name>
    <Symbol>sxx</Symbol>
    <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    <Formula>(eElastMod/(1.0 - ePoisson*ePoisson))*
                ( eStrainXX + ePoisson*eStrainYY )</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStressYY</ID>
    <Name>Stress in y</Name>
    <Symbol>syy</Symbol>
    <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    <Formula>(eElastMod/(1.0 - ePoisson*ePoisson))*
                ( eStrainYY + ePoisson*eStrainXX )</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStressXY</ID>
    <Name>Shear stress (x-y)</Name>
    <Symbol>sxy</Symbol>
    <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    <Formula>(eElastMod * eStrainXY)/(1.0 + ePoisson)</Formula>
  </ComputedVar>
  <ComputedVar>
```

```
      <ID>eBdryDUDX</ID>
      <Name>Bdry du/dx</Name>
      <Symbol>uxb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
      <Formula>RGradX{eFlux_U}</Formula>
    </ComputedVar>
    <ComputedVar>
      <ID>eBdryDUDY</ID>
      <Name>Bdry du/dy</Name>
      <Symbol>uyb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
      <Formula>RGradY{eFlux_U}</Formula>
    </ComputedVar>
    <ComputedVar>
      <ID>eBdryDVDX</ID>
      <Name>Bdry du/dx</Name>
      <Symbol>vxb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
      <Formula>RGradX{eFlux_V}</Formula>
    </ComputedVar>
    <ComputedVar>
      <ID>eBdryDVDY</ID>
      <Name>Bdry du/dy</Name>
      <Symbol>vyb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
      <Formula>RGradY{eFlux_V}</Formula>
    </ComputedVar>
  <UserFunction>
    void vGenericFlux(const double ddudx, const double ddudy,
                      const double ddvdx, const double ddvdy,
                      const double dE, const double dnu,
                      const double dTref, const double dTemp, const double dat,
                      const double dnx, const double dny,
                      double adFlux[2]) const {
      // Generic flux for CSM problem
      double adFluxX[2];
      double adFluxY[2];
      const double dMulConst = dE / (1.0 - dnu*dnu);
      const double dTempAdd  = (1.0 + dnu)*dat*(dTemp - dTref);

      adFluxX[0] = ddudx + dnu*ddvdy - dTempAdd;
      adFluxX[1] = 0.25*(1.0 + dnu)*(ddudy + ddvdx);
      adFluxY[0] = adFluxX[1];
      adFluxY[1] = ddvdy + dnu*ddudx - dTempAdd;

      adFlux[0] = dMulConst*(-adFluxX[0]*dnx - adFluxY[0]*dny);
      adFlux[1] = dMulConst*(-adFluxX[1]*dnx - adFluxY[1]*dny);
    }
  </UserFunction>
  <InteriorFlux>
    <UserCode>
      vGenericFlux(GradX{eFlux_U}, GradY{eFlux_U},
          GradX{eFlux_V}, GradY{eFlux_V},
          eElastMod, ePoisson, eRefTemp, eTemp, eExpCoeff,
          Norm{X}, Norm{Y}, adFlux);
    </UserCode>
  </InteriorFlux>
  <WaveSpeeds>
    <Minimum>-eInvDist * 0.3 * eElastMod</Minimum>
    <Maximum> eInvDist * 0.3 * eElastMod</Maximum>
  </WaveSpeeds>
  <SourceTerm>
  </SourceTerm>
  <BCType>
    <Desc>This BC imposes a free boundary</Desc>
    <ID>eBC_Free</ID>
    <BdryFlux>
      <FluxData>
        <Var>eFlux_U</Var>
        <Formula>0</Formula>
      </FluxData>
      <FluxData>
        <Var>eFlux_U</Var>
        <Formula>0</Formula>
      </FluxData>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes a x-displacement at the wall</Desc>
    <ID>eBC_DispX</ID>
    <BCVar>
      <ID>eBCDispXVar</ID>
      <Name>Wall x-displacement</Name>
      <Symbol>xb</Symbol>
      <Units> <L>1</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    </BCVar>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_U</Var>
      <Formula>eBCDispXVar</Formula>
    </Constraint>
    <BdryFlux>
      <UserCode>
        vGenericFlux(RGradX{eFlux_U}, RGradY{eFlux_U},
            RGradX{eFlux_V}, RGradY{eFlux_V},
            eElastMod, ePoisson, eRefTemp, eTemp, eExpCoeff,
            Norm{X}, Norm{Y}, adFlux);
     </UserCode>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes a y-displacement at the wall</Desc>
    <ID>eBC_DispY</ID>
    <BCVar>
      <ID>eBCDispYVar</ID>
      <Name>Wall y-displacement</Name>
      <Symbol>yb</Symbol>
      <Units> <L>1</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    </BCVar>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_V</Var>
      <Formula>eBCDispYVar</Formula>
    </Constraint>
    <BdryFlux>
      <UserCode>
        vGenericFlux(RGradX{eFlux_U}, RGradY{eFlux_U},
            RGradX{eFlux_V}, RGradY{eFlux_V},
            eElastMod, ePoisson, eRefTemp, eTemp, eExpCoeff,
            Norm{X}, Norm{Y}, adFlux);
     </UserCode>
    </BdryFlux>
  </BCType>
```

```
<BCType>
  <Desc>This BC imposes a normal stress at the wall</Desc>
  <ID>eBC_NormStress</ID>
  <BCVar>
    <ID>eBCNormStressVar</ID>
    <Name>Wall normal stress</Name>
    <Symbol>snb</Symbol>
    <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
  </BCVar>
  <BdryFlux>
    <UserCode>
        vGenericFlux(RGradX{eFlux_U}, RGradY{eFlux_U},
          RGradX{eFlux_V}, RGradY{eFlux_V},
          eElastMod, ePoisson, eRefTemp, eTemp, eExpCoeff,
          Norm{X}, Norm{Y}, adFlux);

        // THis will only work if the boundaries are aligned with the coords
        if (Norm{X} != 0.0) {
          // vertical wall
          // normal stress is sxx
          adFlux[eFlux_U] = (eBCNormStressVar *
                              (1 - ePoisson*ePoisson))/eElastMod;
        }
        else {
          // horizontal wall -- syy = 0
          adFlux[eFlux_V] = (eBCNormStressVar *
                              (1 - ePoisson*ePoisson))/eElastMod;
        }
    </UserCode>
  </BdryFlux>
</BCType>
<BCType>
  <Desc>This BC imposes a shear stress at the wall</Desc>
  <ID>eBC_ShearStress</ID>
  <BCVar>
    <ID>eBCShearStressVar</ID>
    <Name>Wall shear stress</Name>
    <Symbol>ssb</Symbol>
    <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
  </BCVar>
  <BdryFlux>
```

```
      <UserCode>
          vGenericFlux(RGradX{eFlux_U}, RGradY{eFlux_U},
            RGradX{eFlux_V}, RGradY{eFlux_V},
            eElastMod, ePoisson, eRefTemp, eTemp, eExpCoeff,
            Norm{X}, Norm{Y}, adFlux);

          // THis will only work if the boundaries are aligned with the coords
          if (Norm{X} != 0.0) {
            // vertical wall
            // shear stress is sxy
            adFlux[eFlux_V] = (eBCShearStressVar *
                               (1 - ePoisson*ePoisson))/eElastMod;
          }
          else {
            // horizontal wall -- syy = 0
            adFlux[eFlux_U] = (eBCShearStressVar *
                               (1 - ePoisson*ePoisson))/eElastMod;
          }
      </UserCode>
    </BdryFlux>
  </BCType>
    <BCType>
    <Desc>This BC imposes both shear and normal stresses at the wall</Desc>
    <ID>eBC_Stresses</ID>
    <BCVar>
      <ID>eBCStressesVarXX</ID>
      <Name>Wall xx-stress</Name>
      <Symbol>sxxb</Symbol>
      <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    </BCVar>
    <BCVar>
      <ID>eBCStressesVarYY</ID>
      <Name>Wall yy-stress</Name>
      <Symbol>syyb</Symbol>
      <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    </BCVar>
    <BCVar>
      <ID>eBCStressesVarXY</ID>
      <Name>Wall shear stress</Name>
      <Symbol>ssb</Symbol>
      <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
      </BCVar>
      <BdryFlux>
        <UserCode>
          adFlux[eFlux_U] = eBCStressesVarXX * Norm{X} + eBCStressesVarXY * Norm{Y};
          adFlux[eFlux_V] = eBCStressesVarXY * Norm{X} + eBCStressesVarYY * Norm{Y};
        </UserCode>
      </BdryFlux>
    </BCType>
    <BCType>
      <Desc>This BC imposes both shear and normal stresses at the wall</Desc>
      <ID>eBC_IntBdry</ID>
      <BCVar>
        <ID>eIntDUDX</ID>
        <Name>du/dx</Name>
        <Symbol>uxb</Symbol>
        <Units> <L>0</L> <M>0</M> <T>-2</T> <K>0</K> </Units>
      </BCVar>
      <BCVar>
        <ID>eIntDUDY</ID>
        <Name>du/dy</Name>
        <Symbol>uyb</Symbol>
        <Units> <L>0</L> <M>0</M> <T>-2</T> <K>0</K> </Units>
      </BCVar>
      <BCVar>
        <ID>eIntDVDX</ID>
        <Name>dv/dx</Name>
        <Symbol>vxb</Symbol>
        <Units> <L>0</L> <M>0</M> <T>-2</T> <K>0</K> </Units>
      </BCVar>
      <BCVar>
        <ID>eIntDVDY</ID>
        <Name>dv/dy</Name>
        <Symbol>vyb</Symbol>
        <Units> <L>0</L> <M>0</M> <T>-2</T> <K>0</K> </Units>
      </BCVar>
      <BCVar>
        <ID>eIntTemp</ID>
        <Name>Tb</Name>
        <Symbol>vyb</Symbol>
        <Units> <L>0</L> <M>0</M> <T>-2</T> <K>0</K> </Units>
      </BCVar>
    <BdryFlux>
      <UserCode>
        double dudx = 0.5 * (RGradX{eFlux_U} + eIntDUDX);
        double dudy = 0.5 * (RGradY{eFlux_U} + eIntDUDY);
        double dvdx = 0.5 * (RGradX{eFlux_V} + eIntDVDX);
        double dvdy = 0.5 * (RGradY{eFlux_V} + eIntDVDY);
        double dTemp = 0.5 * (eTemp + eIntTemp);

        vGenericFlux(dudx, dudy, dvdx, dvdy,
            eElastMod, ePoisson, eRefTemp, dTemp, eExpCoeff,
            Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes a symmetry BC along a horizontal wall</Desc>
    <ID>eBC_HorSymm</ID>
    <BdryFlux>
      <UserCode>
        vGenericFlux(RGradX{eFlux_U}, RGradY{eFlux_U},
            RGradX{eFlux_V}, RGradY{eFlux_V},
            eElastMod, ePoisson, eRefTemp, eTemp, eExpCoeff,
            Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_V</Var>
      <Formula>0</Formula>
    </Constraint>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_U</Var>
      <Formula>0</Formula>
    </Constraint>
  </BCType>
  <BCType>
    <Desc>This BC imposes a symmetry BC along a vertical wall</Desc>
    <ID>eBC_VerSymm</ID>
    <BdryFlux>
      <UserCode>
```

```
        vGenericFlux(RGradX{eFlux_U}, RGradY{eFlux_U},
            RGradX{eFlux_V}, RGradY{eFlux_V},
            eElastMod, ePoisson, eRefTemp, eTemp, eExpCoeff,
            Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_U</Var>
      <Formula>0</Formula>
    </Constraint>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_V</Var>
      <Formula>0</Formula>
    </Constraint>
  </BCType>
</Physics>
```

B.5 Incompressible Navier-Stokes XML file

```
<?xml version="1.0" ?>
<Physics>
  <ClassID>INS2D</ClassID>
  <ClassDesc>2D Incompressible Navier-Stokes package</ClassDesc>
    <FluxVar>
      <ID>eFlux_P</ID>
      <Name>Pressure</Name>
      <Symbol>P</Symbol>
      <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    </FluxVar>
    <FluxVar>
      <ID>eFlux_U</ID>
      <Name>Velocity in x</Name>
      <Symbol>u</Symbol>
      <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    </FluxVar>
    <FluxVar>
      <ID>eFlux_V</ID>
      <Name>Velocity in y</Name>
    <Symbol>v</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </FluxVar>
  <RequiredVar>
    <ID>eReynolds</ID>
    <Name>Reynolds Number</Name>
    <Symbol>Re</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
  </RequiredVar>
  <RequiredVar>
   <ID>eBeta</ID>
    <Name>Artificial Compressibility</Name>
    <Symbol>B</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
  </RequiredVar>
  <ComputedVar>
   <ID>eBdryP</ID>
    <Name>Bdry Pressure</Name>
    <Symbol>Pb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LSoln{eFlux_P}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryU</ID>
    <Name>Bdry U Velocity</Name>
    <Symbol>Ub</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LSoln{eFlux_U}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryV</ID>
    <Name>Bdry V Velocity</Name>
    <Symbol>Vb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LSoln{eFlux_V}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryDUDX</ID>
    <Name>Bdry du/dx</Name>
    <Symbol>Uxb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LGradX{eFlux_U}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryDUDY</ID>
    <Name>Bdry du/dy</Name>
    <Symbol>Uyb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LGradY{eFlux_U}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryDVDX</ID>
    <Name>Bdry dv/dx</Name>
    <Symbol>Vxb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LGradX{eFlux_V}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eBdryDVDY</ID>
    <Name>Bdry dv/dy</Name>
    <Symbol>Vyb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LGradY{eFlux_V}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eNormVel</ID>
    <Name>Normal Velocity</Name>
    <Symbol>U</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>Soln{eFlux_U}*Norm{X} + Soln{eFlux_V}*Norm{Y}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eNormPGrad</ID>
    <Name>dP/dn</Name>
    <Symbol>Pn</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>GradX{eFlux_P}*Norm{X} + GradY{eFlux_P}*Norm{Y}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eNormUGrad</ID>
    <Name>du/dn</Name>
    <Symbol>Un</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>GradX{eFlux_U}*Norm{X} + GradY{eFlux_U}*Norm{Y}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eNormVGrad</ID>
    <Name>Normal Velocity</Name>
    <Symbol>dv/dn</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>GradX{eFlux_V}*Norm{X} + GradY{eFlux_V}*Norm{Y}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eUpWind_U</ID>
    <Name>Upwind x-velocity</Name>
    <Symbol>u_up</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>((eNormVel > 0) ?  LSoln{eFlux_U} :  RSoln{eFlux_U})</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eUpWind_V</ID>
    <Name>Upwind y-velocity</Name>
    <Symbol>v_up</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>((eNormVel > 0) ?  LSoln{eFlux_V} :  RSoln{eFlux_V})</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eUpNormVel</ID>
    <Name>Upwind Normal Velocity</Name>
    <Symbol>U_up</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>eUpWind_U * Norm{X} + eUpWind_V * Norm{Y}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStrainXX</ID>
    <Name>Strain in x</Name>
    <Symbol>txx</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LGradX{eFlux_U}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStrainYY</ID>
    <Name>Strain in y</Name>
    <Symbol>tyy</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>LGradY{eFlux_V}</Formula>
  </ComputedVar>
  <ComputedVar>
    <ID>eStrainXY</ID>
    <Name>Shear strain (x-y)</Name>
    <Symbol>txy</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    <Formula>0.5 * (LGradY{eFlux_U} + LGradX{eFlux_V})</Formula>
  </ComputedVar>
<UserFunction>
  void vGenericFlux(const double dRe, const double dBeta,
                    const double dNormVel, const double dP, const double du,
                    const double dv, const double ddudx, const double ddudy,
                    const double ddvdx, const double ddvdy, const double dnx,
                    const double dny, double adFlux[3]) const {
    // Generic flux for INS problem
    const double dInvRe = 1.0/dRe;

    adFlux[0] = dNormVel/dBeta;
    adFlux[1] = dNormVel*du + (dP - dInvRe*ddudx)*dnx +
                              (   - dInvRe*ddudy)*dny;
    adFlux[2] = dNormVel*dv + (   - dInvRe*ddvdx)*dnx +
                              (dP - dInvRe*ddvdy)*dny;
  }
</UserFunction>
<UserFunction>
  double dLambda(const double dUNorm, const double dBeta) const {
    // Returns a value needed for wavespeed..
    return sqrt(dUNorm*dUNorm + 1./dBeta);
  }
</UserFunction>
<WaveSpeeds>
  <Minimum>dUNorm - dLambda(dUNorm, eBeta) - eInvDist/eReynolds</Minimum>
  <Maximum>dUNorm + dLambda(dUNorm, eBeta) + eInvDist/eReynolds</Maximum>
</WaveSpeeds>
<InteriorFlux>
  <UserCode>
    double du = eUpWind_U;
    double dv = eUpWind_V;
```

```
      double dUNorm = eUpNormVel;

      vGenericFlux(eReynolds, eBeta, dUNorm, Soln{eFlux_P}, du, dv,
          GradX{eFlux_U}, GradY{eFlux_U}, GradX{eFlux_V}, GradY{eFlux_V},
          Norm{X}, Norm{Y}, adFlux);
    </UserCode>
  </InteriorFlux>
  <SourceTerm>
  </SourceTerm>
  <BCType>
    <Desc>This BC imposes a stationary wall</Desc>
    <ID>eBC_StatWall</ID>
    <BdryFlux>
      <UserCode>
        double dUNorm = 0.;
        vGenericFlux(eReynolds, eBeta, dUNorm, LSoln{eFlux_P}, 0, 0,
                     LGradX{eFlux_U}, LGradY{eFlux_U}, LGradX{eFlux_V},
                     LGradY{eFlux_V}, Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_U</Var>
      <Formula>0</Formula>
    </Constraint>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_V</Var>
      <Formula>0</Formula>
    </Constraint>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_P</Var>
      <Formula>0</Formula>
    </Constraint>
  </BCType>
  <BCType>
    <Desc>This BC imposes an inflow BC</Desc>
    <ID>eBC_Inflow</ID>
    <BCVar>
      <ID>eBCInflowNormVel</ID>
    <Name>Normal inflow velocity</Name>
    <Symbol>Vnb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCInflowTangVel</ID>
    <Name>Tangential inflow velocity</Name>
    <Symbol>Vtb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCInflowNormPressGrad</ID>
    <Name>Normal inflow pressure gradient</Name>
    <Symbol>Pnb</Symbol>
    <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
  </BCVar>
  <Constraint>
    <Type>NGradient</Type>
    <Var>eFlux_P</Var>
    <Formula>eBCInflowNormPressGrad</Formula>
  </Constraint>
  <Constraint>
    <Type>Solution</Type>
    <Var>eFlux_U</Var>
    <Formula>eBCInflowNormVel</Formula>
  </Constraint>
  <Constraint>
    <Type>Solution</Type>
    <Var>eFlux_V</Var>
    <Formula>eBCInflowTangVel</Formula>
  </Constraint>
  <BdryFlux>
    <UserCode>
    double dUNorm = LSoln{eFlux_U}*Norm{X} + LSoln{eFlux_V}*Norm{Y};
    vGenericFlux(eReynolds, eBeta, dUNorm, LSoln{eFlux_P},
                 LSoln{eFlux_U}, LSoln{eFlux_V}, LGradX{eFlux_U},
                 LGradY{eFlux_U}, LGradX{eFlux_V}, LGradY{eFlux_V},
                 Norm{X}, Norm{Y}, adFlux);
    </UserCode>
  </BdryFlux>
</BCType>
```

```
<BCType>
  <Desc>This BC imposes an outflow BC</Desc>
  <ID>eBC_Outflow</ID>
  <Constraint>
    <Type>Solution</Type>
    <Var>eFlux_P</Var>
    <Formula>0</Formula>
  </Constraint>
  <Constraint>
    <Type>Solution</Type>
    <Var>eFlux_V</Var>
    <Formula>0</Formula>
  </Constraint>
  <Constraint>
    <Type>NGradient</Type>
    <Var>eFlux_U</Var>
    <Formula>0</Formula>
  </Constraint>
  <BdryFlux>
    <UserCode>
    double dUNorm = LSoln{eFlux_U}*Norm{X} + LSoln{eFlux_V}*Norm{Y};
    vGenericFlux(eReynolds, eBeta, dUNorm, LSoln{eFlux_P},
                 LSoln{eFlux_U}, LSoln{eFlux_V}, LGradX{eFlux_U},
                 LGradY{eFlux_U}, LGradX{eFlux_V}, LGradY{eFlux_V},
                 Norm{X}, Norm{Y}, adFlux);
   </UserCode>
  </BdryFlux>
</BCType>
<BCType>
  <Desc>This BC imposes an internal boundary</Desc>
  <ID>eBC_IntBdry</ID>
  <BCVar>
   <ID>eBCIntBdryP</ID>
    <Name>Bdry Pressure</Name>
    <Symbol>Pb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCIntBdryU</ID>
    <Name>Bdry U velocity</Name>
    <Symbol>Ub</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCIntBdryV</ID>
    <Name>Bdry V velocity</Name>
    <Symbol>Vb</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCIntBdryDUDX</ID>
    <Name>Bdry dudx</Name>
    <Symbol>Uxb</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCIntBdryDUDY</ID>
    <Name>Bdry dudy</Name>
    <Symbol>Uyb</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCIntBdryDVDX</ID>
    <Name>Bdry dvdx</Name>
    <Symbol>Vxb</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BCVar>
    <ID>eBCIntBdryDVDY</ID>
    <Name>Bdry dvdy</Name>
    <Symbol>Vyb</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </BCVar>
  <BdryFlux>
    <UserCode>
    // Velocities are upwind -- might be from other side
    double dUNorm = LSoln{eFlux_U}*Norm{X} + LSoln{eFlux_V}*Norm{Y};
    double du, dv, dP, dudx, dudy, dvdx, dvdy;

    if (dUNorm > 0) {
      // Get left data from other side
      du = eBCIntBdryU;
      dv = eBCIntBdryV;
    }
    else {
      du = LSoln{eFlux_U};
      dv = LSoln{eFlux_V};
    }

    dUNorm = du*Norm{X} + dv*Norm{Y};

    // Pressure and velocity gradients are the average
    dP   = 0.5 * (LSoln{eFlux_P} + eBCIntBdryP);
    dudx = 0.5 * (LGradX{eFlux_U} + eBCIntBdryDUDX);
    dudy = 0.5 * (LGradY{eFlux_U} + eBCIntBdryDUDY);
    dvdx = 0.5 * (LGradX{eFlux_V} + eBCIntBdryDVDX);
    dvdy = 0.5 * (LGradY{eFlux_V} + eBCIntBdryDVDY);

    vGenericFlux(eReynolds, eBeta, dUNorm, dP, du, dv,
        dudx, dudy, dvdx, dvdy, Norm{X}, Norm{Y},
        adFlux);

    </UserCode>
  </BdryFlux>
</BCType>
<BCType>
  <Desc>This BC imposes an outflow BC</Desc>
  <ID>eBC_Outflow2</ID>
  <BdryFlux>
    <UserCode>
    double dUNorm = LSoln{eFlux_U}*Norm{X} + LSoln{eFlux_V}*Norm{Y};
    vGenericFlux(eReynolds, eBeta, dUNorm, 0., LSoln{eFlux_U}, 0.,
        0., LGradY{eFlux_U}, 0., LGradY{eFlux_V}, Norm{X}, Norm{Y},
        adFlux);
    </UserCode>
  </BdryFlux>
</BCType>
<BCType>
  <Desc>This BC imposes a horizontal symmetry BC</Desc>
  <ID>eBC_SymmHor</ID>
  <Constraint>
    <Type>NGradient</Type>
    <Var>eFlux_P</Var>
```

```
      <Formula>0</Formula>
    </Constraint>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_V</Var>
      <Formula>0</Formula>
    </Constraint>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_U</Var>
      <Formula>0</Formula>
    </Constraint>
    <BdryFlux>
      <UserCode>
      double dUNorm = LSoln{eFlux_U}*Norm{X} + LSoln{eFlux_V}*Norm{Y};
      vGenericFlux(eReynolds, eBeta, dUNorm, LSoln{eFlux_P},
                   LSoln{eFlux_U}, LSoln{eFlux_V}, LGradX{eFlux_U},
                   LGradY{eFlux_U}, LGradX{eFlux_V}, LGradY{eFlux_V},
                   Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes an internal outflow BC</Desc>
    <ID>eBC_IntOutflow</ID>
    <BCVar>
      <ID>eBCIntOutflowPress</ID>
      <Name>Outflow pressure</Name>
      <Symbol>Pb</Symbol>
      <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    </BCVar>
    <BCVar>
      <ID>eBCIntOutflowUn</ID>
      <Name>Outflow du/dn</Name>
      <Symbol>Unb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    </BCVar>
    <BCVar>
      <ID>eBCIntOutflowVn</ID>
      <Name>Outflow dv/dn</Name>
      <Symbol>Vnb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    </BCVar>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_P</Var>
      <Formula>eBCIntOutflowPress</Formula>
    </Constraint>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_U</Var>
      <Formula>eBCIntOutflowUn</Formula>
    </Constraint>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_V</Var>
      <Formula>eBCIntOutflowVn</Formula>
    </Constraint>
    <BdryFlux>
      <UserCode>
      double dUNorm = LSoln{eFlux_U}*Norm{X} + LSoln{eFlux_V}*Norm{Y};
      vGenericFlux(eReynolds, eBeta, dUNorm, LSoln{eFlux_P},
                   LSoln{eFlux_U}, LSoln{eFlux_V}, LGradX{eFlux_U},
                   LGradY{eFlux_U}, LGradX{eFlux_V}, LGradY{eFlux_V},
                   Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes an internal outflow BC</Desc>
    <ID>eBC_IntOutflow2</ID>
    <BCVar>
      <ID>eBC2IntOutflowPress</ID>
      <Name>Outflow pressure</Name>
      <Symbol>Pb</Symbol>
      <Units> <L>-1</L> <M>1</M> <T>-2</T> <K>0</K> </Units>
    </BCVar>
    <BCVar>
      <ID>eBC2IntOutflowUn</ID>
      <Name>Outflow du/dn</Name>
      <Symbol>Unb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    </BCVar>
    <BCVar>
      <ID>eBC2IntOutflowVn</ID>
      <Name>Outflow dv/dn</Name>
      <Symbol>Vnb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
    </BCVar>
    <BdryFlux>
      <UserCode>
      double dUNorm = LSoln{eFlux_U}*Norm{X} + LSoln{eFlux_V}*Norm{Y};
      vGenericFlux(eReynolds, eBeta, dUNorm, eBC2IntOutflowPress,
                   LSoln{eFlux_U}, LSoln{eFlux_V}, eBC2IntOutflowUn,
                   LGradY{eFlux_U}, eBC2IntOutflowVn, LGradY{eFlux_V},
                   Norm{X}, Norm{Y}, adFlux);
     </UserCode>
    </BdryFlux>
  </BCType>
</Physics>
```

B.6 Energy equation XML file

```
<?xml version="1.0" ?>
<Physics>
  <ClassID>Energy2D</ClassID>
  <ClassDesc>2D Energy equation package</ClassDesc>
    <FluxVar>
      <ID>eFlux_T</ID>
      <Name>Temperature</Name>
      <Symbol>T</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    </FluxVar>
    <RequiredVar>
      <ID>eReynolds</ID>
      <Name>Reynolds Number</Name>
      <Symbol>Re</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
    </RequiredVar>
    <RequiredVar>
      <ID>ePrandtl</ID>
      <Name>Prandtl Number</Name>
    <Symbol>Pr</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
  </RequiredVar>
  <RequiredVar>
    <ID>eEckert</ID>
    <Name>Eckert Number</Name>
    <Symbol>Ec</Symbol>
    <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
  </RequiredVar>
  <RequiredVar>
    <ID>eConduct</ID>
    <Name>Conductivity</Name>
    <Symbol>k</Symbol>
    <Units> <L>1</L> <M>1</M> <T>-3</T> <K>-1</K> </Units>
  </RequiredVar>
  <RequiredVar>
    <ID>eNormVel</ID>
    <Name>Normal velocity</Name>
    <Symbol>U</Symbol>
    <Units> <L>1</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </RequiredVar>
  <RequiredVar>
    <ID>eStrainXX</ID>
    <Name>Normal strain in x</Name>
    <Symbol>txx</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </RequiredVar>
  <RequiredVar>
    <ID>eStrainYY</ID>
    <Name>Normal strain in y</Name>
    <Symbol>tyy</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </RequiredVar>
  <RequiredVar>
    <ID>eStrainXY</ID>
    <Name>Shear strain (x-y)</Name>
    <Symbol>txy</Symbol>
    <Units> <L>0</L> <M>0</M> <T>-1</T> <K>0</K> </Units>
  </RequiredVar>
  <ComputedVar>
    <ID>eHeatFlux</ID>
```

```
      <Name>Heat Flux</Name>
      <Symbol>q</Symbol>
      <Units> <L>0</L> <M>1</M> <T>-3</T> <K>0</K> </Units>
      <Formula>eConduct * ( GradX{eFlux_T}*Norm{X} +
                            GradY{eFlux_T}*Norm{Y} )</Formula>
    </ComputedVar>
    <ComputedVar>
      <ID>eAveTemp</ID>
      <Name>Temperature</Name>
      <Symbol>T</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
      <Formula>0.5 * (LSoln{eFlux_T} + RSoln{eFlux_T})</Formula>
    </ComputedVar>
  <UserFunction>
    void vCompWaveSpeeds(const double dUNorm, const double dRe,
                         const double dPr, const double dInvDist,
                         double *dMnS, double *dMxS) const {
      double dFakeSpeed = (10.0/(dRe * dPr)) * dInvDist;
      if (dUNorm >= 0) {
        *dMnS = -dFakeSpeed;
        *dMxS = dUNorm + dFakeSpeed;
      }
      else {
        *dMnS = dUNorm - dFakeSpeed;
        *dMxS = dFakeSpeed;
      }
    }
  </UserFunction>
  <UserFunction>
    void vGenericFlux(const double dUNorm, const double dRe, const double dPr,
                      const double dEc, const double dTempL, const double dTempR,
                      const double dTdx, const double dTdy, const double dnx,
                      const double dny, double adFlux[1]) const {
      // Returns a value for the flux
      double dRePr = 1.0 / (dRe * dPr);
      // Return the flux
      if (dUNorm >= 0) {
        adFlux[0] = dUNorm * dTempL;
      }
      else {
        adFlux[0] = dUNorm * dTempR;
      }

      adFlux[0] -= dRePr * (dTdx * dnx + dTdy * dny);
    }
  </UserFunction>
  <UserFunction>
    void vGenericFluxBdry(const double dUNorm, const double dRe,
                          const double dPr, const double dEc, const double dTempL,
                          const double dTempR, const double dTdn,
                          double adFlux[1]) const {
      // Returns a value for the flux
      double dRePr = 1.0 / (dRe * dPr);
      // Return the flux
      if (dUNorm >= 0) {
        adFlux[0] = dUNorm * dTempL;
      }
      else {
        adFlux[0] = dUNorm * dTempR;
      }

      adFlux[0] -= dRePr * dTdn;
    }
  </UserFunction>
  <InteriorFlux>
    <UserCode>
      // Compute normal velocity first using average values
      double dUNorm = eNormVel;
      double dMinS, dMaxS;
      vCompWaveSpeeds(dUNorm, eReynolds, ePrandtl, eInvDist,
                      &amp;dMinS, &amp;dMaxS);

      vGenericFlux(dUNorm, eReynolds, ePrandtl, eEckert,
                   LSoln{eFlux_T}, RSoln{eFlux_T}, GradX{eFlux_T},
                   GradY{eFlux_T}, Norm{X}, Norm{Y}, adFlux);
    </UserCode>
  </InteriorFlux>
  <WaveSpeeds>
    <Minimum>dMinS</Minimum>
    <Maximum>dMaxS</Maximum>
  </WaveSpeeds>
  <SourceTerm>
```

```
    <SourceData>
      <ID>eDissFunc</ID>
      <Desc>Adds dissipation effects to the energy equation</Desc>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>0</K> </Units>
      <Formula>(-eEckert/eReynolds)*( 2.0*eStrainXX*eStrainXX +
                2.0*eStrainYY*eStrainYY + 4.0*eStrainXY*eStrainXY)</Formula>
    </SourceData>
  </SourceTerm>
  <BCType>
    <Desc>This BC imposes an insulated surface</Desc>
    <ID>eBC_Ins</ID>
    <BdryFlux>
      <UserCode>
        // Compute normal velocity first using average values
        double dUNorm = eNormVel;
        double dMinS, dMaxS;
        vCompWaveSpeeds(dUNorm, eReynolds, ePrandtl, eInvDist,
                        &amp;dMinS, &amp;dMaxS);

        vGenericFlux(dUNorm, eReynolds, ePrandtl, eEckert,
                     LSoln{eFlux_T}, LSoln{eFlux_T}, GradX{eFlux_T},
                     GradY{eFlux_T}, Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_T</Var>
      <Formula>0</Formula>
    </Constraint>
  </BCType>
  <BCType>
    <Desc>This BC imposes a temperature at the wall</Desc>
    <ID>eBC_Temp</ID>
    <BCVar>
      <ID>eTempBCVar</ID>
      <Name>Boundary temperature</Name>
      <Symbol>Tb</Symbol>
      <Units> <L>0</L> <M>0</M> <T>0</T> <K>1</K> </Units>
    </BCVar>
    <Constraint>
      <Type>Solution</Type>
      <Var>eFlux_T</Var>
      <Formula>eTempBCVar</Formula>
    </Constraint>
    <BdryFlux>
      <UserCode>
        // Compute normal velocity first using average values
        double dUNorm = eNormVel;
        double dMinS, dMaxS;
        vCompWaveSpeeds(dUNorm, eReynolds, ePrandtl, eInvDist,
                        &amp;dMinS, &amp;dMaxS);

        vGenericFlux(dUNorm, eReynolds, ePrandtl, eEckert,
                     eTempBCVar, eTempBCVar, GradX{eFlux_T},
                     GradY{eFlux_T}, Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes a heat flux at the wall</Desc>
    <ID>eBC_Flux</ID>
    <BCVar>
      <ID>eHeatBCVar</ID>
      <Name>Boundary heat flux</Name>
      <Symbol>qb</Symbol>
      <Units> <L>0</L> <M>1</M> <T>-3</T> <K>0</K> </Units>
    </BCVar>
    <Constraint>
      <Type>NGradient</Type>
      <Var>eFlux_T</Var>
      <Formula>eHeatBCVar/eConduct</Formula>
    </Constraint>
    <BdryFlux>
      <UserCode>
        // Compute normal velocity first using average values
        double dUNorm = eNormVel;
        double dMinS, dMaxS;
        vCompWaveSpeeds(dUNorm, eReynolds, ePrandtl, eInvDist,
                        &amp;dMinS, &amp;dMaxS);

        vGenericFlux(dUNorm, eReynolds, ePrandtl, eEckert,
                     LSoln{eFlux_T}, LSoln{eFlux_T}, GradX{eFlux_T},
                    GradY{eFlux_T}, Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC imposes a heat flux at the wall</Desc>
    <ID>eBC_BdryFlux</ID>
    <BCVar>
      <ID>eBdryHeatBCVar</ID>
      <Name>Boundary heat flux</Name>
      <Symbol>qb</Symbol>
      <Units> <L>0</L> <M>1</M> <T>-3</T> <K>0</K> </Units>
    </BCVar>
    <BdryFlux>
      <UserCode>
        // Compute normal velocity first using average values
        double dUNorm = eNormVel;
        double dMinS, dMaxS;
        vCompWaveSpeeds(dUNorm, eReynolds, ePrandtl, eInvDist,
                        &amp;dMinS, &amp;dMaxS);
        double dHF = -eBdryHeatBCVar / eConduct;

        vGenericFluxBdry(dUNorm, eReynolds, ePrandtl, eEckert,
                    LSoln{eFlux_T}, LSoln{eFlux_T}, dHF,
                    adFlux);
      </UserCode>
    </BdryFlux>
  </BCType>
  <BCType>
    <Desc>This BC is used at the outflow</Desc>
    <ID>eBC_Outflow</ID>
    <BdryFlux>
      <UserCode>
        // Compute normal velocity first using average values
        double dUNorm = eNormVel;
        double dMinS, dMaxS;
        vCompWaveSpeeds(dUNorm, eReynolds, ePrandtl, eInvDist,
                        &amp;dMinS, &amp;dMaxS);

        vGenericFlux(dUNorm, eReynolds, ePrandtl, eEckert,
                    LSoln{eFlux_T}, LSoln{eFlux_T}, GradX{eFlux_T},
                        GradY{eFlux_T}, Norm{X}, Norm{Y}, adFlux);
      </UserCode>
    </BdryFlux>
  </BCType>
</Physics>
```