On the Automatic Generation and Usage of Gradient Adaptive Transfinite Elements

Jonathan Leigh Dummer
B.S. (University of California, Davis) 1999

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Mechanical and Aeronautical Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Nesrin Sarigul-Klijn _____

Dean C. Karnopp _____

Don Margolis _____

Committee in Charge

2002

Table of Contents

Equation Index....iv
Illustration Index....vi
I. Abstract....1
II. Finite Element Method....2
- **1. Introduction....2**
- **2. Global Level Equations....2**
- **3. Element Level Equations....5**
- **4. Galerkin's Method of Weighted Residuals....9**
- **5. Putting it All Together....11**

III. The Boolean Sum....13
- **1. Introduction....13**
- **2. Basics....13**
- **3. Enter the Boolean Sum....16**
- **4. Generalizing to n-D....25**

IV. Binary Conventions....27
- **1. Introduction....27**
- **2. Basics....27**
- **3. Corner Numbering....28**
- **4. Default Node Specification Order....29**

V. Variable Transformation....34
- **1. Introduction....34**
- **2. The Chain Rule in Action....34**
- **3. The Jacobian....36**
- **4. Inverse of the Jacobian....37**
- **5. Determinant of the Jacobian....39**
- **6. Manifolds....40**

VI. Gauss-Green Reduction of Order....43
- **1. Introduction....43**
- **2. The Chain Rule Revisited....43**

VII. Using the Method....46
- **1. Problem Solution Cycle....46**
- **2. Tools....48**
  - a. The Nonlinear Solver....48
  - b. OpenGATE: The Mesher....49

VIII. Usage: Code Example....55
- **1. Introduction....55**
- **2. Code....55**

IX. Proofs of Concept....59
- **1. Introduction....59**
- **2. The Laplace Equation....60**

a. Equations....60
b. Physical Problem....60
c. Input....62
d. Results....63
3. Thin Flat Plate: Kirchhoff Formulation....67
a. Equations....67
b. Physical Problem....68
c. Input....69
d. Results....70
4. Heat Equation with Time....75
a. Equations....75
b. Physical Problem....75
c. Input....76
d. Results....77
X. Navier-Stokes Equation Derivation....82
1. Which and Why....82
2. The Derivation....82
XI. Navier-Stokes Example....86
1. Physical problem....86
2. Input....86
3. Results....87
XII. Advantages and Limitations of the Method....93
1. Introduction....93
2. Advantages....93
3. Limitations....94
XIII. Future Changes / Improvements....98
1. Introduction....98
2. Additions....98
3. Improvements....99
XIV. References....102

Equation Index

I. Abstract 1
II. Finite Element Method 2
Eq. (1) Elliptic Equation 5
Eq. (2) Approximation for T 8
Eq. (3) Elliptic Element Residual 9
Eq. (4) Elliptic Node Residual 10
III. The Boolean Sum 13
Eq. (5) Generic 2nd Order Polynomial 14
Eq. (6) Desired Conditions 14
Eq. (7) Resultant Simultaneous Equations 14
Eq. (8) Solved 2nd Order Polynomial 14
• Eq. (9) Alternative Conditions 14
Eq. (10) Edge Equations 17
Eq. (11) Projection in the x Direction 19
Eq. (12) Projection in the y Direction 19
Eq. (13) Cross Projection 21
Eq. (14) The Exact Polynomial 22
Eq. (15) The Exact Polynomial: 2-D 25
Eq. (16) The Exact Polynomial: 3-D 25
IV. Binary Conventions 27
Eq. (17) Position Related Value in Arbitrary Numbering Systems 28
Eq. (18) Number of Sides on an N-D Element 30
Eq. (19) Number of 1-D Elements in an N-D Element 30
V. Variable Transformation 34
Eq. (20) Element Residual in Real Space 35
Eq. (21) Modified Element Residual in Computation Space 35
Eq. (22) Change of Variables 35
Eq. (23) Chain Rule In Operation 35
Eq. (24) The Jacobian 36
Eq. (25) General Conversion of Variables in a 1st Derivative 37
Eq. (26) Exact Inverse Jacobian for a 3-D Orthonormal Element 38
Eq. (27) Sample Variable Transformation in 3-D 39
Eq. (28) Sample Jacobian from the Sample Variable Transformation 39
Eq. (29) Volume Correction Term 39
Eq. (30) Differential Length of a Parametric Line 41
VI. Gauss-Green Reduction of Order 43
Eq. (31) Residual of the Elliptic Equation, Revisited 43
Eq. (32) The Chain Rule 44
Eq. (33) The Residual Term 44
Eq. (34) Substitution Assignments 44
Eq. (35) Resultant Gauss-Green Reduction 44
Eq. (36) Gauss-Green Reduction of a 2nd Derivative 44
VII. Using the Method 46

Eq. (37) the Newton-Raphson Linearizing Equation 48
VIII. Usage: Code Example 55
IX. Proofs of Concept 59
Eq. (38) Laplace Equation in 2-D 60
Eq. (39) Laplace Boundary Condition 1 61
Eq. (40) Flat Plate Constant "D" 68
Eq. (41) Flat Plate Mxx 68
Eq. (42) Flat Plate Myy 68
Eq. (43) Flat Plate Mxy 68
Eq. (44) Flat Plate Equilibrium 68
Eq. (45) Midpoint Deflection Under Constant Load 68
Eq. (46) Flat Plate Boundary Conditions 69
Eq. (47) Heat Equation: 1-D with Time 75
Eq. (48) Particular Solution at x=0.5 75
Eq. (49) Heat and Time Boundary Conditions 75
X. Navier-Stokes Equation Derivation 82
Eq. (50) N-S: Mass in 84
Eq. (51) N-S: Mass Out 84
Eq. (52) N-S: Continuity of mass 84
Eq. (53) N-S: F=ma(x) 84
Eq. (54) N-S: F=ma(x) Expanded 84
Eq. (55) N-S: Approximations for Newtonian Fluids 84
Eq. (56) N-S: Momentum in x 84
Eq. (57) N-S: Momentum in y 85
XI. Navier-Stokes Example 86
XII. Advantages and Limitations of the Method 93
XIII. Future Changes / Improvements 98
XIV. References 102

Illustration Index

Illustration II.1 Node Locations for a Simple Element 7
Illustration II.2 Shared Node Example Layout 12
Illustration III.1 Boolean Sum Example Node Locations 17
Illustration III.2 Exact Edges 18
Illustration III.3 Projection in the x Direction 19
Illustration III.4 Projection in the y Direction 20
Illustration III.5 Sum of the x and y Projections 21
Illustration III.6 Cross Projection 22
Illustration III.7 The Exact Polynomial 22
Illustration III.8 Information Loss due to Improper Element Coupling 24
Illustration VII.1 General Problem Solution Flowchart 47
Illustration VII.2 OpenGATE Sample Screenshot 54
Illustration IX.1 Full Laplace Physical Problem 61
Illustration IX.2 One-Half Laplace Physical Problem 62
Illustration IX.3 Laplace Equation Sample Mesh 63
Illustration IX.4 Laplace Equation Sample Mesh's Solution 64
Illustration IX.5 Laplace Equation Convergence Study 66
Illustration IX.6 Kirchhoff Plate Sample Mesh : 4x4 5th 70
Illustration IX.7 Kirchhoff Plate Results : 10x10 1st 71
Illustration IX.8 Kirchhoff Plate Results : 10x10 4th 71
Illustration IX.9 Kirchhoff Plate GATE Mesh : "5 3 1" 73
Illustration IX.10 Kirchhoff Plate Convergence Study 74
Illustration IX.11 Heat with Time : Single Mesh 77
Illustration IX.12 Heat with Time Sample 2-D Solution 78
Illustration IX.13 Heat with Time : Midpoint vs. Time Solutions 79
Illustration IX.14 Heat and Time Endpoint Convergence 79
Illustration IX.15 Heat with Time : Method Error Comparison 81
Illustration X.1 Navier-Stokes Velocity Diagram 83
Illustration X.2 Navier-Stokes Forces Diagram 83
Illustration XI.1 Navier-Stokes Mesh 87
Illustration XI.2 Navier-Stokes Steady-State Solution 88
Illustration XI.3 Navier-Stokes Transient Flow Solution 90
Illustration XI.4 Navier-Stokes Transient Pressure Solution 90
Illustration XII.1 Sample 3rd Order Polynomial 95
Illustration XII.2 Sample 5th Order Polynomial 95
Illustration XII.3 Sample 7th Order Polynomial 96
Illustration XII.4 Sample 9th Order Polynomial 96

I. Abstract

This paper documents a new approach to finite element analysis. Gradient Adaptive Transfinite Elements are generated at runtime, plugged directly into the differential equations which need to be solved, and a series of equations (linear or non-linear) is the output. These equations are coupled with initial or boundary conditions and fed into a non-linear solver.

The first half of the paper covers the fundamentals of the general finite element method as well as some of the particular algorithms used by the code to generate Gradient Adaptive Transfinite Elements. Of special importance are the Boolean sum, the Jacobian matrix, and the Gauss-Green reduction of order.

Examples of this method are given in the second half of this paper. The first few problems are used as "proof of concept" problems, heat flow and flat-plate bending among them. The last example is meant to demonstrate how easy it is to apply this method to a difficult problem: fluid flow, using the non-linear, incompressible, adiabatic Navier-Stokes equations in two spatial dimensions and time.

II. Finite Element Method

1. Introduction

The Finite Element Method is one of the many tools in use today to solve Differential Equations (D.E.'s) numerically. This method is, at heart, one of the simplest. Many commercial programs (e.g. iDEAS, Patran and Nastran / ABAQUS) use finite element discretizations to get results while hiding as many of the formulation details as possible from the user. This can be of great benefit, because the user of the software need not be an expert. Lamentably, there are some fundamental principles of the finite element method which then get glossed over. In practice, this means that the resulting graphs can often hide significant inaccuracies.

What this chapter will attempt to do is present the simple, bare-bones definition of the finite element method, along with its major caveats. While this restricts its use as reading for entertainment, I will try to keep the hard repetitious calculations where they belong: in the source code. When you are done, I hope you will understand the reasoning behind the finite element method, and understand intuitively why certain operations must take place. If this goal is met, then enough groundwork has been laid to use this new method of finite element analysis.

2. Global Level Equations

Even given the simplest Differential Equations, realistically complex boundary

conditions can make it impossible to solve analytically. This is why people have come up with different methods of solving D.E.'s numerically. But regardless of how you solve a set of D.E.'s, two things are needed: a region over which the equations apply, and a set of boundary and / or initial conditions which will drive the solution.

The first requirement is called the domain. The domain is the entire region where we wish to solve the equations. The equations are merely descriptions of what is physically happening (hopefully), so if the equations are right they would in fact be applicable to the entire universe. Practically we only need to find out what is going on in the region surrounding what we are interested in. There is a bit of effort or experience needed to determine exactly how far from our object we wish to solve. If we include too much surrounding space then our solution will take forever to solve. Conversely, choosing a domain too near will solve quickly, but may be wrong because our boundary conditions could affect the solution too much.

Once we have found a suitable region, we will break this up into discrete manageable chunks, called finite elements. Each of these elements will be an approximation of the solution over only its own tiny piece of the domain. If that approximation does not exactly match the right answer, then that small element also has its own little contribution to the total error. How do we know there is an error, when we don't know what the answer is? We use something called a residual to monitor how badly the equations are satisfied. This is covered two sections from now.

The second requirement to numerically solve D.E.' s is to satisfy a set of boundary or initial conditions. This requirement may be easiest to explain with an example. Suppose I was given some data on a steel bar and was asked, "How does it deform?" "It depends.", I reply. How is the bar held in place? Are there any forces acting on the bar? A slight change in any one of these answers would then change my response.

Initial conditions are just a special case of boundary conditions where the boundary is fixed in time, not space. Time can be treated just like any other dimension in the finite element method. This is not to say that it is usually treated as such. For the most part equations which have time in them are split apart ("separation of variables"), and the temporal part is treated very differently from the spatial part. In my code all of the independent variables [time, x, y, and z] are treated like dimensions, and everything else is a field variable.

Boundary conditions allow us to fix "the answer" at some part of the domain, and then we can let the equations and associated solver find all the answers to the equations over the rest of the domain. Here again there is need for experience or effort, because too few boundary conditions will not be able to drive our solution. We can make assumptions (i.e. fake boundary conditions) to let our solver work, but that can be dangerous. If assumptions are required to solve a given problem, be sure to always keep in the back of your mind what they are and how they may distort your answers.

A point worth mentioning here is the difference between linear and non-linear equations. From algebra days you know that to solve for N variables requires N

equations. This is still true of non-linear problems. The difference is that for linear systems N variables and N linearly independent equations always yield exactly one answer. For a non-linear system things get much worse. For example: five variables x1 through x5, and five quadratic equations there are 32 possible answers (2^5). Imagine a more realistic problem with 1,000 variables (still a relatively simple problem), and equations of higher order than 2. The number of possible solutions is staggering, and yet it tends to be fiendishly difficult to get a non-linear solver to converge to even one right answer.

3. Element Level Equations

So now we have the requisite information to get a good solution of our differential equations (domain and initial / boundary conditions). The next step is getting our D.E.' s to apply to each element individually. Remember that they are called differential equations because they were derived with a differential (infinitely small) element. They describe the behavior of the material (or fluid, field, etc.) at a point. We need to go from an infinite number of points which make up our finite element, to a number. And to do that requires integration.

This will be easier with an example. I will use the simple elliptic equation:

Eq. (1) $$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}=0$$ *Elliptic Equation*

This equation states that at every point the second partial derivative of T in the x direction must equal the negative of the second partial derivative of T in the y direction. We use a trick to measure how close to solving the problem we are. If

the LHS of the equation does not equal 0, then we know we are not quite at the right solution. Whatever is left over (i.e. the difference between what we got and 0.0, the correct answer) is called the residual. Now we have something we can integrate. If we integrate the point residual over our entire finite element we know approximately how inaccurate that element' s solution is. Better yet, the integration of the differential equations has left us with algebraic equations, which we do know how to solve.

OK, but how does all this fit together? Well, we need one more fundamental concept explained: the element. I keep saying that an element is an approximation of the answer over a finite region of the domain, but what do I mean? Basically, the element is just an equation for the answer in terms of nodes and independent variables. Nodes are points on the element where the answer is just one number. Lets move to an example.

The elliptic equation given above describes temperature distribution over a 2-D region with no heat generation or convection, etc. Let us derive the simplest rectangular 2-D element possible. First off, looking at the equation I notice that x and y are independent variables, leaving T (temperature) as the field variable for which I am trying to find a solution. So I must want an equation for T in terms of x and y and some nodes T[]. This is written:

- T = T(x,y,T[])

I' ll assume I' m going to apply this equation to a unit square. This makes the math easier and I can change that later if necessary (turns out it isn' t, we just use something called the Jacobian).

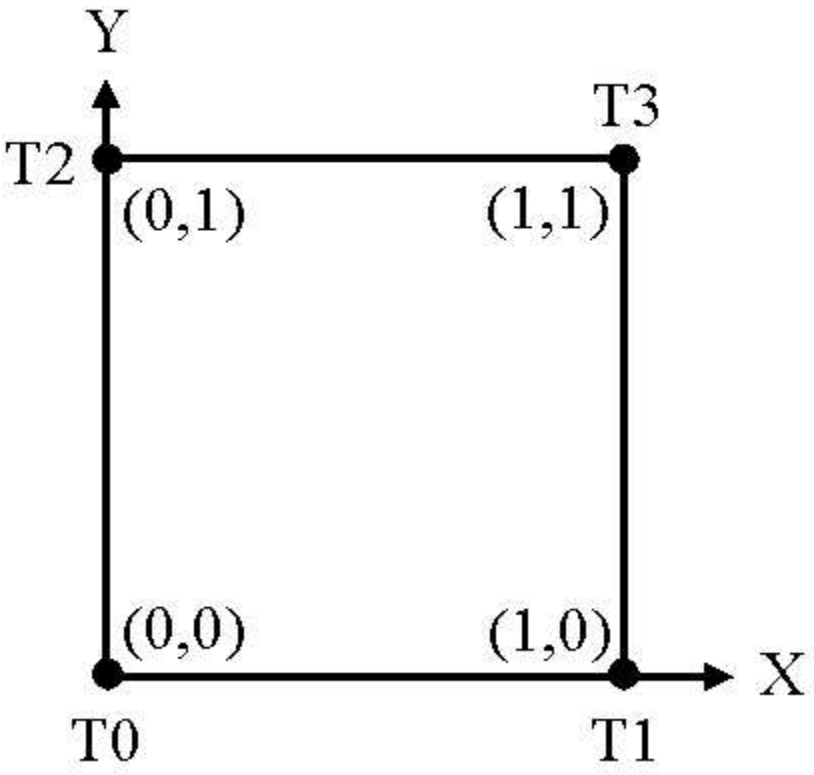


Illustration II.1 Node Locations for a Simple Element

So here we see that T is in fact:

- T = T(x,y,T0,T1,T2,T3)

T0 through T3 are the nodes I was talking about. I started numbering at 0 because I programmed my code in C++, where array indices always start at 0. There are some other nifty reasons too, which I cover in the section on corner numbering. Anyway, what this equation is telling us is that if I give you the values of T0 through T3, and an (x,y) location in that square, I can find the exact value of T that my finite element is approximating.

This is the simplest possible 2-D rectangular element you can get. Triangles

are another matter, and have their own problems to be dealt with. The "answer" at each of the four nodes is a single value, and it is these node values which we are solving for once all the equations have been derived.

I still haven' t explained how to arrive at the equation for T. That will be covered in depth later in the section on GATE elements and their derivation using the Boolean Sum. The derivation of these equations is one of the reasons people switch to finite difference or finite volume methods. It can be very difficult and tedious to do by hand, two partners which often lead to errors. That is why this paper documents a method for the automatic derivation of these. For now I will give the equation for the above example and proceed from there:

Eq. (2) $$T=(1-y)*(1-x)*T0+(1-y)*(x)*T1+(y)*(1-x)*T2+(y)*(x)*T3$$

Approximation for T

To verify the above equation, refer to illustration II.1 above and substitute in the different x and y values at each corner and make sure it turns out to be the node at that corner. Notice that the equation is grouped into terms dealing with each node. This will segue nicely into the next section.

4. Galerkin's Method of Weighted Residuals

Galerkin' s Method of Weighted Residuals was chosen because of its ease of application to my code, and because it is a very robust method. There are, as always, other ways to do things. We' ve already discussed residuals and that leaves the term weighted. Remember that residuals are just a way of monitoring

how close we have come to a solution for that element. If our residual is 0.0 then that means that the sum of all the point residuals totals to 0.0. That does not prove that there is no error within the element. It just means that our element' s approximation is as close as possible to the real answer as can be discerned without an actual analytical solution. The residual can always get worse, so a zero value indicates the solution is close to correct.

The equation for the residual of our element is (remember T() here is our approximation function, not the actual answer):

Eq. (3) $$R = \int_0^1 \int_0^1 \left(\frac{\partial^2 T()}{\partial x^2} + \frac{\partial^2 T()}{\partial y^2} \right) dx\, dy$$ *Elliptic Element Residual*

If, as is often the case, the element' s residual is not equal to 0.0, then we need to know how to change the node values so we can drive the residual to 0.0 for that element. But which nodes do we change, and in what way? This is where weighted residuals come into play. What we want to do is measure how much of the residual comes from each node. The way to do this is to find a function which describes how much weight a given node has to the value of the answer at any given location within the element. Thus each node gets a "weighting function".

Galerkin now makes his entrance by suggesting a very simple and elegant way to find these weighting functions. He proposes that we go into the original element equation and pull out all the terms which have a particular node in them. To demonstrate, the weighting functions as derived from our element would be:

Node T(i)	***Weighting Function W(i)***
T0	(1-x)*(1-y)
T1	(x)*(1-y)
T2	(1-x)*(y)
T3	(x)*(y)

As simple as that is for us, it is even easier for a computer, thus making it an ideal candidate for my thesis code.

Now that we have a weighting function for each node, there is no need to calculate the entire element' s residual. Instead, for each node we calculate its contribution to the residual. So we get an equation which looks like this:

Eq. (4) $$R(i)=\int_0^1\int_0^1 W(i)*(\frac{\partial^2 T()}{\partial x^2}+\frac{\partial^2 T()}{\partial y^2})\,dx\,dy$$ *Elliptic Node Residual*

The great news here is that by setting each R(i) = 0, for N nodes, we finally have N equations. The problem can finally be solved! Mostly. Note that for this example we need to take two second partial derivatives, but our approximation is only linear (no x^2 or y^2 terms). This means we will get zero information from our residual equations. This is only a temporary setback, however, because there are two things we can do at this stage.

First, we can use the Gauss-Green theorem to reduce the order of the equations. This is the route taken by most, if not all, finite element programs. This means we can actually use the linear approximation from this example, but it has the downside of actually attaching extra constraints to the equations. It says that the 1st partial derivatives are actually zero on element boundaries, and that is the same as assuming boundary conditions, because the original

differential equation said nothing about 1st derivatives. This can be bad, but isn' t necessarily. You just need to be aware of it. I use this method in the examples in this paper because it allows comparison to current methods, and allows studies to be done with the simplest elements possible (this is good for convergence studies).

A second method is to get the program to automatically derive 2nd order elements, so when the second derivative is taken we have a non-zero result (2nd order elements can have a 2nd derivative taken, but it yields a constant across the entire element. Therefore it is up to the weighting functions to force the resulting equations to be linearly independent). This second method is hardly ever used (I am unaware of it ever being used in commercial code). In current practice first order elements are preferred, and no elements of higher than second order are ever used. The examples in this paper demonstrate how easy it is to incorporate high order elements with G.A.T.E. formulation.

5. Putting it All Together

You know in theory how to apply differential equations to an element and get algebraic equations. But the finite element method usually requires many elements. How do you put them together? The answer to that is fairly easy, and has to do with how nodes are shared among neighboring elements. Look at the figure below.

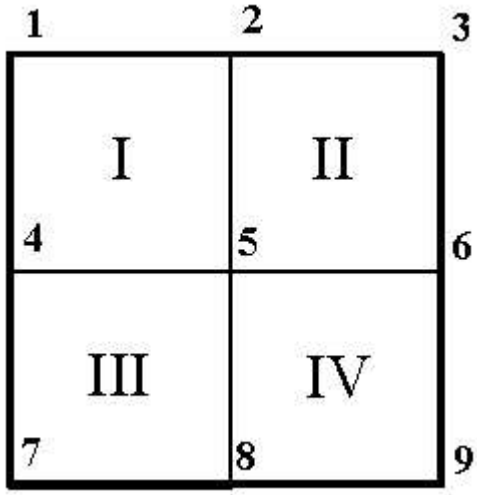


Illustration II.2 Shared Node Example Layout

You can see that node 2 is shared between element I and element II. So for each of those elements you get a residual equation for node 2. Add them together, and you have the total residual caused by node 2. This means that for the above figure you will have a total of 9 equations, because there are 9 nodes. Even though node 5 is shared by all four elements, there is only 1 total residual attributed to node 5. So for this example you can drop those equations into a linear solver, apply the proper boundary conditions, and quickly find the answer.

III. The Boolean Sum

1. Introduction

This chapter is the heart of my thesis, so I discuss it in more detail. The Boolean sum is the method I use to make Gradient Adaptive Transfinite Elements (G.A.T.E.). The concept behind it is quite simple. The difficulty lies in the actual implementation. Deriving these by hand to put into a pre-existing finite element package is hard, tedious, and can be extremely time consuming. In the next section I will describe the algorithm my code uses to derive all the elements needed for a given problem.

2. Basics

The Boolean sum is a method of finding a function over (in the easiest case) a n-unit domain, where n is the number of dimensions of the element. It takes the functions from every side of the element, and finds an interpolation function over the entire region. For example, a 2-D element is found by applying the Boolean sum to the four 1-D sides. A 3-D element is made by applying the Boolean sum to the six 2-D sides (ah yes, recursion). This will become clearer later on, but first I want to cover the derivation of an equation of interpolation for a 1-D element. That will become the basis for all higher dimensioned elements.

The equations which are typically used in numerical studies are polynomials. They are simple, well understood, easily linearized, and there are simple and direct ways of estimating the order of the error term. For all these reasons, and

more, I continue to use polynomials for my elements. There are some limitations inherent in this. Read the chapter entitled "Advantages and Limitations of the Method" for more information. The Boolean sum itself is not limited to polynomials, it is valid for all types of equations.

In the old linear algebra days, if you wanted to fit a polynomial exactly through m points, you needed a polynomial of order (m-1). So to fit a polynomial exactly through 3 points you would need a 2nd order polynomial, something like:

Eq. (5) $p(x)=ax^2+bx+c$ *Generic 2nd Order Polynomial*

because it has 3 unknowns. As you recall, solving for three unknowns requires three linearly independent equations. There are two basic ways to do this: you can specify the value of the polynomial at a point, or you can specify the value of some derivative of the polynomial at a point. The distinction is arbitrary, because the polynomial itself can be thought of as the 0th derivative of the polynomial. So I can find a polynomial by specifying:

Eq. (6) $p(0)=0,\ p'(0)=0,\ p(1)=1$ *Desired Conditions*

This yields the following set of simultaneous equations:

Eq. (7)
$$\begin{aligned} a*0^2+b*0+c&=0\\ 2*a*0+b&=0\\ a*1^2+b*1+c&=1 \end{aligned}$$
Resultant Simultaneous Equations

and then solving the 3 equations yields:

Eq. (8) $p(x)=x^2$ *Solved 2nd Order Polynomial*

Or I can find the exact same polynomial by specifying:

Eq. (9) $p(0)=0, P(0.5)=0.25, P(1)=1$ *Alternative Conditions*

Try it for yourself. The point here is that since my program will be solving for the best approximation to the differential equations, it will be finding the best polynomial of whatever order I specify. It does not care whether I actually specify the order using derivatives, or by specifying point values of the function. So it is easiest for me just to space out my nodes equidistant along any unit 1-D element. This means that if I want a 4^{th} order polynomial, then I will have 5 nodes, spaced 0.2 apart as the unit domain goes from 0.0 to 1.0. The results are the same, but the coding is easier.

The only place where the partial derivatives should explicitly be specified is at the boundaries, and then only if you need to enforce derivative boundary conditions. Derivative degrees of freedom can also be generated for all internal elements to force derivative continuity of a given order, if so desired. Although the Boolean sum can handle derivatives at the boundaries of elements, this ability to specify partial derivatives for a given set of nodes has yet to be implemented in my mesher. Because of this the examples in this paper do not show off this ability.

3. Enter the Boolean Sum

It is relatively simple to find a 1-D polynomial in terms of nodes spaced along its length. How can that be extended to n dimensions? The Boolean sum provides a very nice solution. First, we will work through a very nice 2-D example (remember, for 1-D I just use the method described above; my Boolean

sum subroutine is only called for 2-D elements or above). Then I will show the generalization of this into any number of dimensions. 5-D elements have been generated and verified.

The polynomial class code has a limit on the number of dimensions that can be used. This is because each added dimension requires more memory. Right now I allow any node of a polynomial to have up to 8 variables in it at one time. This is enough to represent a six dimensional non-linear element. If this limit is exceeded a warning is printed, alerting the user to the need to recompile the code with a larger limit.

First look at the following figure:

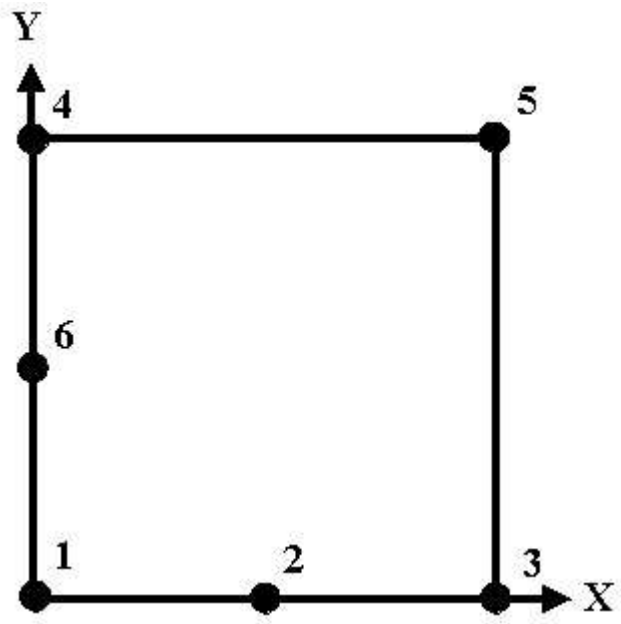


Illustration III.1 Boolean Sum Example Node Locations

This is an overhead view, showing the location and numbering of the nodes. For the following example I have values already assigned, so the visualization

can occur easier. This way I can show you pictures rather than only polynomials.

Node #	***1***	***2***	***3***	***4***	***5***	***6***
Value	0	1	0	1	1	0.25

The first thing to do is the simplest. We build "projections" along each of the axes. By the way, I will refer to the side functions as North, South, West, and East, where West is x=0, and South is y=0 (sorry, people from Down Under). So we have the following equations:

Eq. (10)

$$\begin{aligned} North(x) &= 1.0 \\ South(x) &= (x - x^2) * 4.0 \\ West(y) &= y^2 \\ East(y) &= y \end{aligned}$$

Edge Equations

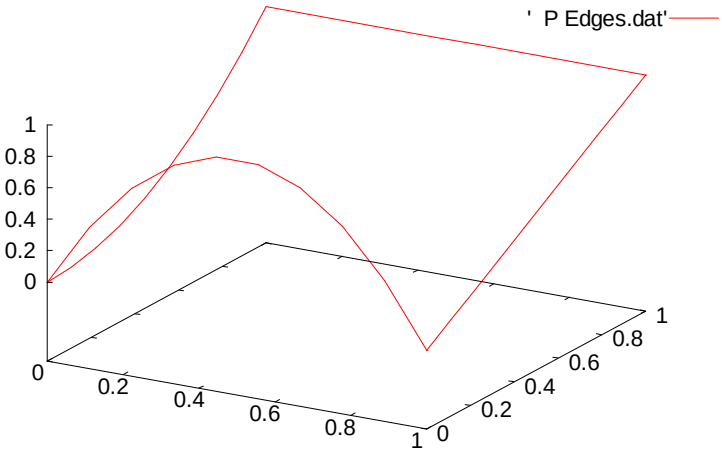


Illustration III.2 Exact Edges

The projections are done in the exact same way as the 1-D case I outlined above, except that my code only allows you to specify a value or a partial derivative at the boundaries of the element, and now the things we specify are

themselves functions. But all of that does not matter yet. So let us move on to the projection in the X direction (Px). We have only two things to fold into this projection: the West and East functions. West is at x=0 and East is at x=1. The way we handle this projection is the way you would do it for the 1-D case: set up equations as if W (for West, of course) and E were just point values. You would set up two equations: Px(0) = W and Px(1) = E. Then, since we have only 2 points, I say Px will be 1st order (linear), and solve for Px in terms of x, W, and E. It turns out that Px = W*(1-x) + E*(x).

This can be applied straightforwardly to the real situation. You can think of this in the two step, technically correct way, or in the 1 step hand waving way. Technically, we must extract the blending functions for each representative point value, i.e. W has a blending function of (1-x), and E has a blending function of (x). Now that we have extracted the blending functions, we apply those to the appropriate function in the original problem. We get:

Eq. (11) $$Px = West*(1-x) + East*(x)$$ *Projection in the x Direction*

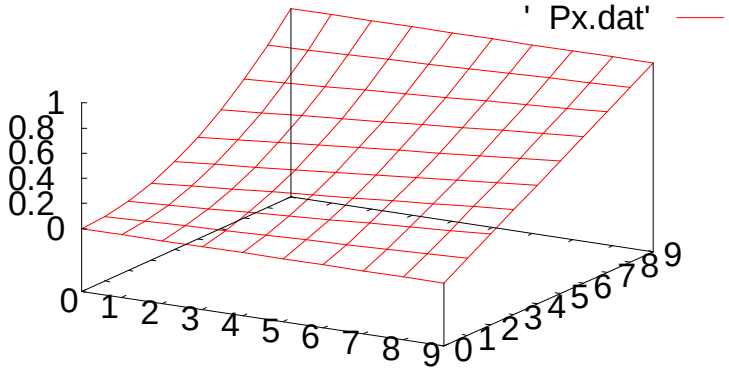


Illustration III.3 Projection in the x Direction

In the hand waving method you say W and E were shorthand for the real thing, so just spell them out, and you' re done. Either way, I now have the blending functions applied to my projection. We need to do the same for Py:

Eq. (12) $Py = South*(1-y)+North*(y)$ *Projection in the y Direction*

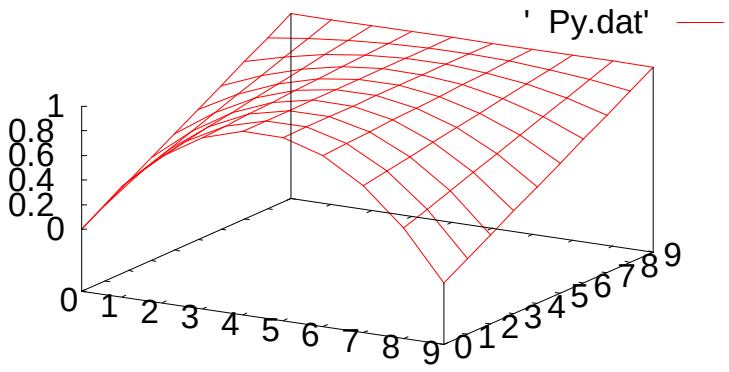


Illustration III.4 Projection in the y Direction

Now I have 1 projection for each axis. On the Px projection you will notice that is it correct on both the West and East sides. The North side also happens to be correct, but that is just because we use linear interpolation and the North side was linear (i.e. do not count on this always being the case!). The South

side is way off. Likewise for Py, both the North and South are correct, East is incidentally correct for the same reason as was mentioned above, but West is obviously wrong. It would be nice if I could just add them, and say that the sum was my final answer. Here is what you would get:

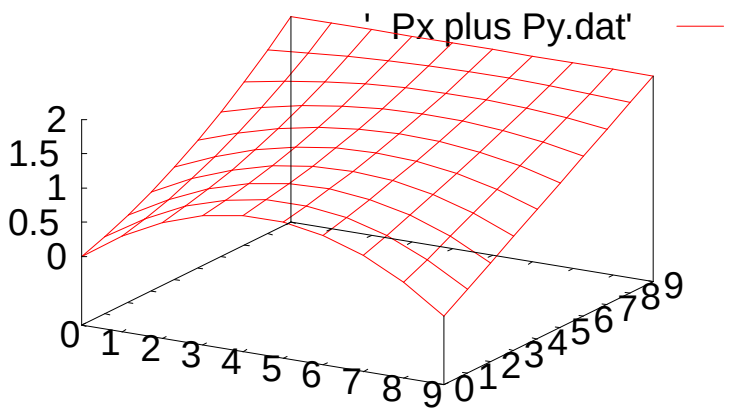


Illustration III.5 Sum of the x and y Projections

Things look better, except for the fact that this goes all the way from 0.0 to 2.0, and should only go to 1.0. Well, fortunately for us, the Boolean sum takes care of that. There is one final projection in 2-D, and it is named, ironically, Pxy. You take the first direction named (x in this case), and start with that projection. So, we are looking again at Px, and remember that the West and East sides are correct, but there are no guarantees for the North and South sides. What we are going to do, is take the spurious north (Px at y=1) and south (Px at y=0) (notice the lower case) from Px, and use the same projection we did for Py to get a correction factor.

So we had Py = South*(1-y) + North*(y), but substituting in the wrong north and south I get :

Eq. (13) $Pxy = south*(1-y) + north*(y)$ *Cross Projection*

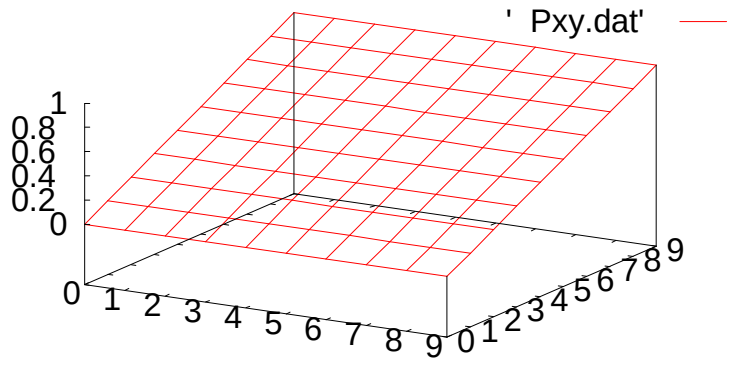


Illustration III.6 Cross Projection

Pxy is the correction factor: it takes into account the "assumptions" which are made about the uninvolved sides of each of the original projections, Px and Py. So our final answer is:

Eq. (14) $Pexact = Px + Py - Pxy$ *The Exact Polynomial*

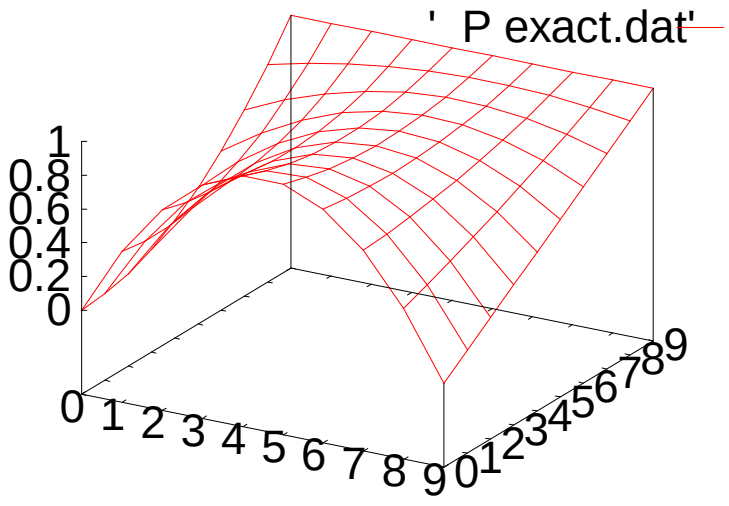


Illustration III.7 The Exact Polynomial

And it does look good. There are a few points I want to mention here before moving on to generalize to n-D. First off, the actual code to generate data points for these plots was simple because I had predefined the values of the nodes so each side was a polynomial I could type into my compiler. In the real case I want the polynomial in terms of the nodes, which are variables. The output I would be looking for was the 2-D polynomial itself, not calculated values. To do this for my thesis, I wrote a small class in C++ to handle polynomials of arbitrary size and dimensions. This class overloaded operators for multiplication, addition, subtraction, the derivative, and the definite integral (must have actual values for the bounds, typically 0 and 1). This in turn allowed me to write out the equations as I would for regular scalars, and the compiler handles all the math. Thus the resulting system of equations is pulled out intact, and can be applied as needed to the finite element problem at hand.

Another thing I would like to point out is that you just witnessed the birth of an element which is not used at all in the pre-packaged Finite Element world. The element we so casually derived is second order on two sides, and first order on the others. This is of great benefit as a transition element (think "Gradient Adaptive" from the G.A.T.E. acronym) between regions where complex things are going on and you want high order elements, and regions where nothing is happening and you cannot afford to waste compute cycles with high order elements. Prior to these elements, you had to make do with one type of element (i.e. all four sides were linear, or all four sides were 2nd order). If you switched between linear and 2nd order elements, information was always lost. Observe the illustration below:

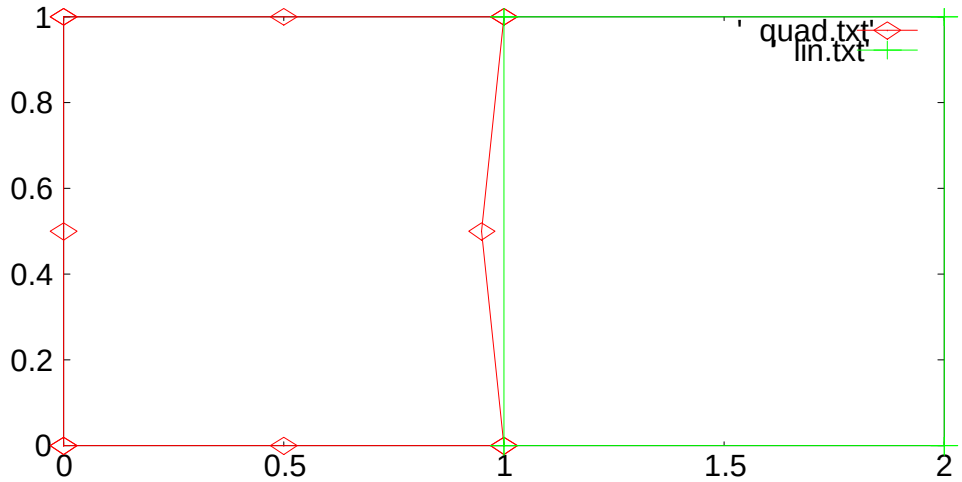


Illustration III.8 Information Loss due to Improper Element Coupling

The other property of the element we just derived, is that it exactly matches the functions we specified for the sides. This is the Transfinite property, which the Boolean sum always provides. As I mentioned before, it can handle any type

of function on the sides, but the math becomes really hard to do by hand, and it is ugly even on a computer (combining like terms from a polynomial is easy compared to trig identities). Sticking with polynomials is a good idea unless you come across behavior which absolutely cannot be modeled by them. An example would be shock waves in air, as they are discontinuous. You could improve your solution by using a small, high-order element, or you could rewrite the equations for that element: both approaches have problems associated with them. There are other methods as well, but further pursuit of this topic is not relevant to this paper.

4. Generalizing to n-D

We have seen that Pexact for the 2-D case had 3 terms and was:

Eq. (15) $Pexact = Px + Py - Pxy$ *The Exact Polynomial: 2-D*

For the 3-D case it has 7 terms and is:

Eq. (16) $Pexact = Px + Py + Pz - Pxy - Pyz - Pxz + Pxyz$ *The Exact Polynomial: 3-D*

The n-D case is simple, once you understand the fundamentals behind the 2-D and 3-D cases. In the following table you will note that on one side I am counting in binary, and on the other I have listed the terms from the Boolean Sum:

2-D : Binary (yx)	***Term from 2-D Boolean Sum***
"01"	Px
"10"	Py
"11"	-Pxy

And for the 3-D case:

3-D : Binary (zyx)	***Term from the 3-D Boolean Sum***
"001"	Px
"010"	Py
"011"	-Pxy
"100"	Pz
"101"	-Pxz
"110"	-Pyz
"111"	Pxyz

Some observations need to be made at this point. If the binary number counts from 1 to the highest number it can with n bits (n comes from the n-dimensional element), then we get all the permutations we need to calculate the Boolean sum. If the number of high bits (1) in a given binary number is odd, then the projection term is positive, otherwise it is negative. Pexact is just a summation of all the projection terms.

So the code would be something like this for a 4-D element:

Pexact = 0

count from 1 to 15 (binary "1111" is 15 decimal [2^n – 1])

Sign = -1.0

start from the least significant bit and go to the most significant bit

if the bit is High

Sign = -Sign

if it is the first bit to go high,

then find the projection for that axis

```
    otherwise
        find the projection of the previous projection for that axis
    finish counting bits
    Pexact = Pexact + Sign * Projection
finish counting in binary
```

Obviously a bit more goes on internally, but that is basically it.

IV. Binary Conventions

1. Introduction

This chapter provides a reference and explanation behind the binary conventions used in my code. As you saw in the last chapter, a binary representation was extremely useful in computing the Boolean sum. It is also useful in other areas, and after a brief introduction, the rest of the chapter will deal with them. The two areas I will discuss are the corner numbering system, and the default order in which the nodes are specified.

2. Basics

As my thesis relies on computer usage, the binary representation of numbers is a natural extension. It is fortuitous that this representation can be used in so many aspects of my thesis. Here I will cover the basics of the binary system, so please feel free to jump ahead if you don' t need the refresher.

"Boolean" algebra was invented by George Boole, and is a method of using algebra-like statements to prove logical problems. As computer logic problems end up with an answer of either True or False (two states), the binary representation is useful.

The binary system is a base two (2) notation, instead of base ten (10) like we currently use. In the binary system, each digit can either be a 0 or 1, and is called a bit in computer terms. In base 10 notation, each digit can go from 0 to 9, ten possibilities in all. In every number system the position of the digit is

important. In base ten, we say that the rightmost digit is in the "ones" place, the next place to the left of that is the "tens" place, then the "hundreds" place, and so on. The value of the digit with respect to the whole number is itself multiplied by the value of its place. The place value can be determined with a simple formula:

Eq. (17) $PlaceValue(i) = Base^{i}$ *Position Related Value in Arbitrary Numbering Systems*

where Base = 10 for decimal and Base = 2 for binary. "i" is the position index, and is equal to 0 at the rightmost place.

3. Corner Numbering

My thesis uses the Boolean sum to generate elements of any dimension. However, the catch is that I cannot derive triangular or tetrahedral elements, since my finite elements must have a total number of sides equal to twice the number of dimensions in that element. Each side is one dimension less than the element (which will later dovetail nicely into recursion). For example in two dimensions there are four 1-dimensional sides. Each 1-D side (a line) has two 0-D sides (points), and so on.

The practical upshot of all of this is that I can have a single simple numbering scheme for the corners of all of my elements, no matter their dimension. First consider that every dimension of my element is scaled between 0.0 and 1.0. This probably does not match the physical system, but I account for that later. This is referred to as "mapping to the unit domain" (so a 2-D element would be in

the bi-unit domain, etc.)

This means that every corner of my element must have a location of 0 or 1 for each axis. Thus for a 2-D element, the four corner positions can be represented (00), (01), (10), (11), where each coordinate pair is in the form (yx). The order is reversed from the normal convention because in binary the first bit is in fact the rightmost one. If you noticed that this progression is exactly the same as counting from 0 to 3 in binary, you now have my numbering scheme. So if I have a 3-D element, and I refer to corner #6, I transform 6(decimal) to 110(binary) and map it to my axes (zyx). Thus corner #6 is located at position x=0, y=1, z=1.

4. Default Node Specification Order

One of the reasons for using the Boolean Sum to generate finite elements is that any function can be specified as a side to the element. This allows for elements which change order from one side to another. This is useful for instances where changing order is desired, usually when one region requires a high order of accuracy while another does not. Without the ability to change the order of the elements, the modeler is left with an unfortunate choice between using computationally expensive elements everywhere, or numerically inadequate elements everywhere, or of ignoring nodes altogether to mesh high and low order elements together.

Unfortunately, because of the very malleability offered by the elements generated by the Boolean Sum, the specification used to build each element

must be similarly dynamic. The only information which can be treated as given is the number of sides on an element, and the ultimate number of 1-D elements used in the construction. It can be verified by inspection that the number of sides and 1-D elements is given by:

Eq. (18) $nS = N * 2$ *Number of Sides on an N-D Element*

Eq. (19) $n1D = N * (2^{N-1})$ *Number of 1-D Elements in an N-D Element*

I chose to build every element out of its component 1-D elements. In the example of a 3-D element (imagine a cube), each 2-D side can be built up from the 1-D elements along its sides. Once all of the 2-D square sides have been built, the Boolean Sum can be called to assemble the 3-D cube.

In order for my code to build each element it must know which nodes are placed along the 1-D sides of every element. This section details the order in which the nodes are specified, and is, in fact, a description of the input file format used by my code.

The first number in my element input file is the total number of elements. Then, for each of these elements I have the following strategy: first, read in an integer whose bits are set to indicate which dimensions are used by this element. The axes / bit pattern standard I am using is (*tzyx*). So if you wanted a 2-D element in *y* and *time* the Dimensions integer would be 1010 (binary) or 10 (decimal). My code counts the number of set bits, and determines how many 1-D elements to expect.

Next, my code reads in one integer for each 1-D element, which tells how many nodes are on that element. So for a 2-D element (time and y) where each

side is 2nd order, the file would look like this: (note: all values in decimal)

10

3 3 3 3

With this out of the way, it only remains for the code to read in the nodes each 1-D element uses. As the list is inclusive (the corner nodes are specified for each 1-D element), the redundant information can be used as a check to see that at least the corner nodes are consistent (i.e. The (00) corner should have the same node residing there, whether it was specified by the *y*=0 line or the *time*=0 line). All that remains to be known is the order of each of the 1-D elements and their associated node numbers, and here again we run into some binary representation trickery.

As I did with the corner numbering scheme, the idea is to use a fairly simple scheme to govern the input order of 1-D elements, no matter what dimension the final finite element is to be. The trick here is to first write down all of the dimensions in use by this element. Continuing with the previous example would be difficult, as it is hard to generalize from the 2-D case, so this element will have the dimensions (*tyx*). "*x*" is considered the 0th dimension, "*y*" is the 1st, and *time* is the 2nd. The fact that "*z*" is missing is transparent to the code, and is handled automatically when necessary.

The first thing to remember in this binary representation, is that I split the binary number (*tyx*) into 2 parts, the Travel variable, and the Location variable. In this case the Travel variable index is looped from 0 to 2, and the Travel variable itself is only 1 bit wide (so it can be only 0 or 1). The Location variable

is made up of all the rest of the bits (2, for this example), and is looped from 0 to 3.

So the method goes like this: first, set the Travel index to 0, so the Travel variable is "x". The Location variable is then relegated to being (ty). As Location loops from 0 (00 binary) to 3 (11 binary), the current 1-D element is defined as the one which goes from x=0 to x=1 while Location=(00). As Location increments to (01), we find that the second 1-D element goes from x=0 to x=1 at Location=(01). So each 1-D element is defined by the numbers of the corners between which it stretches! The example below will demonstrate.

1-D Element #	***From Corner # (tyx)***	***To Corner # (tyx)***
1	000	001
2	010	011
3	100	101
4	110	111
5	000	010
6	001	011
7	100	110
8	101	111
9	000	100
10	001	101
11	010	110
12	011	111

A table just like this one is generated automatically in memory. It would be fairly simple to incorporate a function which reads the 1-D element order from a

file, using paired corner ID numbers. However, I believe that this method is logical, and as simple as you can get. Unless you are importing prior mesh definitions, this format that should be used.

V. Variable Transformation

1. Introduction

As was mentioned before, it is easiest for us to derive and use our elements based on an n-unit object. But in real life all of our elements are not of unit dimensions, nor are they always perfect squares or cubes. What is needed is a method to correct our equations so they are talking about useful shaped elements without invalidating everything we have derived so far. Enter the Jacobian and |**J**|, the determinant of the Jacobian.

2. The Chain Rule in Action

The reason for this transformation can easily be shown with a simple 1-D example. The chain rule is used in the transformation, and you will see the chain rule again in the Gauss-Green section. There are two factors which need to be corrected: the improper scaling of derivatives, and the length / area / volume / etc. term in the integral (i.e. The "dx" or "dx dy" terms).

There are some situations where these are not needed. For example, when all of the elements are the same size and square, everything works out (as all of the equations are multiplied by 1.0 or 1.0^2, etc.). This is the case for many simple problems, including the first two sample problems given in this paper. These issues should not be ignored, however, when there are external forcing functions acting on the elements.

The actual residual that I am solving for looks like this (I will refer to the total

residual for a 1-D element, because it simplifies the math. This principle remains true even when multiplying by the weighting functions):

$$\int_{x0}^{x1} \frac{\partial f(x)}{\partial x} dx \quad \textit{Element Residual in Real Space} \qquad \textit{Eq. (20)}$$

$$\int_{0}^{1} \frac{\partial g(u)}{\partial u} du \quad \textit{Modified Element Residual in Computation Space} \qquad \textit{Eq. (21)}$$

As you can see, what I originally wanted to solve for has been slightly altered. I know that g(u) and f(x) look the same, including going to the correct values at the boundaries, but they do so over a different scale. It is that difference in scale which causes the problem in two different ways.

Imagine that f(x) goes from 0 to 100 linearly over the range x=0 to x=0.1. So the partial of f(x) with respect to x is 1000. Now g(u) also goes from 0 to 100 linearly, but now the range is u=0 to u=1. This time the partial derivative is 0.1 times smaller. Because of how we set up the finite elements, the following is true:

$$g(u) = f(x(u)) \quad \textit{Change of Variables} \qquad \textit{Eq. (22)}$$

Now, because of the chain rule we can see:

$$\frac{d\,g(u)}{du} = \frac{d\,f(x(u))}{du} = \frac{d\,f(x(u))}{dx}\frac{dx}{du} \quad \textit{Chain Rule In Operation} \qquad \textit{Eq. (23)}$$

This equations shows that we are a factor of dx/du smaller, which of course turns out to be the constant 0.1. That error is fairly easily corrected with the inverse of the Jacobian.

The second case of the scaling problem is in the integral of the dx or du term. If we ignore the actual function inside the integral, and just assume it is one, you

can see the problem. The integral of x from x0 to x1 is (x1-x0), whereas the integral of du is always going to be 1.0. This problem is corrected with the determinant of the Jacobian.

3. The Jacobian

The Jacobian is an N x N matrix, where N is the number of dimensions in my equations. For the sake of the following examples, I will use x, y, z, and no time as my dimensions. Thus the example Jacobian, **J**, is a 3x3 matrix. The dimensions x, y, and z, are real dimensions. By which I mean they are the dimensions which physically describe the shape of the element in real space. I also have my computation dimensions: u, v, and w. These are the dimensions which have always been in the range 0-1, which I use for simplicity in derivation and integration of my elements. The Jacobian is defined as follows:

Eq. (24) $$J = \begin{bmatrix} \frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} & \frac{\partial z}{\partial u} \\ \frac{\partial x}{\partial v} & \frac{\partial y}{\partial v} & \frac{\partial z}{\partial v} \\ \frac{\partial x}{\partial w} & \frac{\partial y}{\partial w} & \frac{\partial z}{\partial w} \end{bmatrix}$$ *The Jacobian*

This setup is well suited to my code. Each of the functions for x, y, and z in terms of u, v, and w are easily built using the Boolean sum. The actual x, y, and z values of control points are substituted for the matching node numbers in the recently created polynomial. Thus I can take exact derivatives for each of the functions and build an exact Jacobian. This makes for one less source of error.

4. Inverse of the Jacobian

I have not had to use the inverse of the Jacobian in its full form. For the simple geometries of the sample problems presented in this paper, there is an easy way to deal with this. The actual full implementation of this is shown below in matrix notation. f is the function in real-space (x, y, z basis) and g is in element-space (u, v, w). The 3-D function in matrix notation is:

Eq. (25)

$$\begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} = \boldsymbol{J}^{-1} * \begin{bmatrix} \frac{\partial g}{\partial u} \\ \frac{\partial g}{\partial v} \\ \frac{\partial g}{\partial w} \end{bmatrix}$$

General Conversion of Variables in a 1st Derivative

This looks fairly easy, except that the matrix **J** is made up of polynomials, and not easily invertible. It can be done, up to a point, by Cramer' s Rule for example, but there is no simple way of computing analytically a polynomial divided by another polynomial. I might implement a polynomial fraction class which would have a numerator and denominator polynomial, though there would be no simple way to divide out common terms.

To take a 2nd derivative requires the use of the chain rule. You begin by taking the 1st derivative, then take another 1st derivative of that. Each derivative occasions a use of the inverse of the Jacobian. In my code the derivative subroutine does not reference the Jacobian at all, because it is simply a polynomial class. There are many instances where I need to take a derivative

and have no wish to involve the Jacobian (for example when I am building the actual Jacobian). Thus care must be taken to apply this transformation at the correct point in the process.

Although this step is based on some fundamental mathematics, it seems to be universally ignored in commercial finite element packages. There is a rule of thumb that elements with an aspect ratio of more than 2.5 not be used. This is because without this inverse Jacobian correction, non-square elements do not put as much of a penalty on errors in the short direction as they should. An extremely simple hack would be to find a bounding box about each element, and use the box' s dimensions as the diagonal for a simplified inverse of the Jacobian. This would at least minimize the damage from high aspect ratio elements.

If I had to implement a general inverse of the Jacobian fix right now, I would probably use something similar to what I describe in the Manifolds section. For my current problems, however, there is an easier way. Since my current example elements are perfectly rectangular, there are no off-diagonal terms in the Jacobian, so it is trivial to compute the inverse (delta variables are dimensions of the elements):

Eq. (26) $$\boldsymbol{J}^{-1} = \begin{bmatrix} \frac{1}{\Delta x} & 0 & 0 \\ 0 & \frac{1}{\Delta y} & 0 \\ 0 & 0 & \frac{1}{\Delta z} \end{bmatrix}$$ *Exact Inverse Jacobian for a 3-D Orthonormal Element*

5. Determinant of the Jacobian

The determinant of the Jacobian, |**J**|, is actually a correction factor for volume (in this case; it could be area or length for 2-D or 1-D elements respectively). To see this with a simple example, assume the following functions:

$$\begin{aligned} x &= 5*u+6 \\ y &= 3*v+2 \\ z &= 2*w-4 \end{aligned}$$

Eq. (27) *Sample Variable Transformation in 3-D*

Building the Jacobian, you see that the offsets do not matter, only the scaling factor. What remains is the following matrix:

$$\boldsymbol{J} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Eq. (28) *Sample Jacobian from the Sample Variable Transformation*

with 5, 3, 2 on the main diagonal, and zeros everywhere else. The determinant of this matrix is simply a product of the entries along the main diagonal, or the ratio of the volume of the element in real space (30.0) to the volume of the element in computation space (1.0).

I am already calculating a point residual for integration over the entire element, and it is multiplied by what I called "dx*dy*dz". However, because of the way I always built my polynomials on 0 or 1 straight boundaries, I am actually using "du*dv*dw", which is a differential volume. To correct my differential volume so that is accurately reflects how much volume my point residual covers in real space, I need to multiply by |**J**|. Thus:

$$dx*dy*dz = |J|*du*dv*dw$$

Eq. (29) *Volume Correction Term*

Now I have simply corrected my point residual so it is dealing with real world

elements instead of my unit domain elements. Of course this only corrects for volume changes, it does not correct for angles. Thus if the x axis, y axis, and z axis are not mutually perpendicular, then some error may result, depending on the severity of the distortion. Most, if not all, differential equations are derived assuming that the axes are orthonormal. If any element becomes too skewed, the solution will fail (it will give incorrect answers, without necessarily giving any indication that they are wrong) because this assumption is invalidated. The same is true in commercial codes.

6. Manifolds

Manifolds require a special version of the determinant of the Jacobian. If we are solving in 3-D space, but we want to insert a 2-D or 1-D element, what happens? Take the case of a 2-D element in 3-D space. Imagine it looks like a flag, blown in the wind. It has x, y and z components, but any point on that flag can still be described by two variables. Thus x, y, and z are all functions of u and v, not w. The Jacobian' s entire bottom row is filled with zeros, so $|\mathbf{J}|$ must be equal to 0.0. And this is entirely correct, because the *volume* of my differential area, du*dv, is in fact zero. Correct or not, this means we cannot write linearly independent equations for this element because they are all multiplied by zero. The solution?

It turns out that $|\mathbf{J}|$ is the perfect correction factor if both our real space and computation space have the same number of dimensions. In the case just

outlined, you need to use the more global definition. This states that the correction factor (which I will continue to call |**J**| for ease of use) is the square root of the sum of the squares of all the useful determinants. By useful determinants, I mean the determinants of the largest possible square matrices left in the Jacobian.

For our flag example, this means that the largest possible square matrix we can have is a 2x2. So there are 3 useful determinants, det (columns 1,2), det (columns 2,3), and det (columns 1,3). So the correction factor is the square root of the sum of the squares of these.

If we take a much simpler example, that of a parametric 1-D element, we can see how this correction factor reduces to the old "length of a parametric line" equation from beginning calculus. We have three equations again, x, y, and z as a function of u. Thus we have only the top row of the Jacobian filled with non-zeros. According to our manifold equation the differential length dl is:

Eq. (30) $$dl = \sqrt{\left(\frac{\partial x}{\partial u}\right)^2 + \left(\frac{\partial y}{\partial u}\right)^2 + \left(\frac{\partial z}{\partial u}\right)^2}$$ *Differential Length of a Parametric Line*

Remember that each determinant is a polynomial (in the simplest case it can be a constant, but generally it is not) and it is relatively easy to square a polynomial, and very easy to sum them. The impossible part is taking the square-root of a polynomial. To fake the accomplishment of this, in the case of a manifold element I call a special subroutine to generate my |**J**|. This subroutine finds all of the useful determinants and then squares and sums them, storing this in J_Squared. At this point I have a polynomial which holds the exact value of

the square of the correction factor (|**J**|) at every point within my element. To get from there to my |**J**|, I make a template polynomial of the same type as J_Squared. I leave the nodes as variables at first. Then I systematically substitute for each node variable the square root of J_Squared evaluated at the exact location of that node. My resulting |**J**| is now only an approximation, but it can be integrated exactly, and it is quite close.

Some trials were made to quantify the value of this procedure. I had my code integrating the area of a perfect circle as it was rotated in 3-D space using this method. The results were never worse than 12%, and usually were significantly better. For perfectly rectangular elements there was no error at all. Linear elements in 3-D space were extremely accurate. This approximation allows the inclusion of any dimensional elements into the solution.

VI. Gauss-Green Reduction of Order

1. Introduction

In my programs I use the "weak" form of any parts of the equation which require more than a first partial derivative. To show how this is accomplished, we look at the Laplace equation. The "strong" form is when you plug in the Laplace equation as is into our error term:

Eq. (31) $$Residual = \iint Shape(x,y) * \left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2}\right) dxdy$$ *Residual of the Elliptic Equation, Revisited*

This is easy to do, as long as the elements are higher than first order. A second order element yields a constant for its second derivative, and that can give linearly independent equations because of each different "Shape" function. Unfortunately, to solve an elliptic problem requires boundary conditions all around the domain. While the G.A.T.E. method allows you to specify a derivative condition in the making of an element, my code does not yet implement that. This means I am unable to specify all of the required boundary conditions for a problem which requires derivative boundary conditions. Because I cannot duplicate the boundary conditions, I cannot solve the problem.

2. The Chain Rule Revisited

The alternative is to use the Gauss-Green theorem to reduce the order of the equation (making it "weak"). Gauss-Green is like applying the chain rule in

reverse:

$$d(AB)=(dA)B+A(dB) \quad \textit{Eq. (32)} \quad \textit{The Chain Rule}$$

We have a case like:

$$Shape*(\frac{\partial^2\phi}{\partial x^2}) \quad \textit{Eq. (33)} \quad \textit{The Residual Term}$$

and we want to re-write it somehow, so we set:

$$A=\frac{\partial\phi}{\partial x}, B=Shape \quad \textit{Eq. (34)} \quad \textit{Substitution Assignments}$$

plug these into the chain rule and we see:

$$\frac{\partial(\frac{\partial\phi}{\partial x}*Shape)}{\partial x}=\frac{\partial^2\phi}{\partial x^2}*Shape+\frac{\partial\phi}{\partial x}*\frac{\partial Shape}{\partial x} \quad \textit{Eq. (35)} \quad \textit{Resultant Gauss-Green Reduction}$$

The first term after the equal sign is the one we want. We set the left hand side to 0, which basically sets the first partial of Phi to zero because Shape is always changing. It does not affect the elements inside the domain, because the LHS term would cancel out between elements (e.g. positive for node 3 on the left, negative for node 3 on the right). We move the second term after the equal sign over to the left, and now we have reduced a second partial to two first partials.

$$\frac{\partial^2\phi}{\partial x^2}*Shape=-\frac{\partial\phi}{\partial x}*\frac{\partial Shape}{\partial x}+0 \quad \textit{Eq. (36)} \quad \textit{Gauss-Green Reduction of a 2}^{nd}\textit{ Derivative}$$

Note that this last step occasions a change of sign!

The nice thing about applying the Gauss-Green theorem to the elliptic equation is that even when I don' t specify a boundary condition, one is already imposed because of my ignoring the left hand side. This means proper

boundary conditions for some test problems. It also means I can use linear elements (which is the main reason people use reduction of order) even if a second partial derivative is required. I get a constant out of the first partial of the element, and it is multiplied by the first partial of the Shape function, giving me linearly independent equations for each element.

The bad news is exactly the same as the first part of the good news. I had to impose an arbitrary condition on all of my boundary elements (where the LHS term did not cancel itself out), so I' m not really solving the original equation on the boundaries. For some finite element problems it does not matter too much because I wanted those exact boundary conditions.

VII. Using the Method

1. Problem Solution Cycle

There are a variety of steps involved in actually solving a problem with this method. This chapter details the steps and tools involved in going from problem to solution.

Right now there are three parts: the solver, the mesher, and the code which takes the meshes and generates the sets of equations for the solver to use. The code is refered to as the "GATE Equation Generator". That is what we have been talking about up to this point. However it must be used in conjunction with the solver and mesher to get any results at all.

Right now the differential equations must be hard coded into the GATE Equation Generator. That will change, but for right now it is not hard to do. It merely requires practice and a C++ compiler. For example, I can take my base code and modify it to generate the Navier-Stokes equations discussed later in this paper in under 30 minutes.

The next page shows a flowchart detailing how each tool is used.

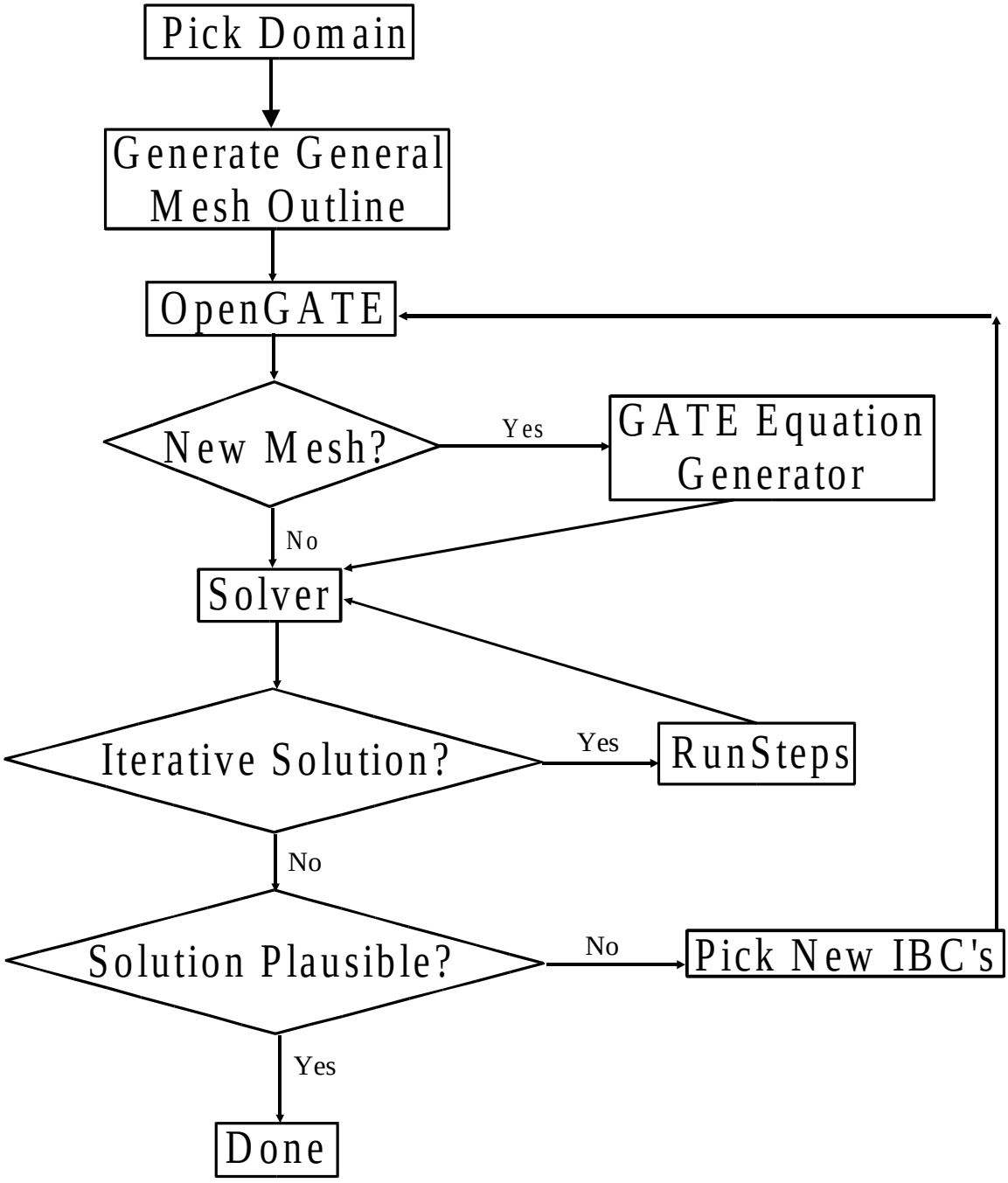


Illustration VII.1 General Problem Solution Flowchart

2. Tools

Here is a quick explanation of the tools used in this method.

a. The Nonlinear Solver

So far all my code has been directed toward getting a series of equations, linear or not, which properly describe the problem. Solving them is necessary to complete the model analysis. The types of equations to be solved can be classified into three groups: linear, nonlinear, and eigenvalue (the last of which is used to find things like natural response modes and frequencies). I have only dealt with the first two, and they are briefly described here. I often use MATLAB (or Octave, its freeware counterpart) to solve the linear or eigenvalue problems, but a solver was needed which could handle the equations as generated by my finite element code.

For the linear solver, I use Gaussian elimination to get the matrix into upper triangular form, then back substitution to find the answers. This technique is well documented in many locations (try http://www.nr.com for the Numerical Recipes online book in FORTRAN, C, or C++).

For my non-linear solver I use the Newton-Raphson method. This method uses the Jacobian matrix (see previous section) to linearize the problem at each step. The linear equations can be written in matrix form:

Eq. (37) $\boldsymbol{J}(x) * \delta x = F(x)$ *the Newton-Raphson Linearizing Equation*

So for each step, both the Jacobian and the residual vector are calculated, and I solve for delta x. To make this scheme slightly more robust, I calculate the

residual (sum of the squares of the evaluated system of equations, F(x)) at x+delta x. If the residual is higher after adding delta x than it was a x, I divide the delta x vector by 2 and try again. Once my new residual is smaller than the old, I permanently update x by adding delta x, and start all over again. This continues until the residual is below a preset tolerance (currently 1e-10), then the answers are written to a file and the solver exits.

In the case where I need iteration in time, a wrapping program called RunSteps is used to call the solver repeatedly and update the initial and boundary condition file with the results from the last time step solved.

b. OpenGATE: The Mesher

In order to actually solve any problems other than trivial ones, it was necessary to make a meshing program. This is especially true when attempting to do a convergence study. The input file required by the G.A.T.E equation generator code is in plain text, but generating one by hand is both tedious and prone to mistakes. Thus a meshing program was required which would take a simpler input file (also plain text), and generate the required (now) intermediate file(s) automatically.

I could not use any of the freeware meshing products to be found on the internet. For one thing many of them output only triangles, which my code does not use. And none of them have the capability of specifying different orders along different edges of a given element. Thus I had to define an input file format for a simple mesher, and then program one.

First I made some assumptions, which were then coded into the mesher. For one thing, everything is done in 2-D (it can be automatically extended into time, if desired). 1-D element input files are easily done by hand, though future generations of the mesher will hopefully support both 1-D and 3-D as well. Also, the auto extension into time can be of any order, but is only ever one element deep. This means that a time solution requires iteration. Finally, there is no way to choose independent polynomial orders for the x and y directions.

I would like to give a simple description of the OpenGATE input file format. After that, a sample input file is shown below, and immediately following that is a screen capture of OpenGATE. The order of the first five lines of the file is important. The description following the number is merely for the benefit of whoever is modifying the input file: nothing is taken but the number.

The first two numbers specify the number of macro blocks in the x and y directions, which makes up the grid. These macro blocks can be active or not, and the resulting grid roughly defines the domain of the problem I want solved. The next section of the input file, following the keyword "grid", specifies which macro blocks are active. And asterisk (*) signifies an active block, and any other character makes it inactive. I tend to use a minus sign (-) because it makes the spacing obvious.

The next two numbers (lines 3 and 4) tell the mesher how to partition each macro block. Each block must be partitioned at least 1 by 1, otherwise no elements will be generated. The combination of active blocks with the partitioning information actually determines how many elements are generated.

The last number (line 5) governs extension of the whole mesh into time. In all the problems I have solved, the time dimension is always of a constant order, and all elements are supposed to move throughout time evenly. I wanted the mesher to make it as easy to extend the mesh into time as possible. I refer to this number (still the one on line 5) as the "Time Order". If it is zero, no extra information is generated in the time dimension, and what you see is what you get (in terms of a 2-D mesh). Any number higher than 0 is interpreted as the desired order of the finite elements in the time dimension. For these cases, basically a copy of the 2-D mesh is located at time = 0 and another at time = 1. If the time order specified is higher than 1 then individual nodes are added along the corners of the (now) 3-D elements in the time direction.

After this the mesher searches for the keyword "grid" and then proceeds to read in the grid. Next, it searches for the keyword "mesh". If the word is found (and the mesher should find at least one), it begins reading in a series of numbers, which I will explain in a minute. The text following the word "mesh" on the same line determines the output filename for this mesh. For example, the line "mesh First Field Variable" would generate a file named "First Field Variable.msh" with the specified elements. As many meshes as desired are generated, each with sequentially numbered nodes, allowing for coupled problems.

The numbers which actually follow the mesh line describe the desired order of the elements at each vertex of a macro block. This is why the matrix is both wider and deeper by 1 than the grid matrix (because a 2x2 grid has 3x3 corners).

For this version I limit the maximum order to 5^{th} (and the minimum order to 1^{st}), which yields at most 6 nodes on a 1-D element (or side of an n-D element). The number of nodes for any given side of an element is calculated by doing a weighted average of the values at the corners of the macro block in which it is housed. The weighting is based simply on position. This allows a convenient way of specifying a changing element order across a domain.

The final function of the OpenGATE program is to generate the initial and boundary conditions for the problem. This is accomplished with the help of the mouse and the three fields: "IC", "RND", and "BC". All boundary conditions (BC' s) are applied with the left mouse button, and they are applied to sides, corners, or whole macro blocks. This makes it easy to specify the same boundary conditions, even when you want to do "h" (add more elements) or "p" (increase the order of the elements) type mesh refinements. The solver and equation generation steps are separate, so the same problem can be rerun quickly with different boundary conditions, but without recalculating the equations.

A left click applies whatever number is in the "BC" field to the applicable nodes. The selected nodes turn dark blue. Previously selected nodes will remain with their original boundary condition. A right mouse click erases the last boundary condition. When the "Done" button is clicked, the mesh is generated, the initial and boundary condition ("IBC") file is augmented, and so is the geometry file (used later in deforming the mesh if desired).

All nodes not found to have been assigned a boundary condition with the

mouse will be assigned the initial condition from the "IC" field, give or take a random number retrieved from the "RND" field. The nodes without boundary conditions are subject to change, but for some solvers, including mine, they must begin with an initial condition. For linear steady state problems, although the initial condition does not matter, it is generated anyway.

The following is a sample input file and screen shot.

"Demo.txt"

4 blocks_x
3 blocks_y
2 div_x
3 div_y
0 time_order

grid
****-***
--
****

mesh T elements
11111
12321
13531
22222

mesh Pos elements
11111
22222
33333
44444

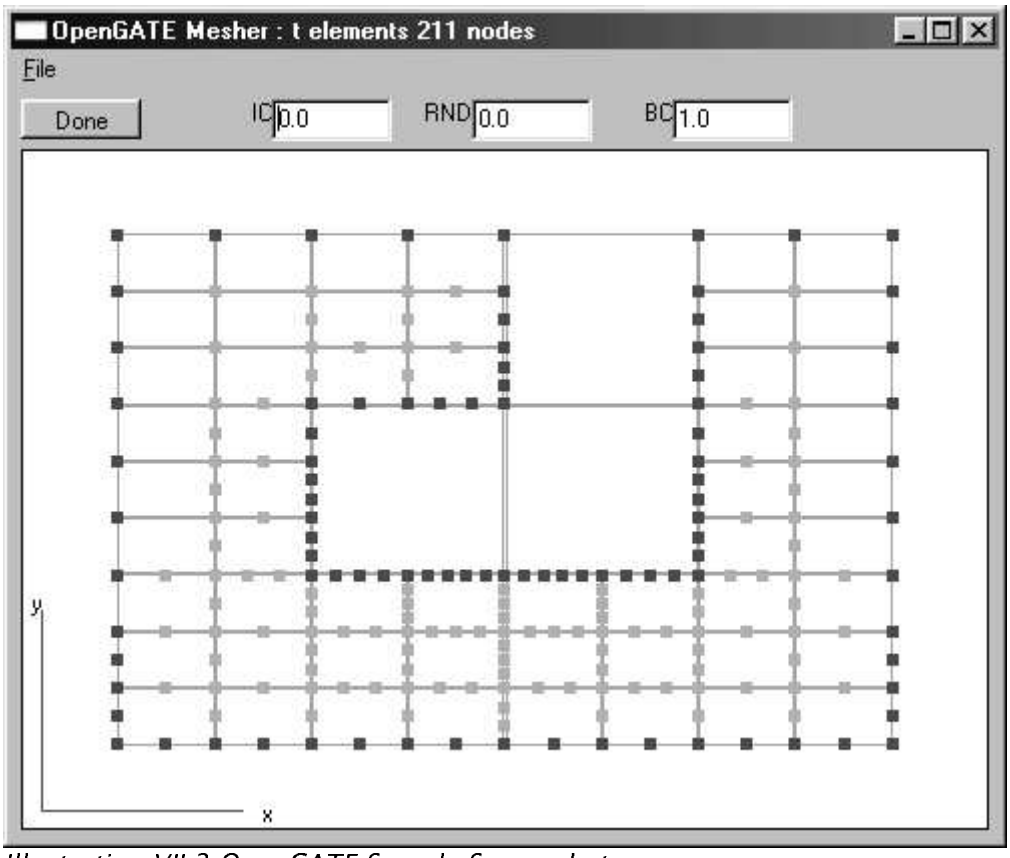


Illustration VII.2 OpenGATE Sample Screenshot

VIII. Usage: Code Example

1. Introduction

The following is almost exactly the code used to generate the equations for the fluid flow example. This snippet comes from the heart of the program where the actual equations are generated. Many of the subroutines which are necessary for this to run are not listed here . The language is C++, and it was compiled and run on many different Win32 platforms. The compiler was the mingw port of the GCC (GNU Compiler Collection: it includes C, C++ and FORTRAN, among others). This compiler was used because of its freeware status and in the hope of maintaining compatibility with the majority of systems (many OS' s or platforms have GCC ports, even the PlayStation 2).

2. Code

```
//     This is needed to tell my functions I have no data stored in them
Init_Element_Data (u_El);
Init_Element_Data (v_El);
Init_Element_Data (p_El);

//  In metric, of course, from my test case
Rho = 1.0;    //  kg/m3
Mu = 0.01;
//  And element scaling
idx = 4.0;
idy = 4.0;
idt = 150.0;

Initial_Time = clock ();

//  OK, let's build up the equations
for (count = 0; count < Num_Elements; ++count)
{
      //    Inform the user of the current status
      cout << endl << "Reading element #" << (count + 1) << " of " << Num_Elements << "." << endl;

      //    Read in the data from the geometry file
```

```
Read_Element (u_File, u_El);
Read_Element (v_File, v_El);
Read_Element (p_File, p_El);

//  And construct each element's polynomial
u_Poly = Build_Partial_Element (u_El);
v_Poly = Build_Partial_Element (v_El);
p_Poly = Build_Partial_Element (p_El);

//  For now do the easy way (this is OK if all elements are the same size.)
Det_Jacobian = 1.0;

//  Optimization
ud0 = u_Poly.Dif (0) * idx;
ud0.Compact ();
ud1 = u_Poly.Dif (1) * idy;
ud1.Compact ();
ud2 = u_Poly.Dif (2) * idt;
ud2.Compact ();
vd0 = v_Poly.Dif (0) * idx;
vd0.Compact ();
vd1 = v_Poly.Dif (1) * idy;
vd1.Compact ();
vd2 = v_Poly.Dif (2) * idt;
vd2.Compact ();

cout << endl << "Eq #1 : ";

//  This is to solve for "u" and "v"
Blending_Matrix = Extract_Blend (u_Poly);
Extract_Blend (v_Poly, Blending_Matrix);

Temp_Fun = ud0 + vd1;
Temp_Fun.Compact ();
for (i = 0; i < Blending_Matrix.Rows; i++)
{
   Temp = (int)Blending_Matrix.Matrix [i * 2].Evaluate (0, NULL) - Var_Cutoff;
   Local_Blend = Blending_Matrix.Matrix [1 + (i * 2)];
   cout << Temp << " ";

   Huh = Local_Blend * Temp_Fun;

   Huh = Huh.Int (0, 0.0, 1.0);
   Huh = Huh.Int (1, 0.0, 1.0);
   Huh = Huh.Int (2, 0.0, 1.0);
   Huh.Compact ();

   Right_Hand_Side [Temp] += Huh;
   Right_Hand_Side [Temp].Compact ();
}

cout << endl << "Eq #2 : ";

//  This is to solve for "u" and "v" and "p"
//  So add "p"
Extract_Blend (p_Poly, Blending_Matrix);

//  This part of the equation does not need Gauss-Green,
//  and can be computed outside the loop
```

```
Temp_Fun = ud2;
Temp_Fun += ud0 * u_Poly;
Temp_Fun += ud1 * v_Poly;
Temp_Fun *= Rho;
Temp_Fun += (p_Poly.Dif (0) * idx);
Temp_Fun.Compact ();
t01 = clock ();
for (i = 0; i < Blending_Matrix.Rows; i++)
{
   Temp = (int)Blending_Matrix.Matrix [i * 2].Evaluate (0, NULL) - Var_Cutoff;
   Local_Blend = Blending_Matrix.Matrix [1 + (i * 2)];
   cout << Temp << " ";

   // Now do the brute work of integrating the residual equations
   Huh = Local_Blend * Temp_Fun;   // The 1st part

   // And now the 2nd derivatives
   Huh2 = ud0 * (Local_Blend.Dif (0) * (idx * Mu));
   Huh2 += ud1 * (Local_Blend.Dif (1) * (idy * Mu));

   // This is a "+=" because the Gauss-Green changes the sign
   Huh += Huh2;

   // And integrate to get the final residual equation
   Huh = Huh.Int (0, 0.0, 1.0);
   Huh = Huh.Int (1, 0.0, 1.0);
   Huh = Huh.Int (2, 0.0, 1.0);

   Huh.Compact ();
   Right_Hand_Side [Temp] += Huh;
   Right_Hand_Side [Temp].Compact ();

   // Report time spent to the user
   t02 = clock ();
   cout << (t02-t01)* 0.001 << "[s] ";
   t01 = t02;
}

cout << endl << "Eq #3 : ";

// This is to solve for "u" and "v" and "p"
// Same as the last

// This part of the equation does not need Gauss-Green,
// and can be computed outside the loop
Temp_Fun = vd2;
Temp_Fun += vd0 * u_Poly;
Temp_Fun += vd1 * v_Poly;
Temp_Fun *= Rho;
Temp_Fun += (p_Poly.Dif (1) * idy);
Temp_Fun.Compact ();
t01 = clock ();
for (i = 0; i < Blending_Matrix.Rows; i++)
{
   Temp = (int)Blending_Matrix.Matrix [i * 2].Evaluate (0, NULL) - Var_Cutoff;
   Local_Blend = Blending_Matrix.Matrix [1 + (i * 2)];
   cout << Temp << " ";

   // Now do the brute work of integrating the residual equations
```

```
        Huh = Local_Blend * Temp_Fun;   // The 1st part

        // And now the 2nd derivatives
        Huh2 = vd0 * (Local_Blend.Dif (0) * (idx * Mu));
        Huh2 += vd1 * (Local_Blend.Dif (1) * (idy * Mu));

        // This is a "+=" because the Gauss-Green changes the sign
        Huh += Huh2;

        // And integrate to get the final residual equation
        Huh = Huh.Int (0, 0.0, 1.0);
        Huh = Huh.Int (1, 0.0, 1.0);
        Huh = Huh.Int (2, 0.0, 1.0);

        Huh.Compact ();
        Right_Hand_Side [Temp] += Huh;
        Right_Hand_Side [Temp].Compact ();

        // Report time spent to the user
        t02 = clock ();
        cout << (t02-t01)*0.001 << "[s] ";
        t01 = t02;
    }

    //Clean_Element_Data (Pos_El);
    Clean_Element_Data (u_El);
    Clean_Element_Data (v_El);
    Clean_Element_Data (p_El);

    cout << endl;

    Current_Time = clock ();
    cout << (Current_Time - Initial_Time) * 0.001 << " seconds have elapsed." << endl;
}

cout << endl << "Done assembling equations." << endl;
```

IX. Proofs of Concept

1. Introduction

In the usual finite element application, pre-computed matrices are stored for each type of element. They are modified trivially to account for whatever properties the element must represent, and then they are assembled into a global "stiffness" matrix. This method is very fast, and it does allow for new elements to be defined and used. On the down side, the differential equations must be hard coded into the matrix, the method is implicitly linear, and it is relatively hard to simulate varying field properties (density, stiffness, etc.).

One of the major strengths of the G.A.T.E. method is the ability to implement any needed polynomial order on each side of an element. While there are obvious benefits to be gained (and I hope to demonstrate some of them) in using the G.A.T.E. method, it does not lend itself well to incorporation into existing finite element packages. To prove the usefulness of a new method, it is unwise to just jump into the solution of a difficult problem, especially if there is no current standard way of solving it well.

The Navier-Stokes equations, even the simplified ones I am using for this paper, are such an example. They are coupled, and non-linear, and the inclusion of odd partial derivatives (in this case 1^{st} derivatives) leads to an asymmetrical local matrix. This translates into "very hard to solve". Because the equations are non-linear there is more than one right answer. All of these factors make benchmarking a Navier-Stokes solver difficult.

The answer, then, is to take much simpler problems and make sure that my code solves them properly. To this end I picked the Laplace equation and thin flat plate bending as two of my benchmarks. I also looked at the heat equation in one spatial dimension and time.

2. The Laplace Equation

a. Equations

We begin with the Laplace equation, as it is one of the simplest useful 2-D equations around:

Eq. (38) $$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$ *Laplace Equation in 2-D*

It is "elliptic", a term used to describe equation which look like the Laplace in nature. It is used for simple heat conduction (and is often referred to as the "heat equation" for this reason) where there are no heat sources. It can also be used for potential flow problems where the action is very slow, like groundwater diffusion.

b. Physical Problem

For my application of the Laplace equation I chose to solve a groundwater seepage problem found in the book by Smith and Griffiths. The setup is as follows. Imagine a 2-D tank of soil twice as wide as it is tall. The lower, left and right sides are impermeable (there can be no flow through them), giving rise to a boundary condition of:

Eq. (39) $\frac{\partial \phi}{\partial n} = 0$ *Laplace Boundary Condition 1*

where "n" is just the appropriate variable (x on the left and right sides, y on the bottom). There is also a wall which extends from the top center of the soil bed halfway down to the bottom. It has the same boundary condition as the bottom, left and right sides. The right top surface is set to a potential (Phi) of 100, and the top left side has been set to 0. Because of symmetry, only half of the problem need be solved since we know the potential will be 50 at the mid-line, directly beneath the wall.

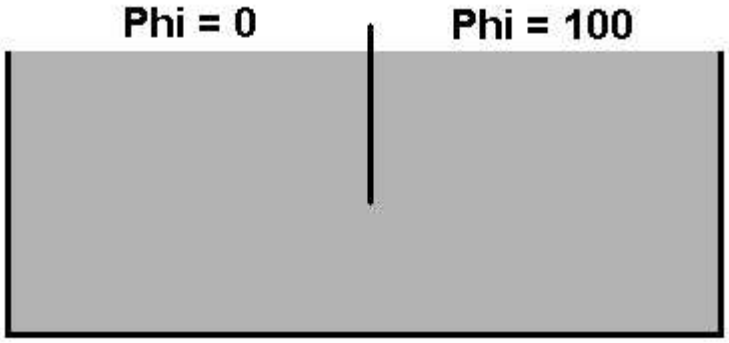


Illustration IX.1 Full Laplace Physical Problem

The book chose to solve the left half, and just to be different, I chose to solve the right half of the problem. The distinction is arbitrary because of the symmetry. Because of the linearity of the problem it could even have been normalized (e.g. From 0.0 to 1.0). Also, the size of the domain does not matter, since all of the elements are perfectly square, but I choose to have it be 2 by 2 meters. Be that as it may, here is my setup:

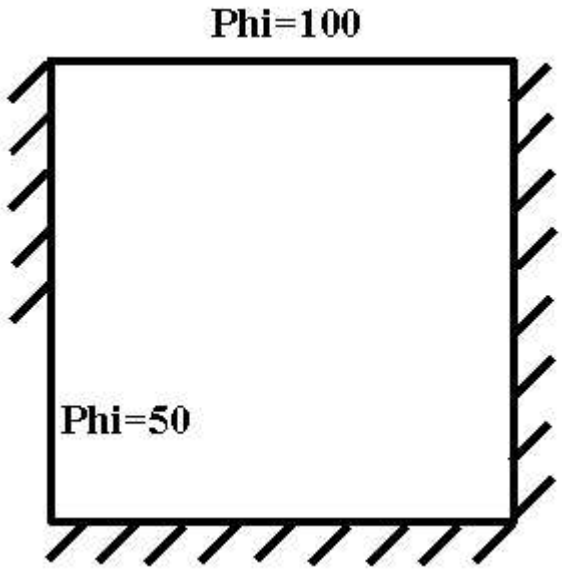


Illustration IX.2 One-Half Laplace Physical Problem

c. Input

For the test problem in the book they divided it into 36 linear elements (a 6x6 grid, for a total of 49 nodes). I solved the problem with the same exact mesh as a check. I also did a convergence study, but here I will only present the input file and mesh from the test case the book chose.

```
2 blocks_x
2 blocks_y
3 div_x
3 div_y
0 time_order

grid
**
**

mesh heat
111
111
111
```

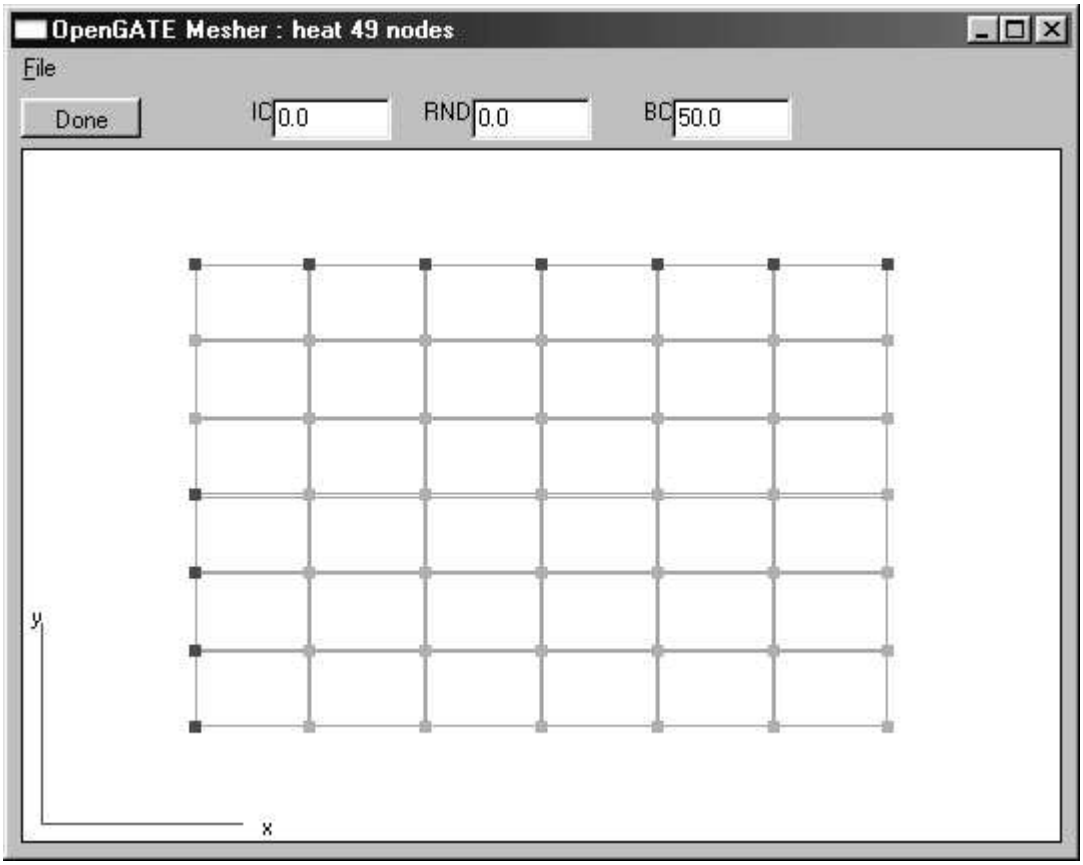


Illustration IX.3 Laplace Equation Sample Mesh

d. Results

The check I chose to use was the value of the center point. In their solution it was given as 20.85 (FORTRAN code and solutions for the problems in the book are available on the web, which is part of the reason I chose this as a benchmark). Using the mirror properties of the solution, I know that the center point on the right side (my side) would have to be 100-20.85, yielding 79.15. The value of my solution at that point was 79.15467, so my method worked for this problem. Following is a graph of this first solution.

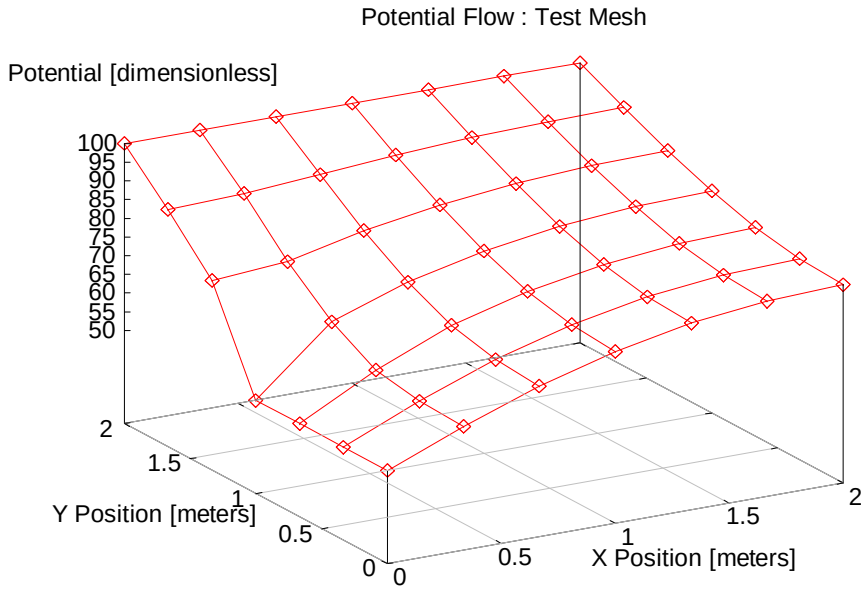


Illustration IX.4 Laplace Equation Sample Mesh's Solution

The convergence study is done to convince all involved (but myself foremost) that the numbers are not a mistake. One of the major problems in finite element analysis is the tendency to believe the numbers a program spits out, knowing full well that you were the one generating the program' s input (if not the program' s code as well). To combat this proclivity, a convergence study obtains solutions at many different mesh refinements while holding the boundary conditions constant. If they are all in roughly the same area, it is a good indication that the solution is progressing in the right direction. If you plot a solution value at three or more different mesh refinements, you may find the asymptotic solution and assume that it is nearly correct.

Typically a more refined mesh gives better solutions, but there are exceptions to this. Very small elements, high aspect-ratio elements (this is much worse if

the inverse of the Jacobian was not applied), very skewed elements, and very high order polynomials can all contribute to numerical inaccuracies which will then degrade the solution at more refined meshes, instead of improving it. In my meshing utility I disallow any polynomial above 5^{th} order.

The mesh can be refined in two ways: either by using more elements, or by using higher-order elements. These are called h- and p- type mesh refinements, respectively. My code is well suited for both, and the following tables show my results:

h-Type Mesh Refinement (all linear elements):

Mesh	***4x4***	***6x6***	***8x8***	***10x10***	***20x20***
Value	78.9197	79.1546	79.2750	79.3531	79.5851

p-Type Mesh Refinement (all within a 4x4 grid):

Order	***1st***	***2nd***	***3rd***	***4th***	***5th***
Value	78.9197	79.1128	79.1082	79.1071	79.1069

The following chart shows the midpoint value versus the total number of nodes in the mesh. The number of nodes (degrees of freedom) was chosen because it is a simple way to characterize any mesh, no matter how convoluted it may be.

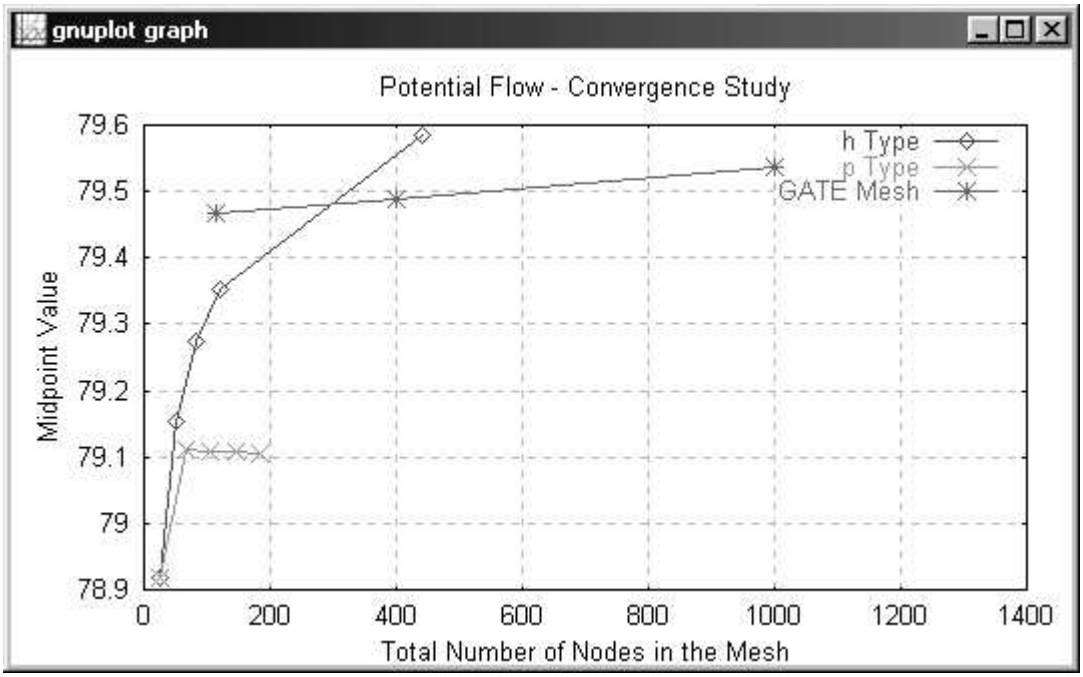


Illustration IX.5 Laplace Equation Convergence Study

As you can see, the values are heading toward 79.6 for the h-Type refinement, and toward 79.1 for the p-Type. Unfortunately, I did not have the analytical answer for the midpoint value: this example was done more to determine if I could duplicate someone else's finite element solution than to see if I could converge to a known value.

This chart shows an example of a byproduct of how the elements are generated. Since the approximation function is driven only by nodes on the boundaries of an element, it is important to have nodes where things are happening. If the mesh is too small (as in our 4x4 case used for the p-type mesh refinement), even using 5^{th} order polynomials on every side does not capture the solution very well.

It is possible to use the Boolean sum with nodes in the center (it merely

requires a simple upgrade to the projections' blending functions). The most difficult part of implementing this would be in the file format for specifying which nodes go where. The corner numbering and default node orders discussed earlier would need to be completely rewritten. And right now it is just as easy to add a few more elements in the area of interest.

One final check was run with meshes which only G.A.T.E.' s could populate. One consisted of a 10x10 grid with all 5th order elements, and the other two meshes had varying orders, with more nodes nearer the point (x=0, y=0.5). These are labeled "GATE Mesh" on the convergence chart above. Here are the mesh order input specifications:

1 1 1	3 3 3	5 5 5
3 1 1	5 3 3	5 5 5
1 1 1	3 3 3	5 5 5

3. Thin Flat Plate: Kirchhoff Formulation

a. Equations

These equations I am testing are for calculating the deformation of a thin flat plate. There are actually more than one series of equations describing this phenomenon, but I chose the Kirchhoff formulation. One very good thing about these equations is that there is a formula for the deformation of the center point given certain boundary conditions and a constant distributed load. This permits me to check my work analytically.

I chose this problem for two reasons. First, it introduces a typical mechanical

engineering problem. And second, this gets us started on coupled partial differential equations. These equations are linear, so it is still a much simpler case than fluid flow.

The variables involved in this are: w = the deflection of the plate in the z direction, Mxx = bending moment in the x direction. Myy = same in the y direction, and Mxy = the cross bending moment (same as Myx). The constant D is just a convenient way to store a bunch of other constants which are always grouped. It is defined to be:

$$D=\frac{1}{12}\frac{E*h^3}{(1-\nu^2)}$$ *Eq. (40) Flat Plate Constant "D"*

Here are the equations for the flat plate bending:

$$Mxx-D(\frac{\partial^2 w}{\partial x^2}+\nu\frac{\partial^2 w}{\partial y^2})=0$$ *Eq. (41) Flat Plate Mxx*

$$Myy-D(\nu\frac{\partial^2 w}{\partial x^2}+\frac{\partial^2 w}{\partial y^2})=0$$ *Eq. (42) Flat Plate Myy*

$$Mxy-D(1-\nu)\frac{\partial^2 w}{\partial x\partial y}=0$$ *Eq. (43) Flat Plate Mxy*

$$\frac{\partial^2 Mxx}{\partial x^2}+2\frac{\partial^2 Mxy}{\partial x\partial y}+\frac{\partial^2 Myy}{\partial x^y}-P=0$$ *Eq. (44) Flat Plate Equilibrium*

And here is the equation for the deflection of the midpoint of a simply supported square of side length L, and Nu = 0.3:

$$w(Midpoint)=\frac{0.044361*P\,L^4}{E\,t^3}$$ *Eq. (45) Midpoint Deflection Under Constant Load*

b. Physical Problem

This test case is a square with a constant pressure applied to one side. It is

simply supported along all four sides. The dimensions of the plate are 1[m] by 1[m] by 0.01[m] thick. E for this plate is 1.0e6, and Nu is 0.3 (of course). The pressure P is a constant 1.0. The boundary conditions are as follows:

$$\begin{bmatrix} \boldsymbol{w=0} \\ \boldsymbol{Mxx=0} \\ \boldsymbol{Myy=0} \end{bmatrix} \rightarrow on\ all\ sides$$
$$[\boldsymbol{Mxy=0}] \rightarrow at(x=0,\ y=0)$$

Eq. (46) *Flat Plate Boundary Conditions*

c. Input

Here is the sample input file and mesher screen shot:

```
1 blocks_x
1 blocks_y
4 div_x
4 div_y
0 time_order

grid
*

mesh w
44
44

mesh mx
44
44

mesh my
44
44

mesh mxy
44
44
```

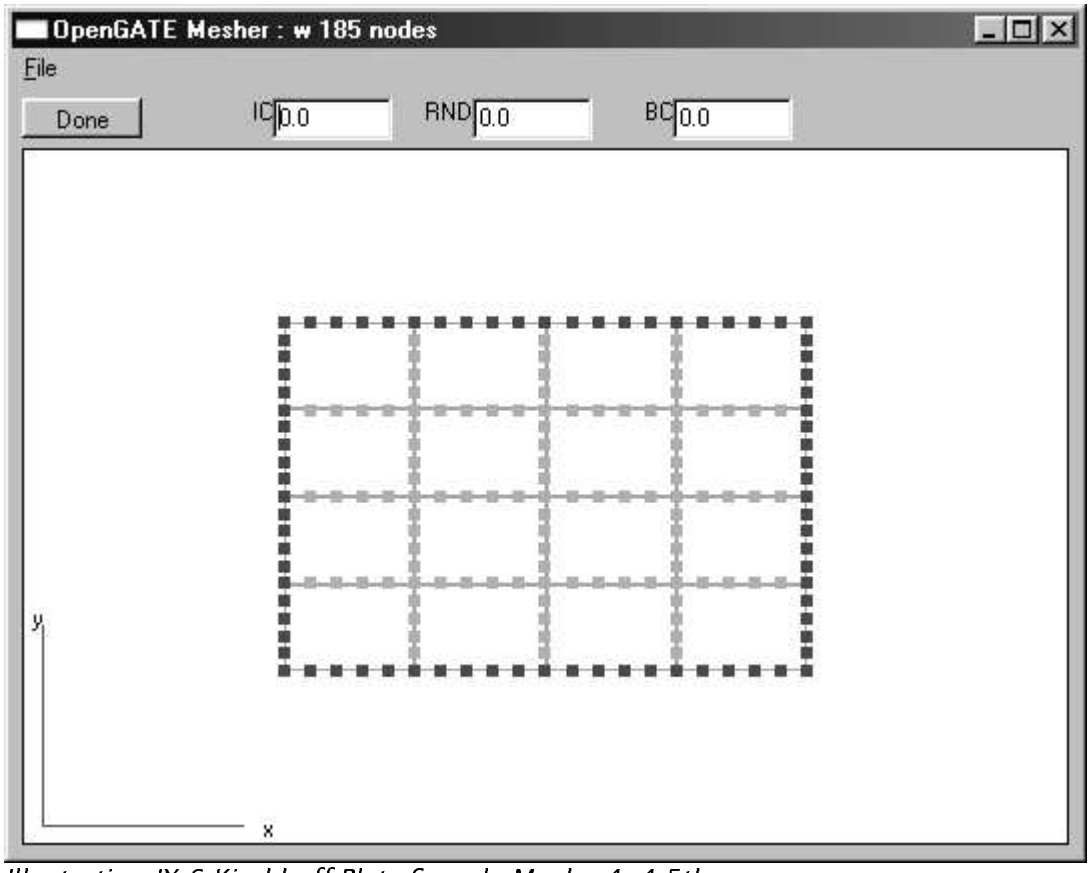


Illustration IX.6 Kirchhoff Plate Sample Mesh : 4x4 5th

d. Results

The following graphs show the results of the 10x10 linear and 4th order cases. The convergence check is presented immediately afterward. The shape of the graph is correct, and it is the value at the midpoint which I am using as a check. According to the formula my midpoint deflection should 0.044361 meters.

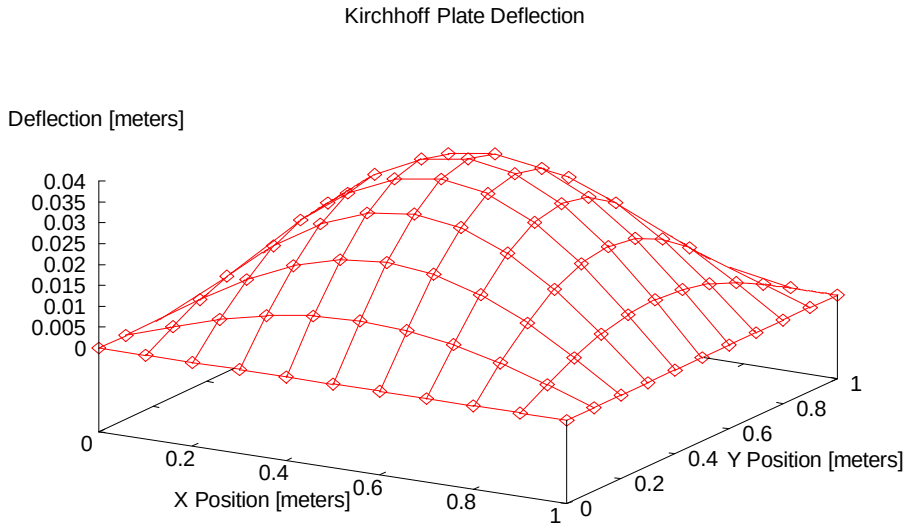


Illustration IX.7 Kirchhoff Plate Results : 10x10 1st

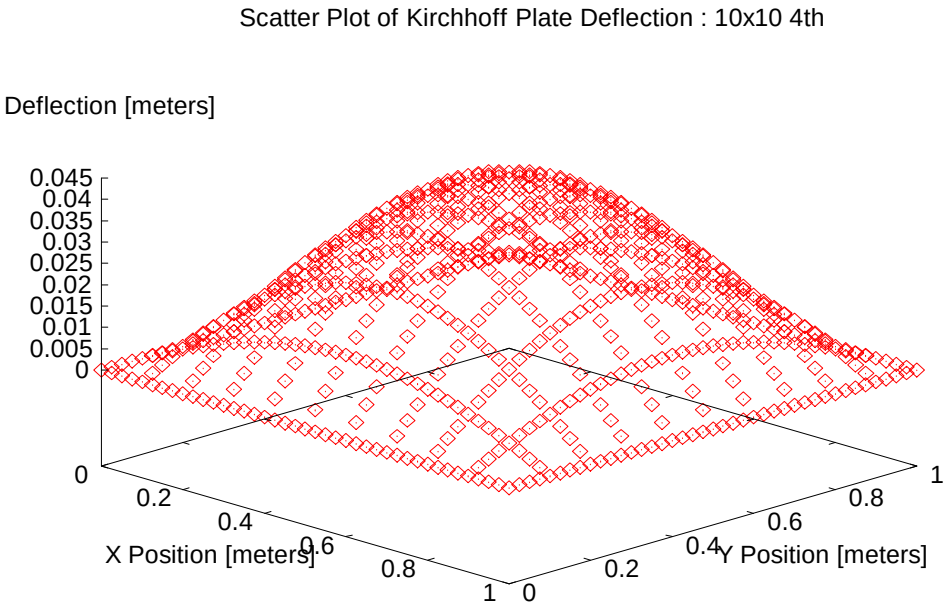


Illustration IX.8 Kirchhoff Plate Results : 10x10 4th

A convergence study is presented below where I increase the mesh in both h

and p (number of elements and polynomial order). Since this is the first time I have multiple field variables, I want to point out that I need not increase the order of every one. It is permissable, sometimes even desirable, to increase the order of one of the field variables and not the others.

In this case I am increasing the order of all of them equally, for the sake of balance (all should improve equally). A rule of thumb which I use is to scale the base order of the elements by the order of each field variables' derivatives. In this case all of them have 2nd derivatives, so that is another reason to keep them all equal for this study.

For the h-type mesh refinement I go from 4x4 linear elements up to 14x14 linear elements. The p-type mesh refinement uses the 4x4 mesh with linear through 5th order elements all around. The "GATE Mesh" takes advantage of the elements' ability to have different orders on different sides.Some simple experimentation showed that high order polynomials were most important on the corners, slightly less important on the sides, and the center was least important. To implement this I split the flat plate into 4 macro blocks, each of which had 3x3 elements in it. The mesh order specification looked like this for the three points on the GATE plot:

2 1 2	4 2 4	5 3 5
1 1 1	2 1 2	3 1 3
2 1 2	4 2 4	5 3 5

Below is the mesh for the last case (5, 3, 1):

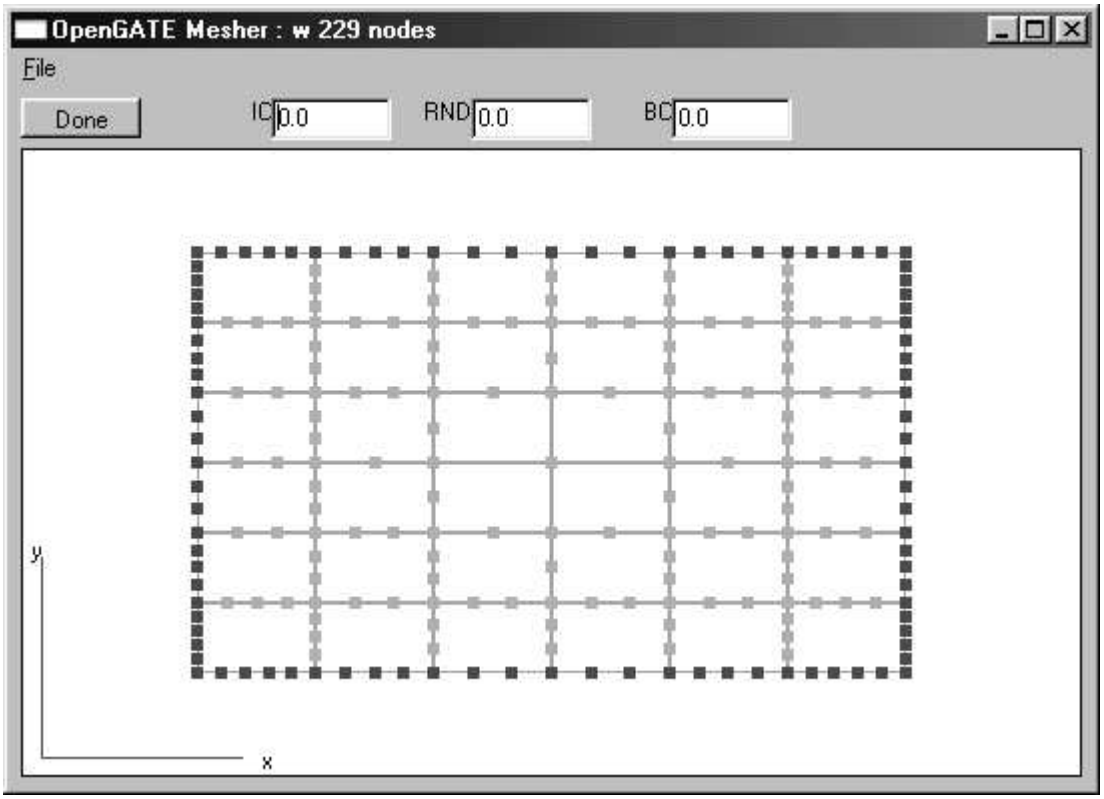


Illustration IX.9 Kirchhoff Plate GATE Mesh : "5 3 1"

For the graph below, the midpoint deflection values were plotted against the total number of nodes in each mesh:

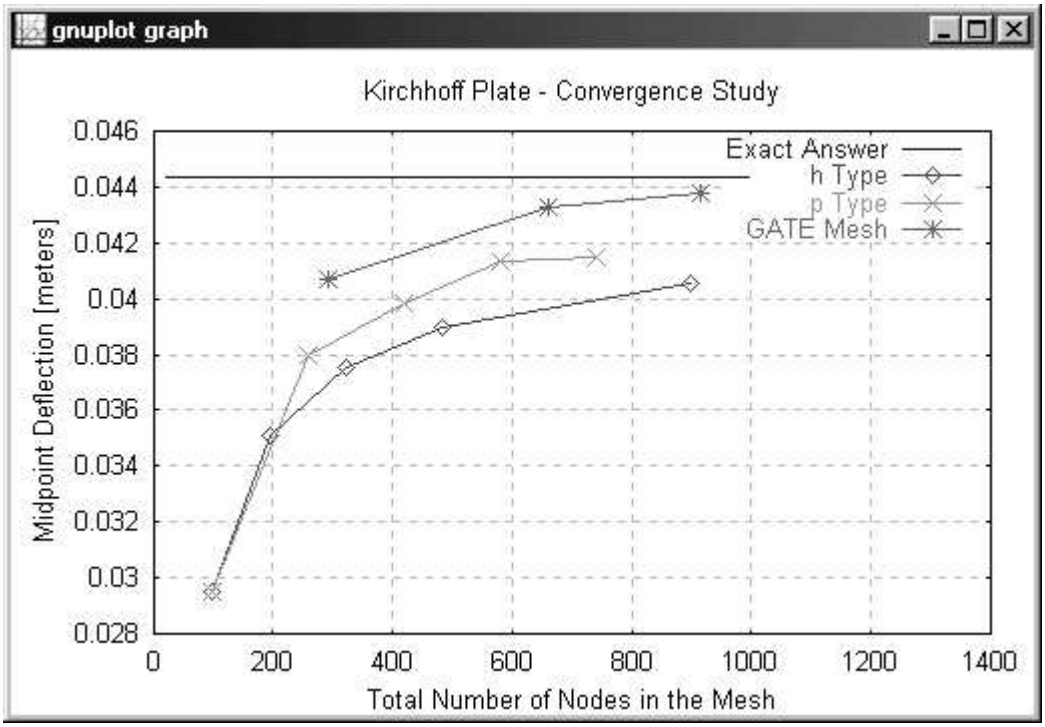


Illustration IX.10 Kirchhoff Plate Convergence Study

As you can see, the problem seems to be converging to a value near the correct one for all of the mesh refinements. It is obvious that taking advantage of the G.A.T.E. formulation can yield a quicker convergence. By allowing more degrees of freedom in the elements where more change occurs, and keeping them simpler elsewhere, the solution converged with many fewer nodes than would be possible by simply throwing more elements at the problem. Also, in most commercial codes you would only be able to use linear or second order elements, and not both at the same time.

4. Heat Equation with Time

a. Equations

Now we go back to the first test case, but with a slight change. This problem is still in two dimensions, but since one of them is time, this changes the fundamental equation. The reason for this is that I can test the behavior of the time stepping method that will be used in the Navier-Stokes case. I will solve the problem, both with multiple elements in the time direction, and also with a single element in the time direction coupled with an iterative time step solution.

The new form of the equation allows me to set the initial conditions of the problem and let it run through time. Because of the simple nature of this problem' s equation, the physical layout and the boundary conditions, I have an explicit analytical solution.

The equation:

$$\frac{\partial^2 \Phi}{\partial x^2} - \frac{\partial \Phi}{\partial t} = 0 \qquad \textit{Eq. (47)} \quad \textit{Heat Equation: 1-D with Time}$$

And the solution at x = 0.5:

$$\Phi(0.5) = \frac{1}{4} e^{-12t} \qquad \textit{Eq. (48)} \quad \textit{Particular Solution at x=0.5}$$

b. Physical Problem

The layout is again the standard square, which goes from 0.0 to 1.0 in both x and t. The boundaries are as follows:

$$\begin{matrix} \Phi(t=0) = x(1-x) \\ \Phi(x=0, x=1) = 0.0 \end{matrix} \qquad \textit{Eq. (49)} \quad \textit{Heat and Time Boundary Conditions}$$

The initial condition is specified at t=0, and the other boundary conditions are

valid throughout all of time. This is the first problem where I do not get to keep the elements square, so here the methods discussed in the "Change of Variables" section must be applied.

c. Input

There are two different ways I am solving this. First, I use a 2-D mesh in x and t. All of the elements are generated at once. For the second case, I will create the mesh only a single element deep in the time direction. In order to get the full solution, I need to iterate. First, I solve the system, then the values of the nodes at time = dt are copied to the values of the nodes at time = 0. The entire thing repeats until the final time is reached (should be t_{final} / dt iterations).

Here is a screen-shot of a simple mesh used for the first method:

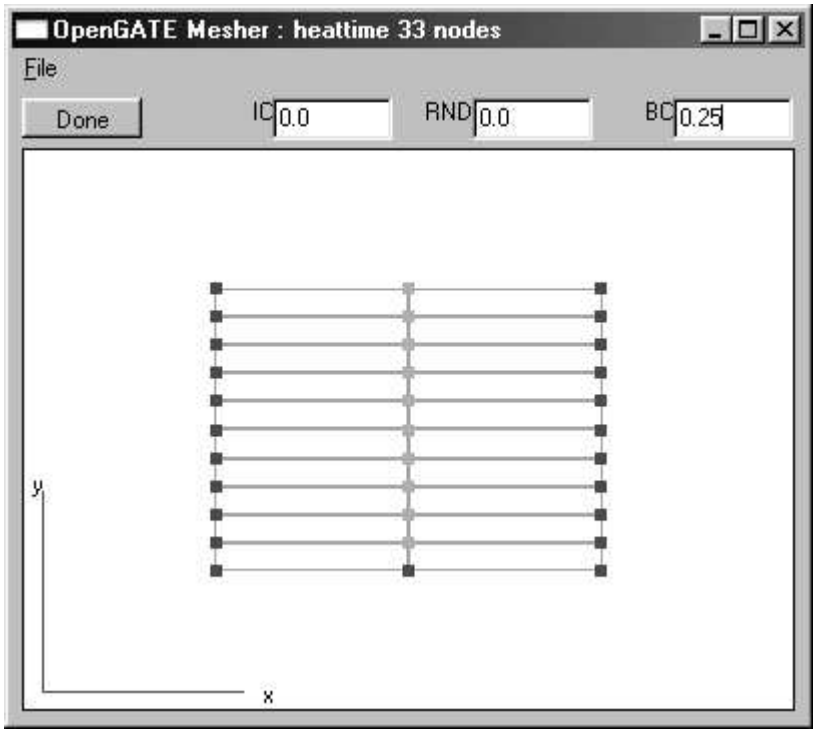


Illustration IX.11 Heat with Time : Single Mesh

The reason for using both methods is mainly to test the second one. It is the method I will use in the fluid flow problem. I want to see how the solution is affected by not having a full mesh to influence the solution at the interior points.

d. Results

First, I want to show this plot, which demonstrates the basic shape of the solution. The initial temperature distribution is seen at the far end of the waterfall plot, and as the ends are clamped at 0, you can see the rest of the heat energy dissipate. The directions are correct, but the x and time scales are off (they are based on matrix position, not actual values).

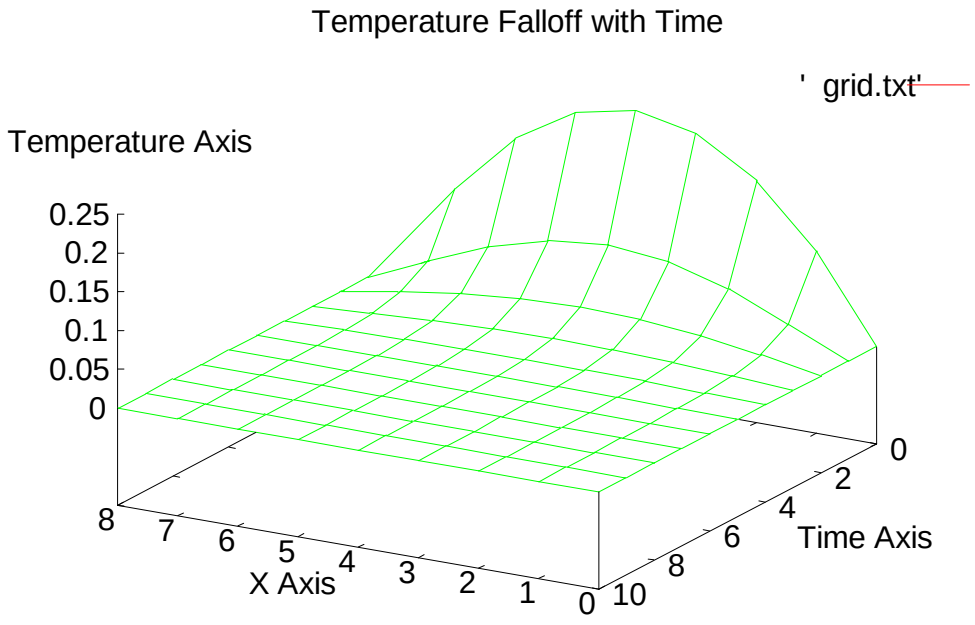


Illustration IX.12 Heat with Time Sample 2-D Solution

This next graph shows the actual values at x=0.5 along with the solutions of multiple meshes. All of the elements are simple linear ones (because of the ease of specifying the boundary conditions), so it is just a plot of h-type mesh refinement. This is with all of the elements generated at once in a 2-D mesh. The next part will show the time marching solution.

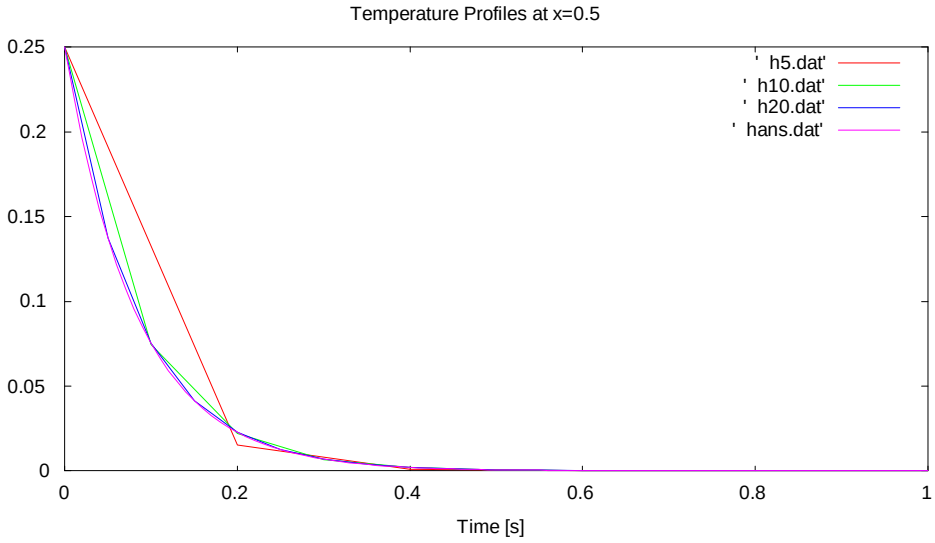


Illustration IX.13 Heat with Time : Midpoint vs. Time Solutions

As you can see, they all look about the same. Even the 2x5 mesh comes pretty close considering how few nodes it had to play with. The last graph in this section shows the convergence of the value at (x=0.5, y=1.0).

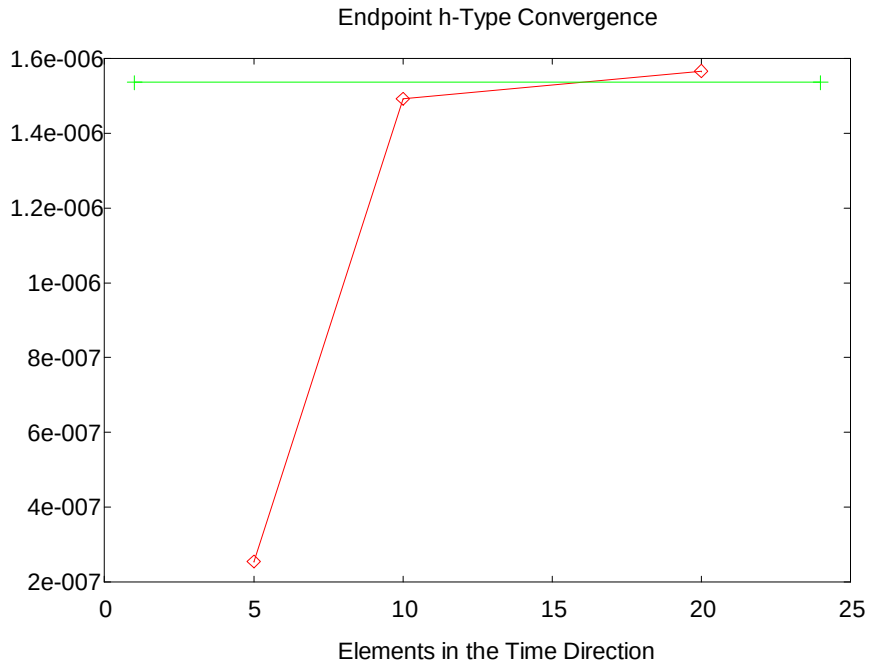


Illustration IX.14 Heat and Time Endpoint Convergence

This next section shows how the iterated solution compared to the original solution. Below is a plot showing the percent error (100 * (Tapprox - Texact) / Texact) versus time for both the iterative scheme and the single-mesh scheme with 100 elements or steps in the time direction. It is plotted on a log scale because of the large difference in magnitudes. As you can see, the iterative solution is worse than its single-mesh counterpart. At the last data point (t=1.0) the single-mesh solution was at 0.12% error, while the iterative solution had a 25% error. This looks bad, but the final data point was 1.54e-6, so the iterative solver was not very far off (at 1.92e-6).

The inferior behavior of the iterative solution is partly due to not having the benefit of the influence of boundary conditions further along in time. All of the node values in the single-mesh version are solved for at the same time, so all of the boundary conditions can contribute to the solution of every node. And at the same time the nodes along the solution front of the iterative solution are on or at the boundaries of the domain, where solutions tend to be less accurate.

Illustration IX.15 Heat with Time : Method Error Comparison

Even though the behavior of the iterative method is not as good as the single-mesh approach, it does yield answers close to the correct values. One major benefit, however, is its severely reduced mesh size. While it makes no difference on a problem of this size, the airflow problem is much more complex. Further study needs to be done to compare this iterative method with the more standard iterative method where the time stepping is done with finite difference formulations.

X. Navier-Stokes Equation Derivation

1. Which and Why

As a precursor to this problem I solved some simple cases as you saw in the previous section. These are much more trivial problems, and there is not much to be gained in moving from pre-packaged software to a fledgeling method to solve them. It is important to prove that these automatically derived G.A.T.E. elements are useful, even advantageous, in solving a more difficult class of problem.

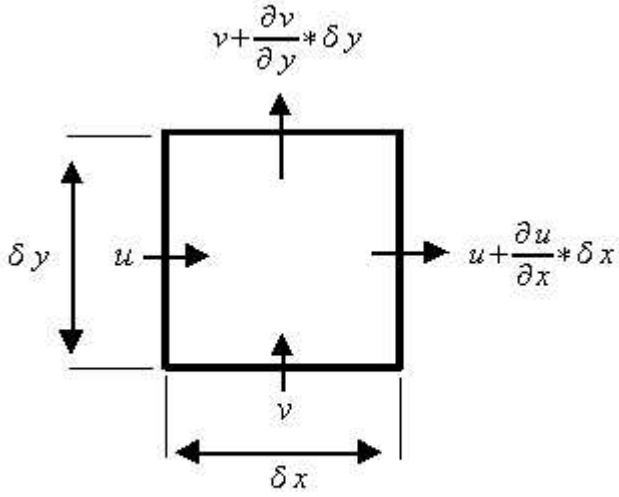
I decided that a simplified set of Navier-Stokes equations would both prove that these elements could indeed be used in a difficult class of problems, and at the same time not overwhelm me. The equations I settled on describe fluid flow when the fluid is incompressible, viscous, and adiabatic (no heat transfer). These are the assumptions I am using. On the hard side, there are still three field variables: pressure, velocity in x and velocity in y. And the equations are non-linear.

2. The Derivation

I will now provide the derivation of the equations I am using. This will be relatively short, and is not intended as a tutorial. It is designed so that anyone who knows their way around finite elements can follow what I am doing.

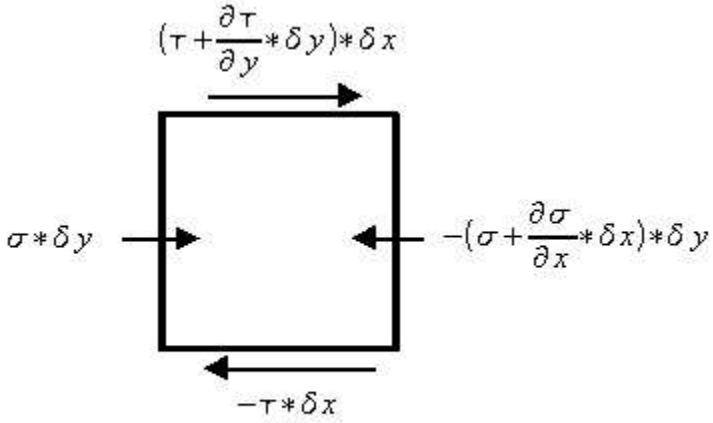
The first equation is just continuity of mass. The following figure illustrates the variables and sign convention I am using. U is the fluid' s velocity in the x

direction, V is velocity in the y direction, p is pressure, sigma (σ) is the normal stress, and tau (τ) is the shear stress. The last two will be substituted away in the final form.



Velocities

Illustration X.1 Navier-Stokes Velocity Diagram



Forces in the x Direction

Illustration X.2 Navier-Stokes Forces Diagram

The mass entering and exiting the control volume is:

Eq. (50) $m_{in} = u * \delta y + v * \delta x$ N-S: Mass in

Eq. (51) $m_{out} = (u + \frac{\partial u}{\partial x} * \delta x) * \delta y + (v + \frac{\partial v}{\partial y} * \delta y) * \delta x$ N-S: Mass Out

Subtracting Mass in from Mass out, removing the area term, and setting it equal to 0 (because it is incompressible) yields:

Eq. (52) $\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$ N-S: Continuity of mass

The next equation is from momentum in the x direction. I will then simplify the equation and extract the momentum equation in the y direction with some mystic hand waving. We start with the basics:

Eq. (53) $F_x = m * a_x = m * \frac{\delta u}{\delta t}$ N-S: F=ma(x)

Eq. (54) $F_x = \sigma * \delta y + \tau * \delta x + \frac{\partial \tau}{\partial y} * \delta y * \delta x - \tau * \delta y - \sigma * \delta y - \frac{\partial \sigma}{\partial x} * \delta x * \delta y$ N-S: F=ma(x) Expanded

Remembering to take a total derivative to get du over dt, and then substituting in the following approximations for Newtonian fluids:

Eq. (55) $\sigma = -p + 2 * \mu * \frac{\partial u}{\partial x}$ $\tau = \mu * (\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y})$ N-S: Approximations for Newtonian Fluids

We get the grand total of the momentum equation in x (keeping all things constant except u, v and p):

Eq. (56) $\rho * [\frac{\partial u}{\partial t} + u * \frac{\partial u}{\partial x} + v * \frac{\partial u}{\partial y}] = -\frac{\partial p}{\partial x} + \mu * (\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2})$ N-S: Momentum in x

And similarly for momentum in the y direction:

Eq. (57) $$\rho*[\frac{\partial v}{\partial t}+u*\frac{\partial v}{\partial x}+v*\frac{\partial v}{\partial y}]=-\frac{\partial p}{\partial y}+\mu*(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2})$$ *N-S: Momentum in y*

Equations (52), (56) and (57) are the three I need to solve to get u, v and p. The constants I use in my code are for air at 20 degrees C at one atmosphere. Density (ρ) is 1.204 [kg/m^3], kinematic viscosity (ν) is 1.51e-5 [m^2/s], and the dynamic viscosity (μ) is just the product of density and kinematic viscosity.

XI. Navier-Stokes Example

1. Physical problem

The Navier-Stokes equations are traditionally tricky to solve. My code generates the series of nonlinear equations from the partial differential equations and the desired mesh. This was in fact the easy part of the problem. It takes less than half an hour to convert my base code to the Navier-Stokes equation generator. Unfortunately, my solver has a very difficult time solving these equations. Because of their non-linear nature, not only are they hard to solve, the equations generated are larger.

Because of this, I had to solve a much simpler case than I had intended to solve. Again, this is a simple square, 1 meter by 1 meter. The elements extend 1/250th of a second in time. The boundary conditions are simple.

- At y=1: u=100 & v=0
- On all other sides: u = v = 0

2. Input

1 blocks_x
1 blocks_y
5 div_x
5 div_y
1 time_order

grid
***

mesh u elements
11

11

mesh v elements
11
11

mesh p elements
11
11

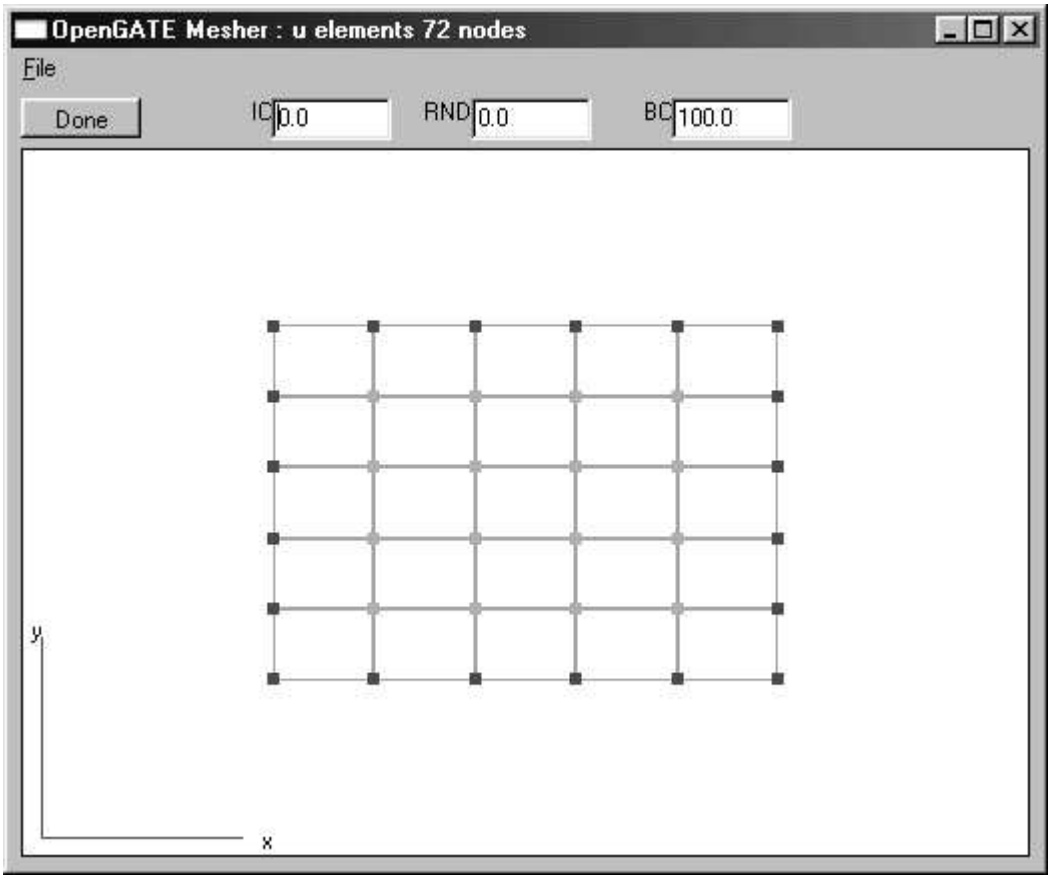


Illustration XI.1 Navier-Stokes Mesh

3. Results

Of all the problems solved for this paper, this particular one was the most difficult. Originally I wanted to start with the steady state case. Given enough time my solver did come up with an answer. The answer was unverifiable: it

looked random. While this was not was I was expecting, it was not necessarily wrong. The Navier-Stokes equations are non-linear, so there are many correct answers, and the solution is extremely sensitive to initial conditions. To verify that my equations were correct, I used a mesh straight out of an example from the Smith and Griffiths text, and then used their final answer as the initial condition for my solution. I then allowed my solver to iterate until the residual was less than 1e-10. Here was the result:

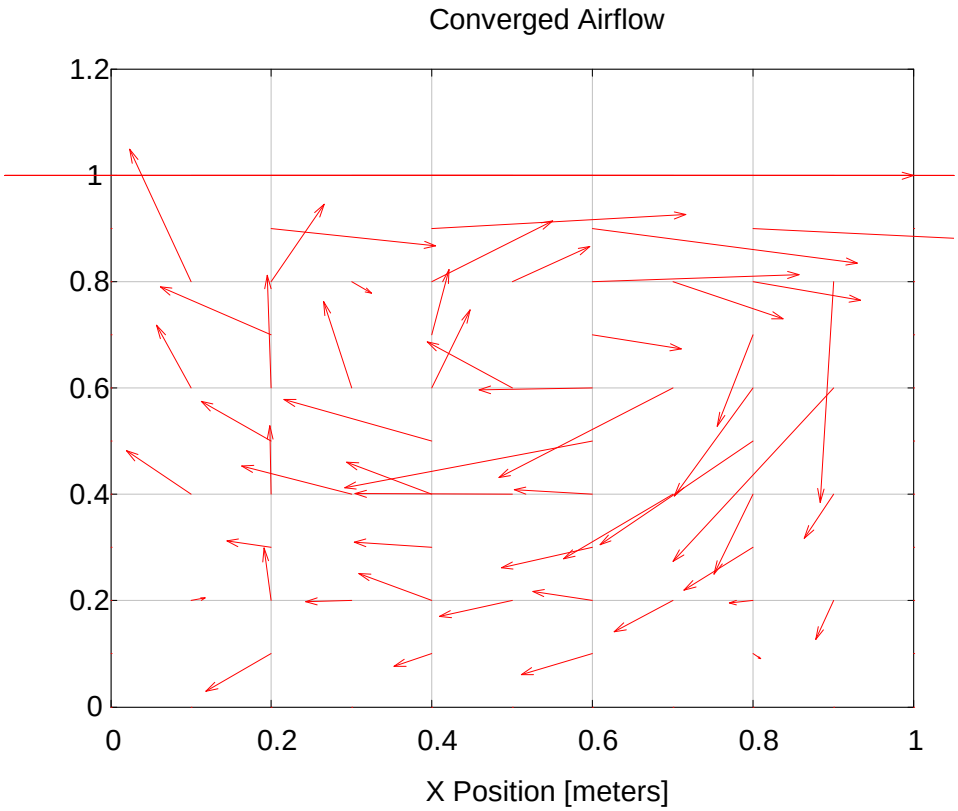


Illustration XI.2 Navier-Stokes Steady-State Solution

This solution is much easier to accept intuitively. It seems that my equations were being generated correctly, but it was my solver which was having such a hard time.

I decided to move on to the unsteady solution of the Navier-Stokes equations. The justification is that while the equations are more complex, they tie the solution of each time step to the solution of the previous time step. This has the effect of discarding any solution which may satisfy the equations but is otherwise completely random. Here again my solver was my downfall.

Starting with the initial conditions described above in the "Physical Problem" section, the residual was fairly small, and my solver was able to drive it below my specified tolerance. However as the flow developed, the initial residual continued to increase, and my solver was unable to drive it to the desired tolerance. Because of this I chose to take a compromised solution.

The data presented below is a snapshot of the process. It is taken before the flow is fully developed, but also before the residual has grown too large. The challenge was to find a solution which was not generating too much residual but still demonstrated correct behavior. I chose to use the 50^{th} time step, so it is at time = 0.2 seconds. The residual at that point was 0.0032, which is a far cry from my desired 1e-10, but on the same order of magnitude as the residual at which the previous steady state example was solved (the book chose to solve till the residual was at or below 0.001).

Here are my results:

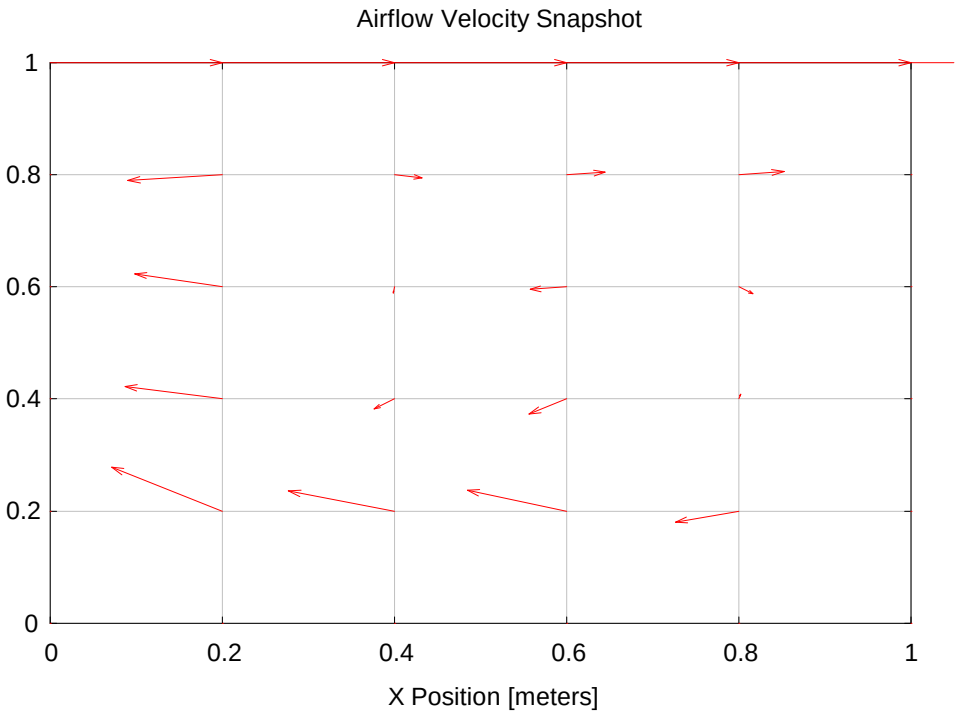


Illustration XI.3 Navier-Stokes Transient Flow Solution

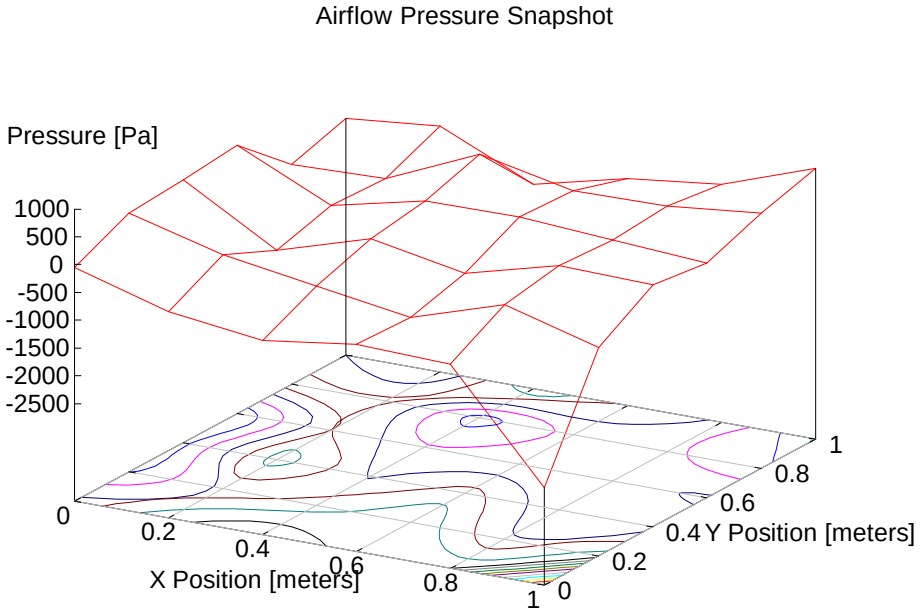


Illustration XI.4 Navier-Stokes Transient Pressure Solution

If the solution continued until a steady state was reached, the behavior looked almost plausible, but the magnitude was much too large (almost 5x the boundary conditions at worst). Of course, at that point the residual was around 300, so not even my solver thought it was a good solution.

Most of the literature I read concerning the unsteady Navier-Stokes equations mentioned that there was a restriction on the time step for stability. The time step had to be such that any given differential piece of fluid could not entirely jump over an element in its path. For example, in my problem, the elements were 0.2 meters wide, and the boundary flow was 100 m/s. To traverse the element at full speed would require 1/500 of a second. I tried a much smaller time step (1/1000 second) and also used the 1/250 second time step, but with reduced boundary speeds (50 m/s, 10 m/s, 1 m/s). I found that while my solutions were not necessarily correct, at least they were stable for all time step sizes.

I attribute this to the fact that I am using a finite element to model the entire region of the domain (including time). Normally the time dimension is separated and treated with a simple finite difference scheme. This would mean that the time step is operating completely independent of boundary conditions. In my method the boundary conditions still apply, regardless of the time step size. So with a huge time step all you lose is accuracy, not stability.

Also, even though there are twice the number of nodes in my mesh (for a 1st order in time formulation) than in a separated scheme, only half of them need be solved (as at each step the solution from the last iteration' s front is used as a

boundary condition). So you gain unconditional stability without having to solve for more nodes. Overall, I am happy with the code which produces the equations, but not with my solver.

XII. Advantages and Limitations of the Method

1. Introduction

This short section recaps the pros and cons of my method of automatic generation and usage of Gradient Adaptive Transfinite Elements. I' ll begin with the advantages.

2. Advantages

One of the benefits that is exclusive to the GATE method is the ability to change the order of elements within a mesh. This allows the most computation to be done where it is needed, maximizing the efficiency of the finite element method.

Another benefit is that I always have exact integration. There are no numerical methods used to introduce error while integrating the residual of an element. The same is true for non-linear equations. The equations are preserved intact, and the linearization method (if desired) is left to the solver.

A benefit of the GATE method which has not yet been demonstrated in my code is the ability to specify derivative boundary conditions explicitly. Current methods require that another approximation of the partial derivative of the element be made, and another set of equations must be superimposed on the equations generated by the finite element method proper. The Boolean Sum has the ability to include the derivative conditions into the element formulation, thus reducing error.

And finally, because of the automatic nature of the element generation, elements of any dimension are feasible. Currently almost everything is attempted in 2-D, with any time dependent terms treated separately. Three dimensional problems are rare, and 4-D problems are non-existent (e.g. 3-D in space plus time). Even higher dimension problems are possible given a suitable mesh generation utility. This could be of great benefit to chemical engineers, to whom the current finite element methods are almost useless because they are so restrictive.

3. Limitations

There are three fundamental limitations of this method which are shared with all other finite element methods. The first, and perhaps most obvious is the problem of machine accuracy. I call for long double variables in my code, which translates at least into 8 bytes per variable. On some machines I get 12 bytes, but in any case the accuracy is limited. For most problems this is sufficient, but some problems (like trans-mach air flow) are notorious for being sensitive to machine accuracy.

Another common problem stems from the use of polynomials as the approximation function. Apart from their many convenient features, polynomials can only simulate smooth functions. It goes back to the old adage, "It looks like things which look like this." This means that shocks or discontinuities of any sort cannot be modeled well with polynomials.

Also, using high order polynomials makes the solution more sensitive to small changes in node values. For some cases the interpolation can extend well beyond the range of the data points. For this reason I limit my code to 5th order polynomials (6 data points) on a 1-D element. This limit can be changed at will, but I did not want people to use 6th order polynomials or above without being aware of the problems. The following graphs show a sample of this issue.

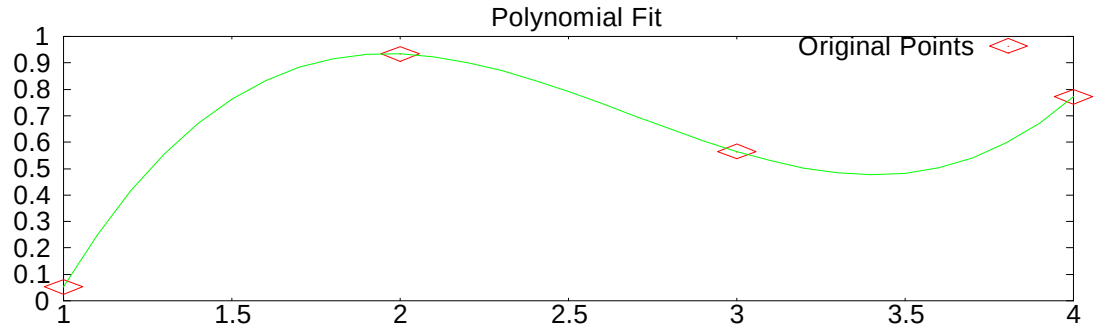


Illustration XII.1 Sample 3rd Order Polynomial

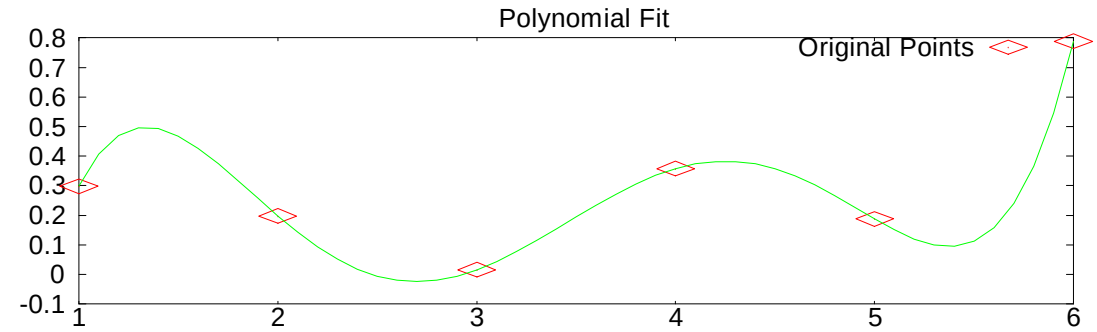


Illustration XII.2 Sample 5th Order Polynomial

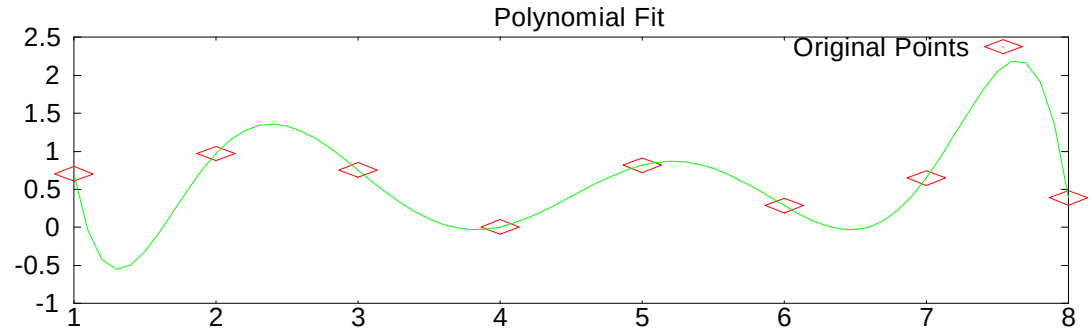


Illustration XII.3 Sample 7th Order Polynomial

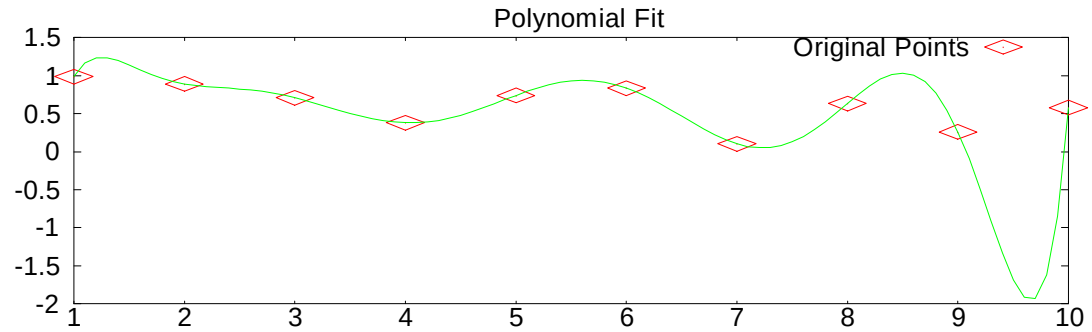


Illustration XII.4 Sample 9th Order Polynomial

The last problem is that of mesh definition. A bad mesh will yield bad results, no matter which solution method you decide to use. The answer: always do a convergence study. If the answer looks the same with n, 2n and 4n nodes (or if you are really masochistic: n, 4n, 16n, etc.), then you may be on the right track.

Next are the limitations which are exclusive to my code. First and worst is speed. Because the method I chose to represent exact polynomials is memory intensive (linked list, dynamically allocated), the speed of execution is bottlenecked by the system memory access speed. Especially in the "Compact" routine, which searches for like terms and combines them. Without this routine even tiny problems will take more resources than a machine can give. A typical case in the Navier-Stokes problem reduced a polynomial comprised of 551,614

terms down to 417. This was done for every node, and this routine is $O(n^2)$.

Often memory is the slowest part of the computer architecture. If your 1GHz machine uses PC100 RAM, then you can only get data at 100 MHz, approximately a factor of 10 less than the processor could handle (or worse with modern processors). Modern RAM is much faster, and the speed continues to increase. Even though most FEA programs are not run on home computers, the issue remains.

Another limitation is that my method is often impractical to implement with existing (commercial) finite element packages. This method stresses flexibility and exactness, while commercial codes are all about fitting a problem into a matrix and then solving it. While my code can easily be used to derive an element' s local matrix for use in other codes, the odds are that all the simple elements have already been derived. Typically it would be less work just to use a bunch of those elements than spend the time creating and integrating a custom element.

And the final limitation of this method is its inability to handle any other elements but the n-unit ones (i.e. No triangles, tetrahedrons, etc.) While this can be overcome somewhat with careful meshing, there are still several instances where it would be nice to be able to incorporate them. As I have mentioned previously, however, triangular elements can suffer from severely reduced accuracy, and it is more difficult to guarantee that you are applying them properly.

XIII. Future Changes / Improvements

1. Introduction

In this section I describe both what I would like to have completed, and the improvements which need to happen for this to become a viable method in commercial finite element analysis. As you saw in the previous section, there are reasons to continue to develop this method, code and tools. On the other hand, there are clear reasons why this code can not be used as it currently exists commercially.

As my approach is new (and unique, as far as I know), there will be little call to adopt this method generally. Thus there will be no highly-paid professional programmers working to make the code faster and richer in features. Any adoption of this method and code will most likely be on a grass-roots (academic) level. At the present time, the best I can hope for seems to be "Cult Classic" status.

2. Additions

One of the most important aspects of the Boolean Sum is its ability to incorporate derivative conditions as easily as it does everything else. In fact, the Boolean Sum subroutine in my code can already generate an element with derivative conditions. The reason this is not in use is because it is difficult to specify the desired derivative boundary condition. Each node may have an unlimited number of derivative BC's (1st in x, 2nd in y, and even cross-

derivatives). Thus the addition I most want to see is a simple intuitive specification of how to indicate the desired derivative BC' s, and a simple way of generating the extra variables (degrees of freedom) needed to include them.

The other addition I would like to see is a tool for specifying arbitrary geometries. Once again, my code already has the ability to compute the Jacobian and the determinant of the Jacobian, thus allowing non-rectangular elements. It can even approximate |J| for manifolds, allowing curved elements of any dimension to exist together. The main reason this tool has not been created is lack of time. Both time to make the tool, and the extra time involved in handling the added level of complexity.

The final addition would be extremely easy, given the nature of the process. I would like to make the code run in parallel on a network. Each processing node on a local network could easily be programmed to generate the equations for a portion of all the elements, and send the results back to the spawning computer. The results could then be compiled. There are already parallel solvers available, and the resulting system of equations could be converted into the form specified by these solvers. This would tremendously increase the speed of the method, and make it capable of solving commercial problems.

3. Improvements

The first improvement must be speed. Extensive profiling of existing code would be a first step. Other methods of storing an arbitrary polynomial might

need to be found if the speed flaw is inherent in my polynomial code' s algorithm.

A quick means of speeding up the code is to derive the equations for a given element once, then simply do a series of variable substitutions for all other elements with the exact same composition. Thus you would still need to generate new equations for each unique element, but any repetition of element type would occasion no extra CPU time.

An extension to this idea is to save the series of equations for each type of element, coupled with a specific set of PDE' s. Over time a library could be built up, so the speed of the equation compilation process would be on the same order of magnitude as the current matrix methods.

Another improvement I would like to see would require the addition of a simple equation parser. If it could read in multiplication, addition, subtraction and partial differentiation, then the PDE' s could be specified at runtime. There are two very nice applications I would like to see come out of this. One would be a web-served Finite Element solver, accessible to students or employees, without the need to install any software on individual computers. The other would be for coupled problems. Each element could be flagged with which PDE' s it is trying to solve. Thus a problem where a turbine engine is simulated could have two regions: one with combustion, and a simpler region where combustion effects could be ignored. This would finally permit the joining of two problems with different PDE' s, without any loss of information at the interface. The problem could be solved simultaneously, instead of toggling between regions. This would also be useful for aerodynamic simulations, fluid-structure

interactions, etc.

XIV. References

1. Brodkey, R.S., The Phenomena of Fluid Motions, Dover Publications, Inc., New York, 1995
2. Buchanan, G.R., Schaum' s Outlines : Finite Element Analysis, McGraw-Hill, New York, 1995
3. Dietz, D.W., *A Method to Predict Automobile Aeroacoustic Near and Far Field Noise*, U.C. Davis, Davis, 1998
4. Fefferman, C.L., *Existence & Smoothness of the Navier-Stokes Equation*, Princeton University, Princeton, 2000
5. Sarigul-Klijn, N., Dietz, D., Karnopp, D., Dummer, J., *A Computational Aeroacoustic Method for Near and Far Field Vehicle Noise Predictions*, AIAA-2001-0513, Davis, 2001
6. Sarigul-Klijn, N., *Efficient Interfacing of Fluid and Structure for Aeroelastic Instablity Predictions,* International Journal of Numerical Methods in Engineering #47, 2000
7. Sarigul-Klijn, N., *On the Formulation of Gradient Adaptive Transfinite Elements (GATE) Family,* (NSK, ISBN 0-9643757-1-0), 1997
8. Segerlind, L.J., Applied Finite Element Analysis, 2nd Edition, John Wiley & Sons, New York, 1984
9. Smith, I.M., Griffiths, D.V. Programming the Finite Element Method, 3rd Edition, John Wiley & Sons, Chichester, 1998