National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

TECHNICAL MEMORANDUM 97/225
April 1997

IMPLEMENTATION OF A FINITE ELEMENT NAVIER-STOKES SOLVER IN **TRANSOM**

David Hally

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

DEFENCE RESEARCH ESTABLISHMENT ATLANTIC

9 GROVE STREET
P.O. BOX 1012
DARTMOUTH, N.S.
B2Y 3Z7

TELEPHONE
(902) 426-3100

CENTRE DE RECHERCHES POUR LA DÉFENSE ATLANTIQUE

9 GROVE STREET
C.P. BOX 1012
DARTMOUTH, N.É.
B2Y 3Z7

National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

IMPLEMENTATION OF A FINITE ELEMENT NAVIER-STOKES SOLVER IN **TRANSOM**

David Hally

April 1997

Approved by R.W. Graham:
Head/Hydronautics Section

TECHNICAL MEMORANDUM 97/225

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

Abstract

TRANSOM is a multi-block, multi-method Reynolds-averaged Navier Stokes solver being developed at DREA to address problems associated with the flow around ships and submarines. It is multi-block because the flow is divided into several distinct regions. It is multi-method because a different solution method may be used on each of the flow regions. At present two different methods of solution can be chosen: a finite volume solver based on the pseudo-compressibility method; and a finite element solver which uses the penalty function method to determine the pressure.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the design of the finite element flow solver in TRANSOM with emphasis on the class hierarchies used to represent elements, finite element grids, degrees of freedom, and the solver itself. Two companion reports describe the overall design of TRANSOM and the design of the pseudo-compressibility solver.

Résumé

TRANSOM est un résolveur Navier-Stokes pondéré Reynolds, multi-blocs, multi-méthodes, que le CRDA est en train de développer pour résoudre des problèmes concernant l'écoulement autour des navires et des sous-marins. Il est multi-blocs parce que l'écoulement se divise en plusieurs régions distinctes. Il est multi-méthodes parce qu'il permet d'utiliser une technique de solution différente pour chacune des régions de l'écoulement. A l'heure actuelle on peut choisir entre deux méthodes de solution: un résolveur à volumes finis basé sur la méthode de pseudo-compressibilité et un résolveur par éléments finis qui utilise la méthode de la fonction de pénalité pour déterminer la pression.

TRANSOM est écrit en C++ en suivant les principes de la programmation orientée objet. Ce document décrit la conception du résolveur d'écoulement à éléments finis de TRANSOM, en mettant l'accent sur les hiérarchies de classes utilisées pour représenter les éléments, les maillages des éléments finis, les degrés de liberté, et le résolveur lui-même. Deux rapports complémentaires décrivent la conception globale de TRANSOM et la conception du résolveur de pseudo-compressibilité.

Table of Contents

List of Figures

List of Tables

DREA Technical Memorandum TM/97/225

Implementation of a Finite Element Navier-Stokes Solver in TRANSOM

by

David Hally

Executive Summary

Background

The computer program TRANSOM is intended to solve a number of ship-related flow problems of interest to DREA: for example, flows around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, and prediction of roll damping coefficients. In particular it will enhance our ability to predict the flow around a ship hull and into the propellers. Knowledge of this flow is crucial for the design of efficient quiet propellers.

TRANSOM is a multi-block, multi-method Reynolds-averaged Navier Stokes solver; multi-block because the flow is divided into several distinct regions called blocks; multi-method because a different solution method may be used on each of the blocks. Unlike the methods currently in use at DREA, TRANSOM solves the full Navier-Stokes equations in conjunction with a turbulence model; hence, when completed, it will be able to solve the flow around complex geometries such as a fully appended ship hull or submarine. The overall design of TRANSOM is described in a companion document.

The current version of TRANSOM includes two different solution methods which can be used on the blocks. The first is a pseudo-compressibility method based on the work of Rogers and Kwak and developed for DREA under contract by the University of Toronto; it is suitable for use on structured blocks. Turbulence can be modelled using a variant of the k-ε model or using the Baldwin-Lomax model. Currently only two-dimensional flow can be modelled. Details of the implementation of this solver in TRANSOM will be documented separately.

Principal Results

The second solver is described in detail in this memorandum. It uses the finite element method and is based on the program MEF developed at Université Laval; it is suitable for use on unstructured blocks. MEF was modified at DREA to incorporate variants of the k-ε turbulence model; however, the version that is currently incorporated in TRANSOM is restricted to two-dimensional laminar flow.

TRANSOM is written in C++ following principles of Object Oriented

Programming. This document describes the implementation of the finite element flow solver in TRANSOM and, in particular, the class hierarchies used to represent unstructured blocks, elements, and finite element solvers.

Significance of Results

When TRANSOM is fully developed it will provide accurate numerical predictions of the flow into the propeller plane. It will also be applicable to many other flow problems related to ships: for example, flow around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, prediction of roll damping coefficients, and prediction of the lift and drag characteristics of lifting surfaces such as submarine control planes, rudders, and fin stabilizers. The finite element solver described here will considerably ease the burden of grid generation in regions of geometric complexity.

Future Work

TRANSOM is still under development; the current version will only solve two-dimensional flows and requires upgrading in many areas, in particular the ways in which different solution methods interact at their common block boundaries. Further development will be done at DREA and under contract.

1 Introduction

At high speeds the noise radiated by a warship is primarily due to cavitation on its propellers. Naval propellers are designed so that cavitation inception is delayed to as high a speed as possible, but the design method requires an accurate prediction of the flow into the propeller plane. Over the past decade computer programs have been developed at DREA to predict the flow into the propeller plane. Known collectively as HLLFLO[1], these programs use a low-order panel method to predict the potential flow and boundary layer methods to calculate the viscous flow.

The HLLFLO programs have the advantage that they are very fast. They yield fairly good predictions for unappended destroyer hulls[2]; however, there are several areas in which the predictions can be improved.

1. As is typical of boundary layer methods, the wake deficit is over-predicted near the stern. This is especially noticeable at model-scale Reynolds numbers.
2. HLLFLO is incapable of handling the flow past appendages. The wakes shed from the propeller shafts and shaft brackets can cause significant spatially concentrated wake defects which, in turn, affect higher harmonics of the radiated propeller noise and the speed at which cavitation first occurs.
3. HLLFLO does not account for the influence of the boundary layer on the potential flow.

These effects are especially important for smaller vessels with low length to beam ratios. Moreover, HLLFLO is incapable of handling the complex geometries of submarines or advanced marine vehicles such as SWATHs.

It was to address these deficiencies that work on the program TRANSOM was begun. TRANSOM is a Reynolds-averaged Navier-Stokes (RANS) solver: in conjunction with a turbulence model, it solves the Navier-Stokes equations on a grid of points which cover the regions of flow.

When TRANSOM is fully developed it will be applicable to many ship-related flow problems, not just the calculation of propeller inflow: for example, flows around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, and prediction of roll damping coefficients.

The chief practical problem in developing a Navier-Stokes solver for the flow past a ship is that the Reynolds Number is so high: for a destroyer moving at 20 knots the Reynolds number is about 10^9. The higher the Reynolds number, the more grid points are necessary to resolve the severe velocity gradients near solid surfaces. For this reason, issues of computer storage are of extreme importance.

Because of the complex geometries of appended surface ships or

submarines, the type of grid used in the calculations is also important because it affects the amount of computer storage required. Methods which allow the use of unstructured grids (e.g. finite element solvers) typically require far greater computer storage than methods which require structured grids. Thus, there is a trade-off between the ease generating the grid, and the storage requirements. To address this problem, TRANSOM has been designed as a multi-block multi-method solver. The numerical grid may consist of several blocks, some of which are structured, some unstructured. On each block a solver appropriate to the block structure is used. Since the areas of complex geometry are usually small in comparison with the whole ship, it should be possible to reduce the computer storage requirements significantly.

The current version of TRANSOM includes two different solution methods which can be used on the blocks. The first is based on the program NSI2D developed for DREA under contract to the University of Toronto[3,4,5]; it is suitable for use on a single structured block. The algorithm is a variant of the pseudo-compressibility method developed by Rogers and Kwak[6]. Currently only two-dimensional flow can be modelled; it will be extended to three dimensions in 1996. Turbulence may be modelled either by the Baldwin-Lomax model[7] or by the Chen-Patel variant of the k-ε model[8]. Implementation of the Baldwin-Barth model[9] is currently underway. details of the implementation of the pseudo-compressibility solver in TRANSOM are given in Reference 10.

The second solution is based on the finite element program MEF developed at Université Laval[11]; it is suitable for use on unstructured blocks. MEF was modified at DREA to model turbulence using a variant of the k-ε model[12]. The implementation of the finite element solver in TRANSOM is described in detail in this memorandum.

TRANSOM is written in C++ according to the principles of Object Oriented Programming. The overall design is described in Reference 13. It is recommended that Reference 13 be read before this memorandum since it describes many of the basic classes used in the finite element solver.

Three main groups of classes are used to implement the finite element solver: elements, classes to organize the degrees of freedom associated with nodes of the unstructured grid, and finite element solvers. After a brief overview of the finite element method in Section 2, class hierarchies for each of these objects are described in Sections 3 to 8. Section 9 contains recommendations for improvements to the program. A user's guide for the program is also provided in Reference 14.

TRANSOM input and output files use the OFFSRF format described in Reference 15. Reference 16 describes C++ classes which implement specialized input and output streams for OFFSRF files and a base class which is inherited by all classes which read from or write to OFFSRF files; it should be read prior to this document for a proper understanding of TRANSOM input and output.

2 An Overview of the Finite Element Method

In this section we give a brief overview of the finite element method as used in TRANSOM. The main purpose is to establish the notation that will be used. There are many textbooks which describe the method in much more detail and generality.

A system of equations is to be solved on a grid of points which are grouped into elements. At each node of the grid there are several degrees of freedom. The values of most of the degrees of freedom are unknown and must be determined by the solution method. For simplicity in the discussion below, we will assume that a single equation must be solved over the whole grid. This is the typical case for fluid flow, although it is not necessary in general, nor in TRANSOM. The system of equations to be solved is written:

$$Fv = f \tag{2.1}$$

where F is a differential operator and v is a vector-valued variable which must be determined.

In TRANSOM the Galerkin method is used. On each element, the unknown v is determined by its values at the nodes of the element:

$$v(x,y) = \sum_{n=1}^{N} v_n N_n(x,y) \tag{2.2}$$

where $N_n(x,y)$ are the nodal functions for the element. The values of v_n are the degrees of freedom for the problem. Variations of the degrees of freedom are defined by

$$\delta v(x,y) = \sum_{n=1}^{N} \delta v_n N_n(x,y) \tag{2.3}$$

The Galerkin method obtains an approximate solution to equation (2.1) by requiring that

$$\int_V \delta v \left(Fv - f\right) dV = 0 \tag{2.4}$$

for any choice of variations δv_n. Substituting equations (2.2) and (2.3) one obtains a linear system to be solved for the vector of unknowns $\{v\}$.

$$[K]\{v\} = \{f\} \tag{2.5}$$

where

$$[K]_{nm} = \int_V N_n(x,y)\, FN_m(x,y)\, dV; \qquad \{f\}_n = \int_V N_n(x,y)\, f\, dV \tag{2.6}$$

For a non-linear problem, $[K]$ and $\{f\}$ may be functions of the degrees of freedom $\{v\}$.

The integrals in equation (2.6) are decomposed into integrals over each of

the N_e elements:

$$[K]_{nm} = \sum_{k=1}^{N_e} \left[K_{E_k}\right]_{nm}; \quad \{f\}_n = \sum_{k=1}^{N_e} \left\{f_{E_k}\right\}_n \tag{2.7}$$

with

$$\left[K_{E_k}\right]_{nm} = \int_{E_k} N_n(x,y)\, FN_m(x,y)\, dV; \quad \left\{f_{E_k}\right\}_n = \int_{E_k} N_n(x,y)\, f\, dV \tag{2.8}$$

The matrix $\left[K_{E_k}\right]$ has only N^2 non-zero elements, where N is the total number of degrees of freedom for the element; hence, it can be stored in a compact $N \times N$ matrix which will be called the local left hand side matrix, or simply the local matrix. Similarly, $\left\{f_{E_k}\right\}$ can be stored as a vector of length N known as the local right hand side vector. The process of adding the local matrix and local vector to the linear system, as in equation (2.7), is called assembling the linear system.

Boundary conditions are often implemented by fixing the value of one of the degrees of freedom at a node. When this is done, the variation of that degree of freedom is zero and one equation is eliminated from the linear system. Suppose that the n^{th} degree of freedom is to be fixed. Then row n must be eliminated from $[K]$ and column n is eliminated by multiplying it by the known value of the n^{th} degree of freedom and incorporating the result in the vector $\{f\}$. The size of $[K]$ is therefore the number of degrees of freedom whose values are not known.

3 Elements

The basis of the finite element method is the decomposition of a complex problem into a large number of simple problems on single elements. The inheritance characteristics of an Object Oriented programming language are particularly useful for this problem.

Figure 1 shows the complete class hierarchy for the classes used by TRANSOM to represent elements. The fat arrows denote an inheritance relation (i.e. a RefElement is a specialization of an ElementNodeOrg and inherits its public and protected member functions) and the thin arrows denote a "part of" relationship.

3.1 Element Node Organizers

Since an element is made up of a sequence of nodes, we define a node organizer to describe the nodes in an element (see Reference 13, Section 2.3 for a general description of node organizers and of the class NodeOrg); it is called an ElementNodeOrg. The prototype of its constructor is:

ElementNodeOrg(unsigned n);

Creates an ElementNodeOrg for an element with n nodes.

Notice that when the ElementNodeOrg is constructed, the values of its nodes are unspecified.

Like all node organizers, an ElementNodeOrg has the member function num_nodes() to return the number of nodes in the element, and get_all_node_iter() to return an iterator over the nodes in the element. The syntax for the iterator is as described in Reference 13, Section 2.2. The following member function is also defined:

int contains(Node n) const;

Returns true if n is one of the nodes governed by the ElementNodeOrg.

3.2 Quadrature Rules

To facilitate interpolation and integration over an element, every element may be transformed to a reference element which has a simple geometry. The coordinate system of the reference element will be denoted (ξ,η), while that of the element itself is denoted (x,y). For simplicity we will assume two dimensions in this discussion, although TRANSOM can represent elements of one, two, or three dimensions.

Each reference element also provides a means of integrating a variable over the element. The quadrature rule used is of the form

$$\int f(\xi,\eta)d\xi d\eta = \sum_{m=1}^{M} w_m f(\xi_m^*,\eta_m^*) \tag{3.1}$$

where w_m, ξ_m^*, and η_m^*, are all fixed values. Since for linear, square, and cubic reference elements these values are chosen to yield standard Gaussian quadrature rules, the w_m are called Gaussian weights and the (ξ_m^*,η_m^*) are called Gaussian points. The order of a quadrature rule is the highest order for which a polynomial is guaranteed to be integrated exactly (or more accurately, to within round-off error). For example, a quadrature rule of order four will integrate the polynomials ξ^3 and $\xi\eta^2$ exactly, while $\xi^4$and $\xi^2\eta^2$ will only be approximate.

In TRANSOM, the class QuadratureRule is used to represent quadrature rules. Each instance of a QuadratureRule stores a pointer to a Mtx, points, which contains the points (ξ_m^*,η_m^*), and a Vec, weights, containing the weights, w_m (the Vec and Mtx classes are described in Appendix A). Thus, (*points)[n] is pointer to the components of the n^{th} Gaussian point and (*weights)[n] is the corresponding weight. The number of points can be obtained using the function weights->len(). Both points and weights are public members.

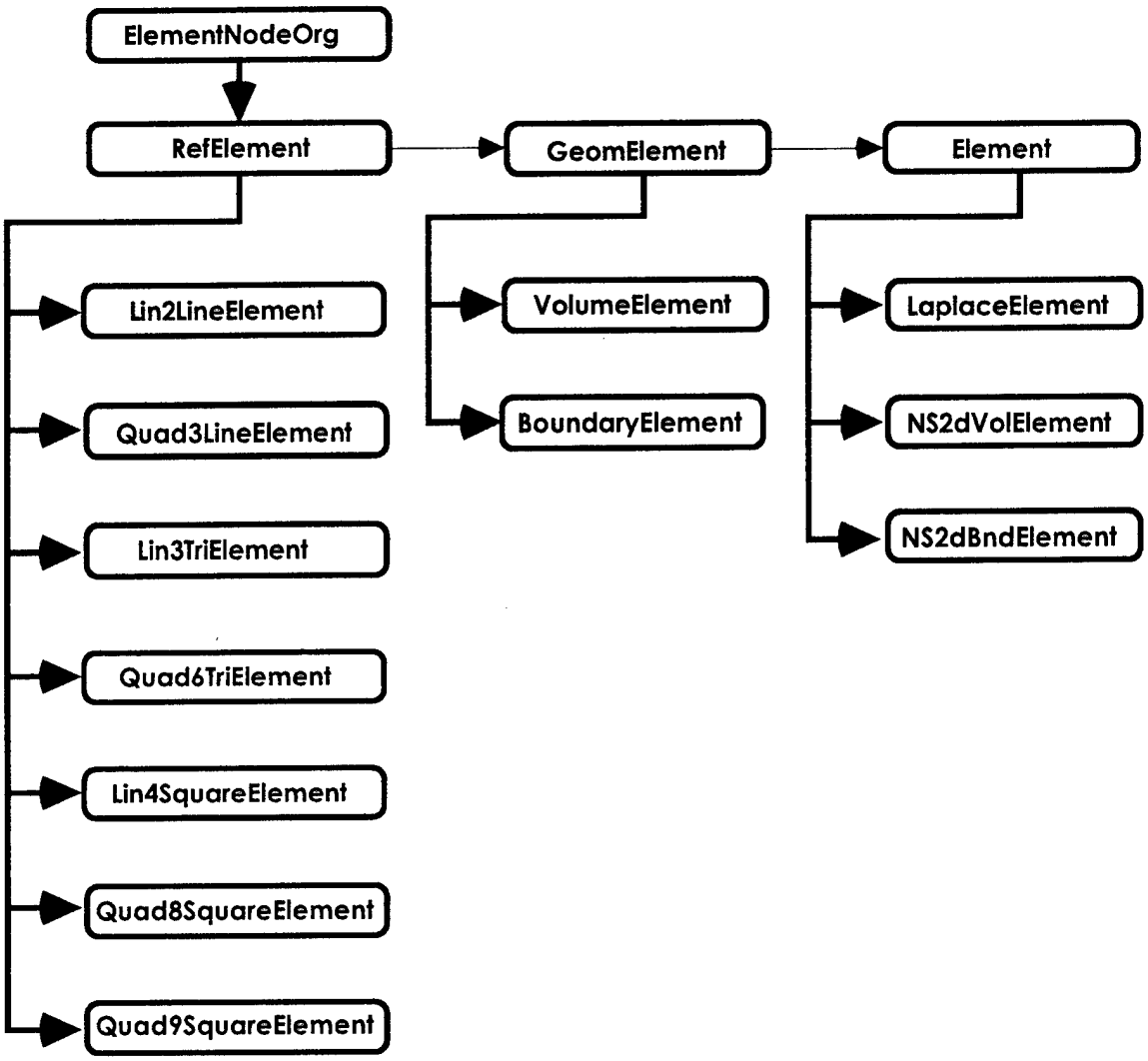


Figure 1: *Class hierarchies for the element classes in TRANSOM*

The order of the quadrature rule may be obtained using the following member function:

```
unsigned max_exact_order() const;
```

The QuadratureRule class also provides three member functions which perform the integration of equation (3.1). They differ in the way in which the function $f(\xi,\eta)$ is specified.

```
double integrate_fnc(double func(const double*, void *), void *p) const;
void integrate_fnc(void func(const double*, void*, Vec&), void *p, Vec &v) const;
void integrate_fnc(void func(unsigned, void*, Vec&), void *p, Vec &v) const;
```

In each of these functions, the argument p may be used to pass data to the function func; it is passed directly to func using its void* argument. Commonly p is the this pointer of the calling class cast to void*.

The first form of integrate_fnc() is suitable for integrating a single-valued function. The const double* argument to func is a pointer to the coordinate values (ξ,η).

The second form of integrate_fnc() is suitable for integrating a multiple-valued function; the values of the function are returned in the Vec v. The const double* argument to func is a pointer to the coordinate values (ξ,η).

The third form of integrate_fnc() is also suitable for integrating a multiple-valued function; the values of the function are returned in the Vec v. Rather than passing the coordinate values (ξ,η) to func, the number of the current Gaussian point, ig, is passed instead. The coordinate values used in equation (3.1) are obtained from (*points)[ig]. This form of integrate_fnc() can increase efficiency when more than one integration is performed over a single element. The values of variables at all the Gaussian points need only be calculated once and stored in an array. Given the index ig, the function func can retrieve the appropriate value when calculating $f(\xi,\eta)$.

3.2.1 Gaussian Quadrature

TRANSOM defines specializations of QuadratureRule which implement one-dimensional Gaussian quadrature of order two, four, six, and eight. They are GaussQuadRuleO2, GaussQuadRuleO4, GaussQuadRuleO6, and GaussQuadRuleO8, respectively. The corresponding rules in two dimensions are Gauss2DQuadRuleO2, Gauss2DQuadRuleO4, Gauss2DQuadRuleO6, and Gauss2DQuadRuleO8. The points and weights for Gaussian integration are well known. They are tabulated in Abramowitz and Stegun[17].

3.2.2 Quadrature over Equilateral Triangles

Quadrature rules are also defined for integration over an equilateral triangle centred at the origin and with one vertex placed at (1,0), as shown in Figure 2.

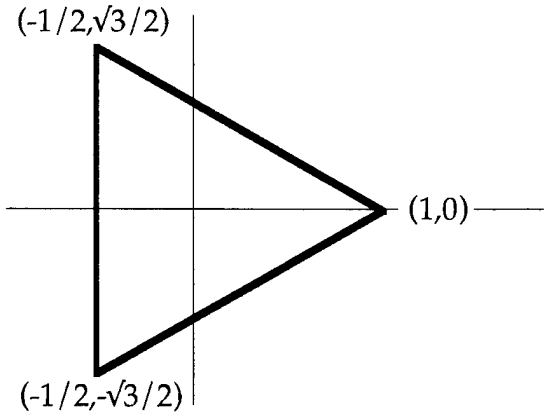
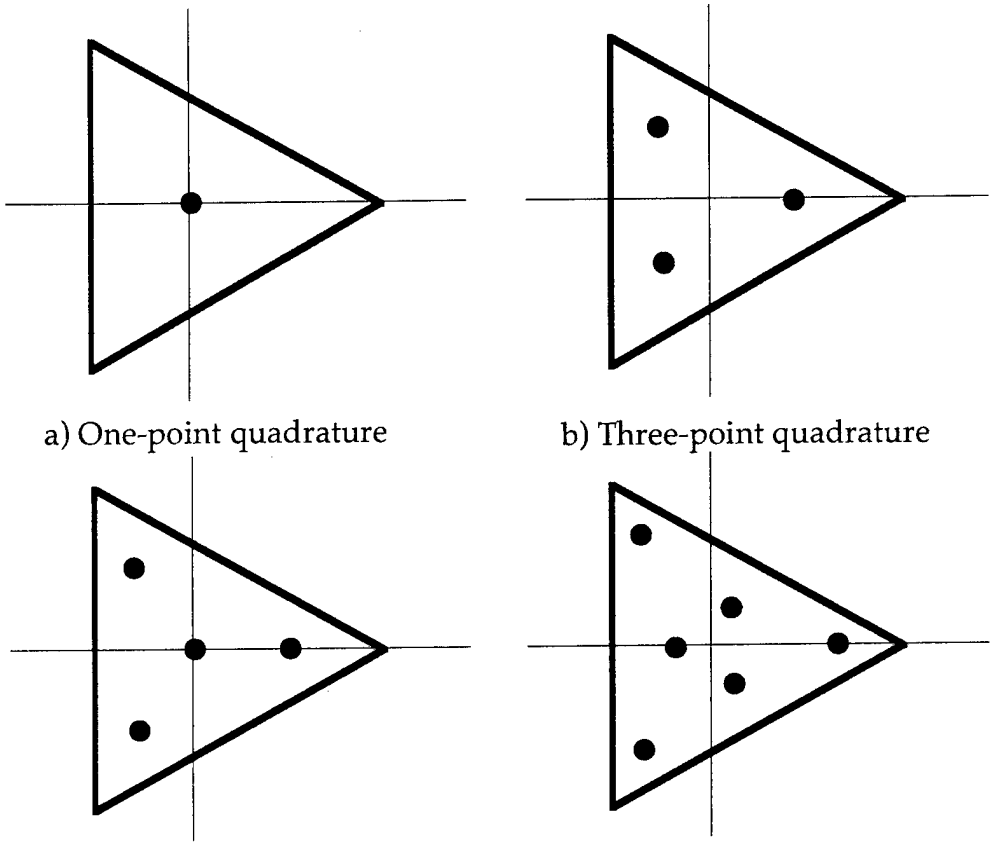


Figure 2: *The reference triangle for triangular reference elements*

There are quadrature rules available for triangular reference elements. The locations of the quadrature points for each option are shown in Figure 3. The quadrature points and weights are given in Abramowitz and Stegun[17]. Table 1 gives the order of each of the quadrature rules.



a) One-point quadrature

b) Three-point quadrature

c) Four-point quadrature

d) Six point quadrature

Figure 3: *Quadrature points for integration over triangles*

Number of points	Max. Exact Order	Class Name
1	2	TriQuadRuleO2
3	3	TriQuadRuleO3
4	4	TriQuadRuleO4
6	6	TriQuadRuleO6

Table 1: *Accuracy of triangular quadrature schemes*

3.2.3 Adjustable Quadrature Rules

An adjustable quadrature rule is one whose order can be specified. They are represented by the class AdjQuadRule and its specializations. The AdjQuadRule class is a specialization of QuadratureRule. It is implemented as a list of QuadratureRules of different order. Only one of the QuadratureRules is active at any time. TRANSOM includes three specializations of AdjQuadRule:

1. AdjGaussQuadRule: implements an AdjQuadRule using the rules GaussQuadRuleO2, GaussQuadRuleO4, GaussQuadRuleO6, and GaussQuadRuleO8.
2. AdjGauss2DQuadRule: implements an AdjQuadRule using the quadrature rules Gauss2DQuadRuleO2, Gauss2DQuadRuleO4, Gauss2DQuadRuleO6, and Gauss2DQuadRuleO8.
3. AdjTriQuadRule: implements an AdjQuadRule using the rules TriQuadRuleO2, TriQuadRuleO3, TriQuadRuleO4, and TriQuadRuleO6.

In addition to the member functions which it inherits from QuadratureRule, AdjQuadRule has the following constructors and member functions.

AdjQuadRule(unsigned ord);
: Creates an AdjQuadRule. The member function set_max_order() (see below) is called to set the order of the quadrature rule to ord. Since AdjQuadRule is a pure virtual class, this constructor is only available to specializations of the class. The constructors for all the specializations have similar arguments.

unsigned set_max_order(unsigned ord);
: Requests that the order of the quadrature rule be set to ord. Depending on the rules available, the actual order used may be greater or smaller than ord. The order used is returned by the function.

3.3 Reference Elements

For each reference element, the position of the nodes in (ξ,η) coordinates is fixed. For an N-noded element these reference nodes will be denoted (ξ_n,η_n). The reference element also defines N nodal functions $N_n(\xi,\eta)$ which have the property that

$$N_n(\xi_m,\eta_m) = \delta_{nm} \tag{3.2}$$

where δ_{nm} is the Kronecker delta. If the values of a variable at each of the nodes is v_n, then its value anywhere in the element can be determined using the interpolation rule

$$v(\xi,\eta) = \sum_{n=1}^{N} v_n N_n(\xi,\eta) \tag{3.3}$$

Each reference element also provides an adjustable quadrature rule for performing integration over the element. Member functions are provided to return a pointer to the quadrature rule and to change its order. Since the Gaussian points and weights of the quadrature rule are public, they are available to any class which might require them to perform efficient evaluations of a function to be integrated.

In TRANSOM, reference elements are represented by the virtual class RefElement which is derived from the class ElementNodeOrg. The member functions num_nodes(), get_all_node_iter(), and contains() are all inherited. The following public constructors and member functions are also defined.

RefElement(unsigned n);
Makes a reference element having n nodes.

Node& operator[](unsigned n);
const Node& operator[](unsigned n) const;
Returns the n^{th} node of the reference element. Thus, if re is a RefElement, re[n] is the number of its n^{th} node

virtual unsigned max_order() const = 0;
Returns the maximum order of any component (ξ or η) of the polynomials used to define $N_n(\xi,\eta)$: e.g. the term $\xi^2\eta^2$ is of order 3 not 5.

virtual unsigned dim_element() const = 0;
Returns the dimensions of the element, d: i.e. the dimensions of the reference coordinate system. In the examples shown here d is two, since there are two reference coordinates, ξ and η, but TRANSOM will handle elements of one, two, or three dimensions.

virtual const Mtx& get_ref_coords() const = 0;
Returns an $N \times d$ matrix which contains the reference coordinates of the nodes, (ξ_n,η_n). The Mtx class is described in Appendix A. If rc is the Mtx returned, then rc[n][0] is the value of ξ_n and rc[n][1] is the value of η_n.

virtual AdjQuadRule* get_quadrature_rule() = 0;
Returns a pointer to the quadrature rule used by this reference element.

virtual const Array<Vec*>& get_node_funcs_at_gp() const = 0;
Returns an array of pointers to vectors containing the values of nodal functions at the Gaussian points: $N_n(\xi_m^*,\eta_m^*)$. The array is of length M and each vector is of length N. If nf is the array returned, then (*nf[m])[n] is the value of $N_n(\xi_m^*,\eta_m^*)$. This function could have been written to return a Mtx, but the Array<Vec*> was chosen to

correspond more closely with the returned value of get_grad_node_funcs_at_gp(). Note that the returned value cannot be Array<Vec>& since an Array can only be constructed from classes which have constructors without arguments; all the Vec constructors require arguments (see Appendix A.1).

virtual const Array<Mtx*>& get_grad_node_funcs_at_gp() const = 0;

Returns an array of pointers to $N \times d$ matrices containing the gradient with respect to the reference coordinates of the nodal functions at the Gaussian points: i.e. returns $\nabla N_n(\xi_m^*, \eta_m^*)$. The array is of length M. If g is the array returned, then (*g[m])[n][0] is the value of $(\partial N_n/\partial\xi)(\xi_m^*, \eta_m^*)$ and (*g[m])[n][1] is the value of $(\partial N_n/\partial\eta)(\xi_m^*, \eta_m^*)$.

virtual const Array<Mtx*>& get_grad_node_funcs_at_nodes() const = 0;

Returns an array of pointers to $N \times d$ matrices containing the gradient with respect to the reference coordinates of the nodal functions at the nodes: i.e. returns $\nabla N_n(\xi_m, \eta_m)$. The array is of length N. If g is the array returned, then (*g[m])[n][0] is the value of $(\partial N_n/\partial\xi)(\xi_m, \eta_m)$ and (*g[m])[n][1] is the value of $(\partial N_n/\partial\eta)(\xi_m, \eta_m)$.

void values_at_gp(const IOVar &v, Mtx &vgp);

Using equation (3.3), evaluates the values of the variables in v at each of the Gaussian points. Returns the values in the $M \times L$ matrix vgp, where M is the number of Gaussian points and L is the number of variables per node in v. If the dimensions of vgp are incorrect a fatal error results.

void values_at_gp(double *d, Vec &vgp);

The argument d is a pointer to the values of a variable at each of the nodes of the element. Determines the corresponding values at each of the Gaussian points using equation (3.3). Returns the values in the vector vgp of length M.

Vec node_funcs(const Vec &xi, int *kder = 0);

Evaluates the nodal functions or their derivatives at the point xi and returns them in a Vec of length N. The length of xi must be d or a fatal error will result. The d integers stored at kder are used to specify the derivative which is required. When d is two, kder[] = { 0,0 } signifies that the values of the nodal function are returned; kder[] = { 1,0 } signifies that values of $\partial N_n/\partial\xi$ are returned; kder[] = { 0,1 } signifies that values of $\partial N_n/\partial\eta$ are returned; and so on. If kder is null, the values of the nodal functions are returned.

void integrate_fnc(void func(unsigned, void*, Vec&), void *p, Vec &v);

Integrates the vector function func over the reference element using the quadrature formula of equation (3.1). The vector-valued integral is returned in v. The function func(ig,p1,f) returns its value in f; The argument ig is the number of the current Gaussian point; i.e. it is in

the range $[0,M]$. The pointer p may be used to pass data to func; the value of the func argument p1 is always set to p. Usually p is the this pointer of the calling class cast to a void. The length of the vector f is always the same as the length of v.

As an illustration of the member function integrate_fnc(), consider the following code. It integrates the squares of the nodal functions over an arbitrary reference element represented by the RefElement pointer re. The result is returned in the Vec v of length N. First we define func.

```
void func(unsigned ig, void *p, Vec &f)
{
  RefElement *re = (RefElement*)p;
  Vec &nf = *(re->get_node_funcs_at_gp()[ig]); // nf contains the nodal functions
                                               // at the Gaussian point ig
  for (unsigned n = 0; n < f.len(); n++)
    f[n] = nf[n]*nf[n];
}
```

A call to integrate_fnc() will now perform the integration. Notice that the argument p is set to re (cast to a void*) which is then passed to func.

```
Vec v(re->num_nodes());    // Create a Vec of appropriate size
integrate_fnc(func, (void*)re, v);
```

3.3.1 Line Reference Elements

Line reference elements are one-dimensional. The reference coordinate ξ extends from –1 to 1. An AdjGaussQuadRule is used for the quadrature rule. For all line reference elements the nodes are numbered consecutively along the line segment.

A Lin2LineElement is a line reference element whose nodal functions are linear. The element contains only two nodes at $\xi = -1$ and $\xi = 1$. The default quadrature order is four. The nodal functions are

$$N_1(\xi) = \frac{1-\xi}{2}; \quad N_2(\xi) = \frac{1+\xi}{2} \tag{3.4}$$

A Quad3LineElement is a line reference element whose nodal functions are quadratic. The element contains three nodes at $\xi = -1$, 0, and 1. The default quadrature order is eight. The nodal functions are:

$$N_1(\xi) = -\frac{(1-\xi)\xi}{2}; \quad N_2(\xi) = (1+\xi)(1-\xi); \quad N_3(\xi) = \frac{(1+\xi)\xi}{2} \tag{3.5}$$

a) Lin2LineElement b) Quad3LineElement

Figure 4: *Reference points for line elements*

3.3.2 Triangular Reference Elements

Triangular reference elements are two-dimensional elements whose reference coordinates (ξ,η) cover an equilateral triangle centred at the origin and with one vertex placed at (1,0), as shown in Figure 2. An AdjTriQuadRule is used for the quadrature rule.

A Lin3TriElement is a triangular reference element whose nodal functions are linear. The element contains three nodes: one at each vertex of the reference triangle. The default quadrature order is four.

A Quad6TriElement is a triangular reference element whose nodal functions are quadratic. The element contains six nodes: one at each vertex of the reference triangle and one at the centre of each of its sides. The default quadrature order is six. The nodes are numbered consecutively around the perimeter starting at one vertex of the triangle.

a) Lin3TriElement b) Quad6TriElement

Figure 5: *Reference points for triangular elements*

3.3.3 Square Reference Elements

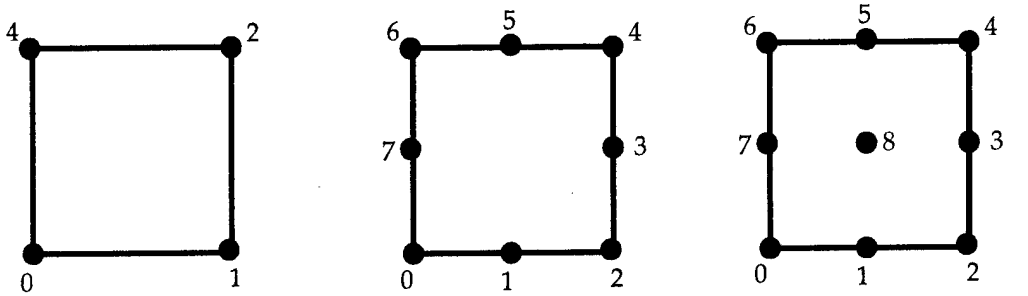
Square reference elements are two-dimensional elements whose reference coordinates (ξ,η) cover a square centred at the origin with sides of length two; thus its vertices are at $(\xi,\eta) = (\pm 1,\pm 1)$. An AdjGauss2DQuadRule is used for the quadrature rule.

A Lin4SquareElement is a square reference element whose nodal functions are bi-linear. The element contains four nodes: one at each vertex of the

reference square. The default quadrature order is four.

A Quad8SquareElement is a square reference element whose nodal functions are linear combinations of the polynomial basis functions 1, ξ, η, ξ^2, $\xi\eta$, η^2, $\xi^2\eta$, and $\xi\eta^2$. The element contains eight nodes: one at each vertex of the reference square and one at the centre of each of its sides. The default quadrature order is eight.

A Quad9SquareElement is a square reference element whose nodal functions are bi-quadratic. They are linear combinations of the polynomial basis functions 1, ξ, η, ξ^2, $\xi\eta$, η^2, $\xi^2\eta$, $\xi\eta^2$, and $\xi^2\eta^2$. The element contains nine nodes distributed as a 3×3 grid. The default quadrature order is eight.



a) Lin4SquareElement b) Quad8SquareElement c) Quad9SquareElement

Figure 6: *Reference points for square elements*

3.4 Geometric Elements

TRANSOM elements are iso-parametric; the nodal functions for interpolation of the coordinates are the same as for any other variable.

The class GeomElement represents an iso-parametric element including the location of its nodes in space. None of the physical properties (i.e. the nodal degrees of freedom) are represented.

It is important to recognize the difference between the dimension of the element, d, and the dimension of the element coordinates, d_c. For a square element d is two, but when it is used in a three dimensional problem, d_c is three. There are two main types of geometric elements: volume elements, for which the d and d_c are the same; and boundary elements, for which d is less than d_c. These are represented by the classes VolumeElement and BoundaryElement, both of which are specializations of GeomElement.

Every instance of a GeomElement contains a RefElement which defines the organization of its nodes. The GeomElement defines the transformation between the Cartesian coordinates and the reference coordinates. Since the element is iso-parametric, this transformation is defined, using equation (2.2), by

$$x(\xi,\eta) = \sum_{n=1}^{N} x_n N_n(\xi,\eta) \tag{3.6}$$

where x is the Cartesian coordinate vector and the x_n are the coordinate vectors at the nodes.

Like a RefElement, a GeomElement provides functions for integrating over the element, but in this case the integration variables are the Cartesian coordinates, not the reference coordinates. The integration is performed by converting the integral to one over the reference coordinates by multiplying by the determinant of the Jacobian. If $d = d_c = 2$ we have

$$\int f(x,y)\,dxdy = \int f(\xi,\eta)\,\left|\det J(\xi,\eta)\right|\,d\xi d\eta \tag{3.7}$$

where the Jacobian J is given by

$$J(\xi,\eta) = \frac{\partial(x,y)}{\partial(\xi,\eta)} = \begin{bmatrix} \dfrac{dx}{d\xi} & \dfrac{dy}{d\xi} \\ \dfrac{dx}{d\eta} & \dfrac{dy}{d\eta} \end{bmatrix} \tag{3.8}$$

Using equation (3.6), one gets the following formula for $\det J$:

$$\det J(\xi,\eta) = \sum_{n=1}^{M}\sum_{m=1}^{M} x_n y_m \left(\frac{dN_n}{d\xi}\frac{dN_m}{d\eta} - \frac{dN_n}{d\xi}\frac{dN_m}{d\eta} \right) \tag{3.9}$$

Now using equations (3.1) and (3.7), one obtains the following quadrature formula:

$$\int f(x,y)dxdy = \sum_{m=1}^{M} W_m f(\xi_m^*,\eta_m^*) \tag{3.10}$$

where

$$W_m = w_m \left|\det J(\xi_m^*,\eta_m^*)\right| \tag{3.11}$$

If the value of a variable at the n^{th} node is v_n, then its gradient anywhere in the element can be determined using the interpolation rule

$$\nabla v(\xi,\eta) = \sum_{n=1}^{N} v_n \nabla N_n(\xi,\eta) = \sum_{n=1}^{N} v_n \begin{bmatrix} \dfrac{\partial N_n}{\partial x} \\ \dfrac{\partial N_n}{\partial y} \end{bmatrix} = \sum_{n=1}^{N} v_n J^{-1} \begin{bmatrix} \dfrac{\partial N_n}{\partial \xi} \\ \dfrac{\partial N_n}{\partial \eta} \end{bmatrix} \tag{3.12}$$

For a boundary element with $d = 1$, an integral over the element is equivalent to a line integral. The quadrature formula is similar to equation (3.10) but with

$$W_m = w_m \left| \frac{dx}{d\xi}(\xi_m^*) \right| = w_m \left| \sum_{n=1}^{M} x_n \frac{dN_n}{d\xi}(\xi_m^*) \right| \tag{3.13}$$

A GeomElement does not store the values of the coordinates at each of its nodes; that would be far too costly in memory. Instead, the coordinates for all nodes are stored as a PointColl (see Reference 13, Section 2.4); the GeomElement stores a static pointer to the coordinates along with a pointer to a RefElement which defines its nodes. The pointer to the RefElement is defined when the GeomElement is constructed.

```
GeomElement(RefElement *re);
```

The pointer to the coordinates is set using a static member function.

```
static void set_coords(PointColl *pc);
```

This allows GeomElements on different blocks to be used (set_coords() is called whenever the block is changed) without requiring the storage of an extra pointer in every GeomElement.

The following member functions all return the same values as the corresponding member functions of the RefElement re used to construct the GeomElement.

```
unsigned dim_element() const;
unsigned num_nodes() const;
AllNodeIter *get_all_node_iter() const;
AdjQuadRule* get_quadrature_rule();
const Array<Vec*>& get_node_funcs_at_gp() const;
const Array<Mtx*>& get_grad_node_funcs_at_gp() const;
const Array<Mtx*>& get_grad_node_funcs_at_nodes() const;
void values_at_gp(const IOVar &v, Mtx &vgp);
Node& operator[](unsigned i);
const Node& operator[](unsigned i) const;
```

In addition, the following member functions are defined.

`unsigned dim_coords() const;`
: Returns the dimension of the coordinates used by the element. Note, as discussed above, that this may be different from the dimension of the element.

`double* coord(unsigned n) const;`
: Returns a pointer to the coordinates of the n^{th} node of the element. Thus the x value of the second node of a GeomElement ge is ge.get_coords(2)[0].

`RefElement* get_reference_element() const;`
: Returns a pointer to the reference element which defines the nodes of the element.

`operator RefElement ();`
: Converts the GeomElement to the RefElement which defines its nodes.

`void integrate_fnc(void func(unsigned, void*, Vec&), void *p, Vec &v);`
: Integrates the vector function func over the element using the quadrature formula of equation (3.10) in conjunction with either equation (3.11) or equation (3.13). The RefElement quadrature rule is used to perform the integration.

An extractor and an inserter are defined for reading from an istream and writing to an ostream.

```
friend istream& operator>>(istream &in, GeomElement &ge);
friend ostream& operator<<(ostream &out, GeomElement &ge);
```

These functions simply read and write the reference element associated with the geometric element. They are equivalent to

```
in >> *ge.get_reference_element();
```

and

```
out << *ge.get_reference_element();
```

3.4.1 Volume Elements

A VolumeElement is a geometric element for which the dimensions of the element are the same as the dimensions of the coordinates. This implies that the reference coordinates and Cartesian coordinates are related by a square Jacobian matrix.

A Volume2dElement is a specialization of a VolumeElement for which the dimension of the element is two. Its member functions are all inherited from VolumeElement. The prototypes for the VolumeElement and Volume2dElement constructors are:

```
VolumeElement(RefElement *e);
Volume2dElement(RefElement *e);
```

The following member functions are inherited from GeomElement.

```
unsigned dim_element() const;
unsigned num_nodes() const;
AllNodeIter *get_all_node_iter() const;
AdjQuadRule* get_quadrature_rule();
const Array<Vec*>& get_node_funcs_at_gp() const;
const Array<Mtx*>& get_grad_node_funcs_at_gp() const;
const Array<Mtx*>& get_grad_node_funcs_at_nodes() const;
void values_at_gp(const IOVar &v, Mtx &vgp);
Node& operator[](unsigned i);
const Node& operator[](unsigned i) const;
unsigned dim_coords() const;
double *coord(unsigned n) const;
RefElement* get_reference_element() const;
operator RefElement ();
void integrate_fnc(void func(unsigned, void*, Vec&), void *p, Vec &v);
Mtx get_cart_grads_at_gp(unsigned ig);
```

The main function of the VolumeElement class is to provide a means for calculating the Jacobian of the transformation between Cartesian coordinates and the coordinates of the reference element. This is done by the following two member functions.

```
virtual void jacob(const Mtx &gnf, SqMtx &jmtx, SqMtx &jinv, double &detj) = 0;
```

Calculates the Jacobian of the transformation between Cartesian and reference coordinates, returning it in jmtx. Its determinant and its

inverse are also calculated and are returned in detj and jinv respectively. The input argument gnf is an $N \times d$ matrix containing the gradients (with respect to reference coordinates) of the node functions at the point where the Jacobian is to be evaluated; thus gnf[n][1] is the value of $\partial N_n / \partial \eta$. Since this is a pure virtual function, it must be defined by specializations of VolumeElement. The class Volume2dElement provides a concrete version of this function.

void jacob_at_gp(unsigned ig, SqMtx &jmtx, SqMtx &jinv, double &detj);
Calculates the Jacobian, its inverse, and its determinant at the igth Gaussian point.

3.4.2 Boundary Elements

A BoundaryElement is a GeomElement for which the dimension of the element is one less than the dimension of the coordinates. This implies that the reference coordinates and Cartesian coordinates are related by a Jacobian matrix which is not square.

A Boundary2dElement is a specialization of a BoundaryElement for which the coordinate dimension is two and the element dimension is one. Its member functions are all inherited from BoundaryElement. The prototypes for the BoundaryElement and Boundary2dElement constructors are:

```
BoundaryElement(RefElement *e);
Boundary2dElement(RefElement *e);
```

The following member functions are inherited from GeomElement.

```
unsigned dim_element() const;
unsigned dim_coords() const;
unsigned num_nodes() const;
AllNodeIter *get_all_node_iter() const;
AdjQuadRule* get_quadrature_rule();
const Array<Vec*>& get_node_funcs_at_gp() const;
const Array<Mtx*>& get_grad_node_funcs_at_gp() const;
const Array<Mtx*>& get_grad_node_funcs_at_nodes() const;
void values_at_gp(const IOVar &v, Mtx &vgp);
Node& operator[](unsigned i);
const Node& operator[](unsigned i) const;
double *coord(unsigned n) const;
RefElement* get_reference_element() const;
operator RefElement ();
void integrate_fnc(void func(unsigned, void*, Vec&), void *p, Vec &v);
Mtx get_cart_grads_at_gp(unsigned ig);
```

The BoundaryElement class has two main functions: it provides a means to calculate the scaling factor $|dx/d\xi|$ used in equation (3.13) when integrating over the element; and it provides a means to calculate the normals to the element which are often required by surface integrals after an integration by parts (see, for example, equation (6.4)) This is done by the following member functions.

virtual void jacobs(const Mtx &gnf, double &detj, Vec &normal) = 0;

At a single point, calculates the scaling factor $|dx/d\xi|$ used in equation (3.13) and the outward pointing normal to the element. The input argument gnf is an $N \times d_e$ matrix containing the gradients (with respect to reference coordinates) of the node functions at the point where the scaling factor and normals are to be evaluated; thus gnf[n][0] is the value of $\partial N_n/\partial \xi$. Since this is a pure virtual function, it must be defined by specializations of BoundaryElement. The class Boundary2dElement provides a concrete version of this function.

void jacobs_at_gp(unsigned ig, double &detj, Vec &normal);

Performs the calculations in jacobs at the igth Gaussian point.

virtual Array<Vec*> &get_normals_at_gp() = 0;

Returns the normals at each Gaussian point in an array of length M each of whose elements is a pointer to a Vec of length d. Thus, if normals is the array returned, then (*normals[n])[1] is the y-component of the nth Gaussian point of the element's quadrature rule. Since this is a pure virtual function, it must be defined by specializations of BoundaryElement. The class Boundary2dElement provides a concrete version of this function.

virtual Array<Vec*> &get_normals_at_nodes() = 0;

Calculates the normals at each node and returns them in an array of length N each of whose elements is a pointer to a Vec of length d. Thus, if normals is the array returned, then (*normals[n])[1] is the y-component of the nth node of the element. Since this is a pure virtual function, it must be defined by specializations of BoundaryElement. The class Boundary2dElement provides a concrete version of this function.

3.5 The Element Class

An Element is a finite element which represents not only the geometry of the element, but also the degrees of freedom used to describe the physical model. The geometry of the element is obtained from a GeomElement. Typically values for the degrees of freedom are obtained from global IOVars (see Reference 13, Section 3.4), but this is left up to specializations of the class. The prototype for the Element constructor is:

Element(GeomElement *ge);

Makes an element with geometry given by ge.

The following two member functions allow access to the geometric element and reference element from which the element was constructed.

GeomElement* get_geometric_element() const;

Returns the geometric element on which the element is based.

RefElement* get_reference_element() const;

Returns the reference element on which the element is based.

The following member functions all simply execute the corresponding function for the GeomElement from which the Element was constructed.

```
unsigned num_nodes() const;
unsigned dim_coords() const;
unsigned dim_element() const;
AdjQuadRule* get_quadrature_rule();
double *coord(unsigned n) const;
Node& operator[](unsigned i);
const Node& operator[](unsigned i) const;
```

The main function of the Element class is to calculate the local matrix and right hand side vector for assembly into the global linear system. This is done using the following member functions.

virtual unsigned deg_of_freedom(unsigned n) = 0;
: Returns the number of degrees of freedom at the n^{th} node.

virtual SqMtx* contribution_to_lhs();
: Computes the local left hand side matrix for the element. If there is no contribution to the global matrix from this element, a null pointer is returned. The calling class is responsible for deleting the returned matrix.

virtual Vec* contribution_to_rhs();
: Computes the direct contribution of the element to the right hand side of the linear system. If there is no contribution to the right hand side from this element, a null pointer is returned. The calling class is responsible for deleting the returned vector.

For unsteady problems, the local mass matrix is an important part of the local left hand side matrix. The mass matrix is defined by

$$[M]_{nm} = \int_V N_n(x,y)\, N_m(x,y)\, dV \tag{3.14}$$

The Element class provides a member function to compute the mass matrix. It is not used in the current version of TRANSOM.

virtual SqMtx* mass();
: Computes the local mass matrix of the element.

An element may also have extra properties required for the evaluation of the local matrix and local right hand side vector. In TRANSOM the element properties are used to set the imposed pressure and shear stress on boundary elements: see Section 6.1. The following member functions may be used to determine the number of element properties and to set their values.

virtual int num_properties() const;
: Returns the number of physical properties associated with the element.

virtual void set_properties(const Vec &v);
: Use the values in the v to set the properties of the element. If the length of v is not equal to the number of element properties, a fatal

error results.

An inserter and extractor are also defined. They simply call the inserter and extractor for the geometric element on which the element is based. Thus the extractor and inserter read in and write out the numbers of the nodes for the element.

4 Finite Element Grids

A finite element grid is a PointColl (see Reference 13, Section 2.4) in which the nodes are organized as a collection of reference elements. Like all PointColls, it has a node organizer which imparts the structure to the grid. The class used to represent the node organizer is called FENodeOrg. The grid itself is represented by the class FEBlock.

4.1 The FENodeOrg Class

The FENodeOrg class represents a node organizer for a finite element block. It is derived from the class NodeOrg described in Reference 13, Section 2.3. It has the following two constructors.

FENodeOrg(unsigned n);
: Constructs a node organizer which governs n nodes. In the local node numbering the first node will be node zero.

FENodeOrg(unsigned n, const Node &f);
: Constructs a node organizer which governs n nodes. In the local node numbering the first node will be node f.

An FENodeOrg stores an array of pointers to RefElements. The following member functions allow access to this array.

void set_num_elems(unsigned n);
: Sets the length of the array of reference elements.

int num_elems() const;
: Returns the number of reference elements.

RefElement* get_element(unsigned n) const;
: Returns a pointer to the n^{th} reference element.

An extractor is also defined for the FENodeOrg and OFFSRF_ifstream classes. It reads an ELEMENT record from an OFFSRF file (see References 15 and 16) and uses the data there to define the reference elements in the grid. Thus, if fe_norg is an FENodeOrg and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> fe_norg;
```

reads the next sequence of ELEMENT records in the OFFSRF file associated with the stream in_stream.

The format of the ELEMENT record is as follows.

```
{ELEMENT: number-of-elements
 {Element-Type-1
  node numbers for first element of type Element-Type-1
  node numbers for second element of type Element-Type-1
      ⋮
 }Element-Type-1
      ⋮
 {Element-Type-n
  node numbers for first element of type Element-Type-n
  node numbers for second element of type Element-Type-n
      ⋮
 }Element-Type-n
}ELEMENT
```

The following element types are currently defined.

L2LE: The element is a Lin2LineElement.

Q3LE: The element is a Quad3LineElement.

L3TE: The element is a Lin3TriElement.

Q6TE: The element is a Quad6TriElement.

L4SE: The element is a Lin4SquareElement.

Q8SE: The element is a Quad8SquareElement.

Q9SE: The element is a Quad9SquareElement.

For example, the following record defines three linear square elements and two linear line elements.

```
{ELEMENT: 5
 {L4SE
  0  1  5  4
  1  2  6  5
  2  3  7  6
 }L4SE
 {L2LE
  4  0
  3  7
 }L2LE
}ELEMENT
```

An ElementIter is an iterator over the elements governed by an FENodeOrg. It obeys the syntax used by all TRANSOM iterators as described in Reference 13, Section 2.2. The prototype of its constructor is:

```
ElementIter(FENodeOrg &no);
```

4.2 Finite Element Blocks

The class FEBlock represents a finite element block. It is a PointColl whose node organizer is an FENodeOrg. Its constructors have the following prototypes.

FEBlock(unsigned nd, unsigned n);
Construct a finite element block having n nodes. The dimension of the coordinates is nd.

FEBlock(unsigned nd, unsigned n, const Node &f, double *d);
Construct a finite element block with n nodes of dimension nd. Numbering of the nodes begins at f and the coordinate values are stored at memory location d.

The following member functions are defined for the FEBlock.

FENodeOrg* get_node_org() const;
Returns a pointer to the node organizer governing the block

int num_elems() const;
Returns the number of reference elements in the block.

An extractor is also defined for the FEBlock and OFFSRF_ifstream classes. It combines the extractors of the FENodeOrg class and the PointColl class; thus, it reads both ELEMENT and COORDINATE records from an OFFSRF file and uses them to define the reference elements and coordinates of the grid. Thus, if feb is an FEBlock and in_stream is an OFFSRF_ifstream, then the code

```
in_stream >> feb;
```

reads the next sequence of ELEMENT and COORDINATE records in the OFFSRF file associated with the stream in_stream. The format for the ELEMENT record is described in Section 4.1 and the format of the COORDINATE record is described in Reference 13, Section 2.4.

5 Finite Element Solvers

TRANSOM provides three finite element solvers. An FE2dNSSolver solves the two-dimensional steady laminar Navier-Stokes equations. An FESfncSolver uses a given incompressible velocity field to calculate a streamfunction for the flow (the streamfunction is very useful when displaying a flow since its iso-contours are streamlines). Finally, an FELaplaceSolver solves Laplace's equation; it can be used to calculate potential flow.

The sweep for each of these solvers is derived from the class FESweep, which handles the assembly of the linear system of equations and it solution. The main function of the specialized solvers is to convert the reference elements in the grid to elements appropriate for the problem to be solved.

The class hierarchy for the finite element solvers is shown in Figure 7.

5.1 Degrees of Freedom

As discussed in Section 2, there are degrees of freedom (DOFs) associated with every node. The set of all DOFs will be called the total DOFs. Some of the DOFs are unknowns to be determined by the solver; they are called free DOFs. Others have known values; they are called fixed DOFs. It is the job of

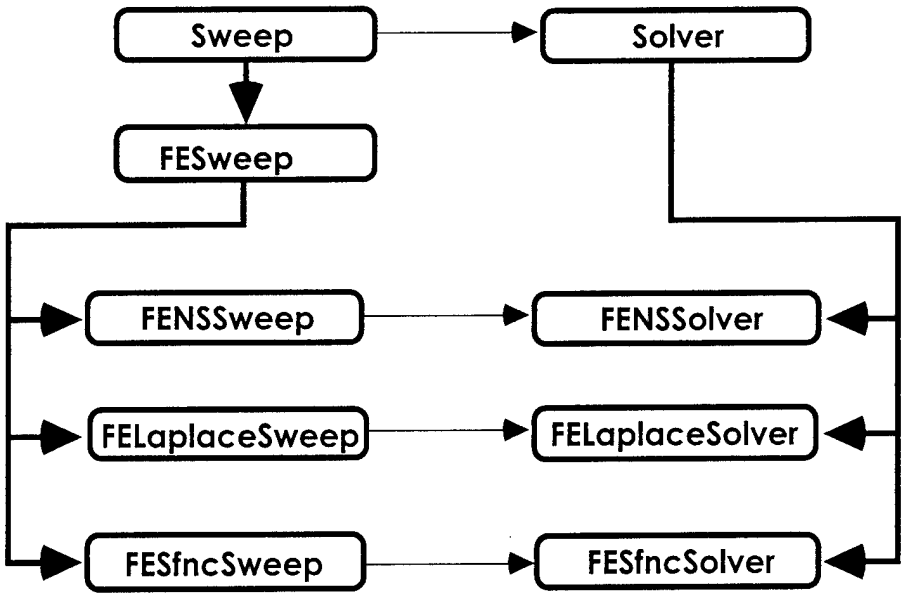


Figure 7: *Class hierarchy for finite element solvers and sweeps.*

the DataRep class to keep track of which DOFs are fixed, which are free, and which DOFs are associated with which node.

TRANSOM uses three different indices for DOFs.

1. A DOF is an index for the total DOFs. Its value will be in the range zero to $N_t - 1$, where N_t is the total number of DOFs.
2. A NodeDOF is an index for the DOFs associated with a single node. Its value will be in the range zero to $N_n - 1$, where N_n is the number of DOFs associated with the node. Note that N_n may be different for different nodes within the same element. The iterator class ElementDOFIter may be used to iterate over the node DOFs.
3. A FreeDOF is an index for the free DOFs. Its value will be in the range zero to $N_f - 1$, where N_f is the total number of free DOFs. The FreeDOF indices are not actually used by DataReps; instead, they are provided as a service to the classes FESolnMtx and FESolnVec which are used to represent the global linear system of equation (2.5).

The main task of the DataRep class is to provide mappings which convert between these three types of indices. An index for the fixed DOFs would also be possible but it is unnecessary.

The linear system of equation (2.5) is strongly dependent on the mappings provided by the DataRep. The indices for the matrix $[K]$ and the vector $\{f\}$ are FreeDOFs. The structure of the matrix can be altered by changing the

mapping between the DOF and FreeDOF indices. Use of well known methods such as the reverse Cuthill-McKee algorithm[18] can improve efficiency significantly by reducing the bandwidth of the global matrix. So far no such methods have been incorporated in TRANSOM.

TRANSOM provides four iterator classes for iterating over DOFs: TotalDOFIter, NodeDOFIter, ElementDOFIter, and ElementFreeDOFIter. Each uses the standard TRANSOM syntax for iterators described in Reference 13, Section 2.2.

A TotalDOFIter returns, in succession, each of the total DOFs. Casting a TotalDOFIter to a DOF returns the current DOF in the iteration. It is constructed from a DataRep.

```
TotalDOFIter(const DataRep &dr);
```

A NodeDOFIter returns, in succession, each of the total DOFs associated with a given node. Note that it returns DOF indices, not NodeDOF indices. Casting a NodeDOFIter to a DOF returns the current DOF in the iteration. A NodeDOFIter is constructed from a DataRep and a Node.

```
NodeDOFIter(const DataRep &dr, Node n);
```

In addition to the standard reset() function to restart the iteration, the following member function is also provided.

```
void reset(Node n);
```
Resets the iterator to return the DOFs for node n.

An ElementDOFIter returns, in succession, the total DOFs associated with all the nodes of an element. Casting an ElementDOFIter to a DOF returns the current DOF in the iteration. It is constructed from a DataRep and an Element.

```
ElementDOFIter(const DataRep &dr, const Element &e);
```

An ElementFreeDOFIter returns, in succession the free DOFs associated with all the nodes of an element. Casting an ElementFreeDOFIter to a FreeDOF returns the current FreeDOF in the iteration. It is constructed from a DataRep and an Element.

```
ElementFreeDOFIter(const DataRep &dr, const Element &e);
```

5.2 The DataRep Class

The main functions of the DataRep class were discussed in the previous section. Here the specific member functions which it provides are presented. A DataRep is constructed from an array of Elements.

```
DataRep(const Array<Element*> &elems);
```

The constructor uses the function Element::deg_of_freedom() to get the number of degrees of freedom for every node in every element of the array. If different elements report a different number of DOFs for the same node, a warning message is generated.

By default all DOFs are free. To fix DOFs the following member functions

must be called.

void add_fixed(Node n, NodeDOF d, double v);
Fixes the d^{th} DOF of node n, giving it the value v.

void set_DOF_map();
This member function must be called after all required DOFs have been fixed. It sets up maps between the different types of DOF. Calling add_fixed after set_DOF_map will result in a fatal error.

The following member functions are used to get information about the DOFs.

unsigned num_total() const;
Returns the total number of DOFs.

unsigned num_free() const;
Returns the number of free DOFs.

unsigned num_fixed() const;
Returns the number of fixed DOFs

int is_free(DOF d) const;
Returns true if DOF d is free.

int is_fixed(DOF d) const;
Returns true if DOF d is fixed.

FreeDOF free_DOF(DOF d) const;
Returns the FreeDOF corresponding to DOF d. A fatal error is generated if d is not free.

double& operator[](DOF n);
const double& operator[](DOF n) const;
Returns the value of the n^{th} total DOF.

5.3 The Linear System of Equations

The matrix $[K]$ and the vector $\{f\}$ in equation (2.5) are represented by the classes FESolnMtx and FESolnVec, respectively. The class FESolnMtx is a specialization of the class SkylineMtx which implements storage and inversion of square matrices using the skyline algorithm; for a description of this method of storage see Reference 11. The skyline profile above the diagonal is the same as the profile below the diagonal; this is always true for the matrices used in finite element calculations. The constructors and member functions of the SkylineMtx class are the following.

SkylineMtx();
Creates a skyline matrix of unspecified size. This is a protected constructor which is used by the class FESolnMtx.

SkylineMtx(const Array<unsigned> &prof):
Creates a skyline matrix with profile specified by prof. prof[n] is the height of the profile at the n^{th} row and column. The height does not include the term on the diagonal; thus, prof[n] must be in the range zero to n (row and column numbering begins at zero).

SkylineMtx(const SkylineMtx&);
A copy constructor.

int solve(Vec &rhs);
Solves the linear system whose right hand side is rhs. The solution is returned in rhs. This function first converts the matrix to LU-decomposed form. Subsequent calls to solve will work correctly (and more efficiently since the LU-decomposition need not be performed again), but other operations will yield incorrect results.

unsigned num_rows() const;
Return the number of rows in the matrix.

unsignbed num_cols() const;
Returns the number of columns in the matrix. Since the matrix is square, this function returns the same result as num_rows(). They are both included for compatibility with the Mtx class.

unsigned height(unsigned n) const;
Returns the height of the profile for the n^{th} row and column. The height does not include the term on the diagonal.

double& element(unsigned i, unsigned j);
Returns the matrix element (i,j) if it lies within the skyline profile; otherwise generates a fatal error.

double operator()(unsigned i, unsigned j) const;
Returns the matrix element (i,j). Correct values are returned for all elements of the matrix, whether within the skyline profile or not.

Vec operator*(const Vec &v);
Multiplies the vector v by the matrix.

void zero();
Sets all the elements of the matrix to zero. After this function is called the matrix is not considered LU-decomposed.

An FESolnMtx is a SkylineMtx with a member function which assembles a local element matrix into the skyline matrix. The n^{th} row of the skyline matrix is the approximation of the equation associated with the variation of the n^{th} free DOF, δv_n. The m^{th} column of the matrix is the coefficient which multiplies v_m in this equation. An algorithm to assemble a local matrix, $[K_e]$, of an element e is given in Figure 8.

An attempt to assemble an LU-decomposed matrix will result in a fatal error. The prototype for the member function is:

void assemble(const SqMtx&, const Element&);

An FESolnMtx is constructed from a DataRep and an array of pointers to elements. The skyline profile is determined by iterating over all the elements and determining the assembled locations of each element in the local matrices. The constructor prototype is:

FESolnMtx(const DataRep&, const Array<Element*>&);

```
i ← 0
Use an ElementDOFIter to set p to successive DOFs for e
    If p is free
        Use DataRep::free_DOF to get FreeDOF n corresponding to p
        j ← 0
        Use an ElementDOFIter to set q to successive DOFs for e
            If q is free
                Use DataRep::free_DOF to get FreeDOF m corresponding to q
                Add [K_e]_ij, to [K]_nm
            End if
            j ← j + 1
        End iteration over q
    End if
    i ← i + 1
End iteration over p
```

Figure 8: *Algorithm to assemble the local matrix of element e*

An FESolnVec is a specialization of a Vec: an array of doubles with member functions to implement arithmetic operators. An FESolnVec has two member functions for assembling the global right hand side vector, $\{f\}$. One assembles the fixed DOFs of the local element matrix into the vector. The other assembles the local right hand side vector into the global vector. The assembly algorithms are similar to that for an FESolnMtx. The prototypes for these functions are:

```
void assemble(const SqMtx&, const Element&);
void assemble(const Vec&, const Element&);
```

Like an FESolnMtx, an FESolnVec is constructed from the DataRep on which it depends. A copy constructor is also provided. The prototype for the constructors are:

```
FESolnVec(const DataRep&);
FESolnVec(const FESolnVec&);
```

5.4 The FESweep Class

The FESweep class is the basis of the sweeps for all the finite element solvers. It is quite simple, all the hard work being performed by the DataRep, FESolnMtx, and Element classes.

An FESweep stores an array of pointers to the Elements used by the solver, a DataRep, and an FESolnMtx and an FESolnVec to describe the linear system:

```
Array<Element*> elems;
DataRep *data;
FESolnMtx *mtx;
FESolnVec *rhs;
```

The sweep algorithm is:

Zero the linear system
For every element pointer e in elems
Get the local matrix using e->contribution_to_lhs()
Assemble the local matrix into the linear system
Get the local right hand side vector using e->contribution_to_rhs()
Assemble the local vector into the linear system
End iteration over elements
Using SkylineMtx::solve(), *solve the linear system return the solution in rhs*
Swap the solution from rhs to data

Notice that since the functions contribution_to_lhs() and contribution_to_rhs() are virtual, the FESweep need not contain any information about the actual problem being solved; that is localized within the elements themselves. However, this also implies that the FESweep class cannot define the pointers in elems; that must be done by specializations of the class which use specific element types to solve specific problems.

An FESweep is constructed from an FEBlock:

```
FESweep(FEBlock *b);
```

Its other member functions are redefinitions of the standard Sweep virtual functions (see Reference 13, Section 4.1).

void initialize();
: Creates an FESolnMtx and an FESolnVec in which to store the linear system. Also sets GeomElement::coords to point to the coordinates in the FEblock from which the sweep was constructed.

void sweep();
: Implements the sweep algorithm described above.

void finalize();
: Writes timing information to cout.

The sweep extractor is also overloaded. It is able to read the records FIXED BOUNDARY and PROPERTIES from an OFFSRF file. The FIXED BOUNDARY record allows degrees of freedom to be set to a fixed value at a collection of nodes. The format of the record is:

{FIXED BOUNDARY: *DOF value*
node-numbers
}FIXED BOUNDARY

where *DOF* is the number of the Node DOF to be set at each node, *value* is the value to give to the DOF, and *node-numbers* is a collection of node numbers whose DOFs are to be set. For example, NSElements associate two DOFs with each node; the two components of the velocity. The following FIXED BOUNDARY records will therefore set the velocity to zero at the nodes 0 to 10 (the FE2dNSSweep provides an easier way to do this: see Section 6.2).

```
{FIXED BOUNDARY: 0 0.0
  0 1 2 3 4 5 6 7 8 9 10
}FIXED BOUNDARY
{FIXED BOUNDARY: 1 0.0
  0 1 2 3 4 5 6 7 8 9 10
}FIXED BOUNDARY
```

The PROPERTIES record allows values of element properties to be defined. Its format is:

```
{PROPERTIES: number-of-properties  property-values
  element-numbers
}PROPERTIES
```

For example, the record

```
{PROPERTIES: 3 0.0 1.0 2.0
  11 12 13 14 15 16 17 18 19 20
}PROPERTIES
```

sets the three properties of elements 11 through 20 to the values 0.0, 1.0, and 2.0; the virtual function Element::set_properties is used. If one of these elements has more or fewer properties than three, a fatal error will be generated.

Because the FESweep contains only generic information about its elements, the extractor cannot perform rigorous checking of the input data, since it cannot tell whether the values given are appropriate. For that reason it is preferable that use of the FIXED BOUNDARY and PROPERTIES records be avoided. Instead, specializations of FESweep provide extractors which read and check input which is specific to the elements which they contain.

6 A Solver for the Navier Stokes Equations

The preceeding sections have described classes to implement a generic finite element solver. This section and the two following describe solvers for specific problems. In this section we describe a solver for the two-dimensional laminar Navier-Stokes equations.

6.1 Elements to Solve the Navier-Stokes Equations

We now consider elements which can be used to solve the steady two-dimensional laminar Navier-Stokes equations using a penalty function method to determine the pressure. Following Pineau[12] we write the variational form of the problem as follows.

$$\int_V \delta u \left\{ u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{\partial p}{\partial x} - \frac{1}{R_e}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \right\} dV = 0 \tag{6.1}$$

$$\int_V \delta v \left\{ u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{\partial p}{\partial y} - \frac{1}{R_e}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \right\} dV = 0 \tag{6.2}$$

$$\int_V \delta p \left\{ \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{p}{\lambda} \right\} dV = 0 \tag{6.3}$$

where λ is a very large number ($\lambda \approx 10^8$) called the penalty coefficient. Integrating equation (6.1) by parts yields

$$\int_V \left\{ \delta u \left(u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} \right) + p \frac{\partial \delta u}{\partial x} + \frac{1}{R_e} \left(\frac{\partial u}{\partial x} \frac{\partial \delta u}{\partial x} + \frac{\partial u}{\partial y} \frac{\partial \delta u}{\partial y} \right) \right\} dV - \int_S \left\{ p n_x \delta u + \frac{\delta u}{R_e} \left(n_x \frac{\partial u}{\partial x} + n_y \frac{\partial u}{\partial y} \right) \right\} dS = 0 \tag{6.4}$$

with a similar transformation of equation (6.2). The full integration is approximated by the sum of integrals over each of the elements. Two types of element are needed: a VolumeElement for the integral over V, and a BoundaryElement for the integral over S. The former is represented by the class NS2dVolElement, the latter by the class NS2dBndElement.

6.1.1 The NS2dVolElement Class

An NS2dVolElement has two degrees of freedom at each node: the two components of the velocity. Over the element the velocity is approximated according to equation (3.3).

$$u(\xi,\eta) = \sum_{n=1}^{N} u_n N_n(\xi,\eta); \qquad v(\xi,\eta) = \sum_{n=1}^{N} v_n N_n(\xi,\eta) \tag{6.5}$$

The velocity variations are defined by

$$\delta u(\xi,\eta) = \sum_{n-1}^{N} \delta u_n N_n(\xi,\eta); \qquad \delta v(\xi,\eta) = \sum_{n=1}^{N} \delta v_n N_n(\xi,\eta) \tag{6.6}$$

The pressure is not represented by the nodal functions. Instead we write

$$p(\xi,\eta) = \sum_{n=1}^{N_p} p_n P_n(\xi,\eta); \qquad \delta p(\xi,\eta) = \sum_{n=1}^{N_p} \delta p_n P_n(\xi,\eta). \tag{6.7}$$

For a consistent approximation of pressure and velocity, equation (6.3) requires that the polynomial order of the pressure be of order one less than the polynomial order of the velocity. Therefore, for linear reference elements, the pressure is constant over the element: $N_p = 1$ and $P_1(\xi,\eta) = 1$. For quadratic reference elements, the pressure is linear: $N_p = 3$, $P_1(\xi,\eta) = 1$, $P_2(\xi,\eta) = \xi$, and $P_3(\xi,\eta) = \eta$. The contribution to the integrals over V in equations (6.3) and (6.4) may then be written

$$\{\delta u, \delta v, \delta p\} \begin{bmatrix} [K_{uu}] & [K_{up}] \\ [K_{pu}] & \frac{1}{\lambda}[K_{pp}] \end{bmatrix} \begin{Bmatrix} u \\ v \\ p \end{Bmatrix} \tag{6.8}$$

where $[K_{uu}]$ is a $2N \times 2N$ matrix, $[K_{up}]$ is a $2N \times N_p$ matrix, $-[K_{pu}]$ is the transpose of $[K_{up}]$, and $[K_{pp}]$ is a $2N_p \times 2N_p$ matrix. In equation (6.8) the row vector $\{\delta u, \delta v, \delta p\}$ is short for $\{\delta u_1, \delta v_1, \cdots, \delta u_N, \delta v_N, \delta p_1, \cdots, \delta p_{N_p}\}$; similar notation is used for the column vector. The elements of the sub-matrices are as follows:

$$[K_{uu}]_{2n-1,2m-1} = [K_{uu}]_{2n,2m} = \int_V \left\{ N_n \left(u \frac{\partial N_m}{\partial x} + v \frac{\partial N_m}{\partial y} \right) + \frac{1}{R_e} \left(\frac{\partial N_n}{\partial x} \frac{\partial N_m}{\partial x} + \frac{\partial N_n}{\partial y} \frac{\partial N_m}{\partial y} \right) \right\} dV \quad (6.9)$$

$$[K_{uu}]_{2n-1,2m} = [K_{uu}]_{2n,2m-1} = 0 \quad (6.10)$$

$$[K_{up}]_{2n-1,m} = -[K_{pu}]_{m,2n-1} = \int_V P_m \frac{\partial N_n}{\partial x} dV \quad (6.11)$$

$$[K_{up}]_{2n,m} = -[K_{pu}]_{m,2n} = \int_V P_m \frac{\partial N_n}{\partial y} dV \quad (6.12)$$

$$[K_{pp}]_{n,m} = \int_V P_n P_m dV \quad (6.13)$$

The elements of the sub-matrices $[K_{uu}]$, $[K_{up}]$, and $[K_{pp}]$ are evaluated simultaneously using the function GeomElement::integrate_fnc(). The integrands at each Gaussian point are evaluated by the function NS_lhs_at_gp(). The matrix elements are passed to NS_lhs_at_gp() as a single array of length $4N^2 + 2NN_p + N_p^2$. The values of the velocity at each Gaussian point (needed to evaluate $[K_{uu}]$) are determined using GeomElement::values_at_gp().

Equation (6.3) must hold for the integral over each element, so that

$$[K_{pu}] \begin{Bmatrix} u_n \\ v_n \end{Bmatrix} + \frac{1}{\lambda} [K_{pp}] \{p_n\} = 0 \quad (6.14)$$

from which one obtains

$$\{p_n\} = -\lambda [K_{pp}]^{-1} [K_{pu}] \begin{Bmatrix} u_n \\ v_n \end{Bmatrix} \quad (6.15)$$

The contribution to the left hand side of equation (6.4) can therefore be rewritten as

$$\{\delta u_n, \delta v_n\} [K_E] \begin{Bmatrix} u_n \\ v_n \end{Bmatrix} \quad (6.16)$$

with

$$[K_E] = [K_{uu}] - \lambda [K_{up}] [K_{pp}]^{-1} [K_{pu}] \quad (6.17)$$

The $2N \times 2N$ matrix, $[K_E]$ is returned by the NS2dVolElement member function

contribution_to_lhs().

Since $[K_E]$ depends on the values of the velocity at the nodes of the element, the problem is non-linear. The NS2dVolElement uses the currently defined values of velocity when computing $[K_E]$; hence, it implements a simple substitution method. Faster convergence would likely be obtained, for laminar problems at least, if a Newton-Raphson iteration scheme were used instead. This has not yet been implemented in TRANSOM.

The NS2dVolElement class has a public static member pv which is a pointer to a PVIOVar (see Reference 13, Section 3.5.2) which stores values for the pressure and velocity at all nodes of the grid. During the calculation of $[K_E]$ the velocity values are obtained from pv. The value of pv must be set before assembling the linear system.

The value of the pressure is not continuous at the element boundaries. Hence, at every node there are multiple values of the pressure: one associated with every NS2dVolElement which contains the node. Therefore, it is not possible simply to store the pressure in pv. Instead, every NS2dVolElement contains an array which stores the values of the pressure at its nodes. The pressure values are calculated by NS2dVolElement::contribution_to_lhs using equation (6.15). The solver which uses the NS2dVolElements must provide some averaging procedure to reduce the multiple pressure values at the nodes to a single value: see Section 6.2. The vector of pressure values may be obtained using the following member function.

double get_pressure(unsigned n) const;
Returns the pressure at the n^{th} node of the element.

The NS2dVolElement class also has two public static members of type double, reyno and lambda. They store the values of the Reynolds number and the penalty parameter, respectively.

6.1.2 The NS2dBndElement Class

An NS2dBndElement contributes to the right hand side of the linear system through the surface integral of equation (6.4). The local right hand side vector is

$$\{f_E\} = \int_S \begin{Bmatrix} N_n f_x \\ N_n f_y \end{Bmatrix} dS \tag{6.18}$$

with

$$f_x = pn_x + \tau_x; \quad f_y = pn_y + \tau_y \tag{6.19}$$

$$\tau_x = \frac{1}{R_e}\left(n_x \frac{\partial u}{\partial x} + n_y \frac{\partial u}{\partial y}\right); \quad \tau_y = \frac{1}{R_e}\left(n_x \frac{\partial v}{\partial x} + n_y \frac{\partial v}{\partial y}\right) \tag{6.20}$$

and where n_x and n_y are the components of the outward pointing normal to

the element.

Like the NS2dVolElement class, NS2dBndElement contains a public static pointer to a PVIOVar which is used to obtain the values of the pressure at each of the nodes of the element. The pressure over the element is then represented using equation (3.2).

The values of the shear stress (τ_x, τ_y) at each node are treated as properties of the element. They are stored in an array of two-vectors whose values can be set using the following member functions.

virtual void set_properties(const Vec &v);

Uses the values in v to set the values of pressure and shear stress. The vector v must be of length $3N$ and is arranged as $(p_1, \tau_{x1}, \tau_{y1}, \cdots, p_N, \tau_{xN}, \tau_{yN})$.

void set_shear_stress(const Array<Mtx*> &gradv);

Uses the values in gradv to set the values of the shear stress at each node. The array gradv is of length N. gradv[n] is a pointer to a 2×2 matrix containing the gradient of the velocity at node n; thus for example, (*gradv[n])[0][1] is the value of $\partial u / \partial y$ at node n.

6.2 The FE2dNSSweep Class

We now look at the classes FE2dNSSolver and FE2dNSSweep used to implement a finite element solver for the steady laminar Navier-Stokes equations in two dimensions. The class hierarchy for both classes is shown in Figure 7.

An FE2dNSSweep is a specialization of an FESweep which uses NS2dVolElements and NS2dBndElements. Its member reyno is used to store the Reynolds number. The pressure and velocity values are stored in a PVIOVar to which pv is a pointer.

FE2dNSSweep(FEBlock *b, PVIOVar *p);

Creates an FE2dNSSweep on the block b which uses the pressure and velocity values in p: i.e. pv is set to p. The Element pointers in the array FESweep::elems are assigned by using each of the RefElements in b to create either a new NS2dBndElement or a new NS2dVolElement, the choice being made according to whether the dimension of the RefElement is one or two.

virtual void initialize();

Initializes the sweep by calling FESweep::initialize(). It then initializes all free velocity components to zero, and, finally, copies the velocity values from PVIOVar pv into the DataRep FESweep::data.

virtual void sweep();

Sets the PVIOVars NS2dVolElement::pv and NS2dBndElement::pv to pv and NS2dVolElement::reyno to reyno. FESweep::sweep is called to perform the sweep. The pressure values are then calculated using the

algorithm described below, and the new pressure and velocity values are copied to pv.

During each call to NS2dVolElement::contribution_to_lhs(), the pressure at each node is updated using the current values for the velocity. On the first call, since the initial velocity field may not be incompressible, the pressure values are usually of order λ: i.e. very large. If the pressure values are shared with another solver in a multi-block solution, such large values will often hinder convergence. To avoid this problem, the first time the sweep is performed after initialization, FESweep::sweep() is called twice. After the second call the pressure will have more physically reasonable values.

FEBlock* get_block() const;
Returns the FEblock used by the sweep.

void set_reynolds_number(double r);
Sets the Reynolds number to r.

The FE2dNSSweep extractor is overloaded to read the following records from an OFFSRF file.

1. REYNOLDS NUMBER: sets the value of the Reynolds number. The format for this record is:

   {REYNOLDS NUMBER: *Reynolds-number* }

2. PENALTY PARAMETER: sets value of NS2dVolElement::lambda. The format for this record is:

   {PENALTY PARAMETER: *penalty-parameter* }

3. SOLID BOUNDARY: sets the velocity to zero at specified nodes. The format for this record is:

   {SOLID BOUNDARY
   node-numbers
   }SOLID BOUNDARY

   where *node-numbers* is a list of the nodes on the solid boundary.

4. FIXED VELOCITY: sets the velocity to a fixed value at specified nodes. The format for this record is:

   {FIXED VELOCITY: v_x v_y
   node-numbers
   }FIXED VELOCITY

   where v_x and v_y are the components of the velocity (if the dimension of the coordinates is three, there will also be a v_z value) and *node-numbers* is a list of the nodes at which the velocity is to be set.

5. FIXED BOUNDARY: this record is described in Section 5.4.

6. PROPERTIES: this record is described in Section 5.4.

The FE2dNSSweep inserter is overloaded so that it calls the inserter for the PVIOVar pv. This causes the pressure and velocity values to be written into an OFFSRF file. The pressure values are written into a Pressure record and the velocity values into a Velocity record. See Reference 13, Section 3.5.2 for more details.

After each sweep, the velocity values are copied from FESweep::data to pv. The pressure in pv must also be updated, but this is more complicated. Every NS2dVolElement which contains node n stores a value for the pressure there; however, this value will be different for every element. Moreover, if the node also belongs to an NS2dBndElement, there is a fixed value for the pressure. The pressure at each node is calculated as follows. If the node belongs to an NS2dBndElement, use the fixed value associated with that element. Otherwise use the average of the values in all NS2dVolElements which contain the node.

6.3 The FE2dNSSolver Class

The FE2dNSSolver class is a specialization of the Solver class which has an FE2dNSSweep for its sweep and a combination of an NTimesStopTest and a CompDataStopTest for its stop test (these classes are described in Reference 13, Sections 4.2.2 and 4.2.3). Thus, it solves the two-dimensional steady laminar Navier-Stokes equations by solving equation (2.5) repeatedly until successive values of the pressure and velocity have changed by no more than small pre-defined amount, ε. It has the following member functions and constructors.

FE2dNSSolver(FEBlock *b);
: Makes an FE2dNSSolver which uses the reference elements and coordinates in the FEBlock b. A new PVIOVar is allocated to store the pressure and velocity values.

FE2dNSSolver(FEBlock *b, const IOVar &vars, const Node &n);
: Makes an FE2dNSSolver which uses the reference elements and coordinates in the FEBlock b. If vars contains a Pressure and a Velocity field, then the pressure and velocity values will be shared with those in vars starting at node n. For example, suppose that vars contains pressure and velocity values on a block with 1000 nodes numbered from 0 to 999. If n is 500, and the block b contains 500 nodes, then pv shares the last 500 pressure and velocity values in vars; changing the pressure value at node zero in pv causes the pressure value at node 500 in vars to be changed as well. Similarly changes to vars will cause corresponding changes to pv.

FEBlock* get_block() const;
: Returns the block used by the solver.

Element* get_element(unsigned n) const;
: Returns the n^{th} element used by the solver.

void set_reynolds_number(double r);
: Sets the Reynolds number to r.

The FE2dNSSolver extractor reads the following records from an OFFSRF file; they allow control over the stop test.

1. MAXIMUM NUMBER ITERATIONS: sets the maximum number of iterations for the sweep. The format of this record is:

   {MAXIMUM NUMBER OF ITERATIONS: n }

   where n is the maximum number of iterations.

2. ACCURACY: sets the value of ε_{acc} for the CompDataStopTest. The iteration will end if successive values of pressure and velocity all differ by less than ε_{acc}. The format for this record is:

   {ACCURACY: ε_{acc} }

3. MAXIMUM ACCURACY: sets the value of ε_{div} for the CompDataStopTest. The iteration will end if successive values of pressure and velocity all differ by more than than ε_{div}. The format for this record is:

   {MAXIMUM ACCURACY: ε_{div} }

7 A Solver for Laplace's Equation

In this section we describe a solver for Laplace's equation.

$$\nabla^2 \varphi = 0 \tag{7.1}$$

This equation is important in many branches of physics, but in fluid mechanics is used chiefly in the calculation of potential flow.

7.1 The LaplaceElement Class

A LaplaceElement has a single degree of freedom at each node; the value of φ. A finite element approximation to equation (7.1) is obtained by rewriting it as

$$\int_V \delta\varphi \, \nabla^2 \varphi \, dV = 0 \tag{7.2}$$

Integrating by parts one obtains

$$\int_S \delta\varphi \, \frac{\partial \varphi}{\partial n} \, dS - \int_V \nabla \delta\varphi \, \nabla \varphi \, dV = 0 \tag{7.3}$$

where $\partial\varphi / \partial n$ denotes the derivative normal to the surface S. If we restrict the problem to either Neumann or Dirichlet boundary conditions, then the surface integral vanishes. Using equations (2.2) and (2.3) to approximate φ and $\delta\varphi$, one gets the following expression for the local matrix:

$$[K_E]_{nm} = \int_E \nabla N_n(x,y) \cdot \nabla N_m(x,y) \, dV = 0 \tag{7.4}$$

The LaplaceElement class is a specialization of the Element class. Its contribution_to_lhs member function calculates the local matrix according to equation (7.4); the contribution_to_rhs function returns a null pointer, indicating that the local vector is zero. A LaplaceElement can be constructed from any VolumeElement.

7.2 The FELaplaceSweep Class

An FELaplaceSweep is a sweep for solving Laplace's equation using the finite element method. It inherits all its member functions from FESweep. An FELaplaceSweep is constructed from an FEBlock and an IOVar (see Reference 13, Section 3.4):

```
FELaplaceSweep(FEBlock *b, IOVar *v);
```

The first active field in the IOVar is used to store the values of φ. The constructor creates LaplaceElements from the RefElements in b.

An FELaplaceSolver is a Solver which has an FELaplaceSweep for its sweep. Since Laplace's equation is linear, the sweep need only be executed once. Therefore an NTimesStopTest is used for the stop test. It is set to execute only once. The following constructors are available for the FELaplaceSolver class:

```
FELaplaceSolver(FEBlock *b);
```
Creates an FELaplaceSolver using the block b. A new IOVar is created to store the values of φ. It has a single active field of length one which is named "Potential".

```
FELaplaceSolver(FEBlock *b, const IOVar &v, const Node &n);
```
Creates an FELaplaceSolver using the block b. The values of φ will be stored in the first active field of IOVar v starting at node n.

8 A Solver to Calculate Streamfunctions

In this section we describe a solver which calculates a streamfunction for an incompressible two-dimensional velocity field.

8.1 The SfncElement Class

For a given two-dimensional flow field (v_x, v_y), the streamfunction ψ satisfies

$$\frac{\partial \psi}{\partial x} = v_y; \qquad \frac{\partial \psi}{\partial y} = -v_x \tag{8.1}$$

An approximation to ψ is found by minimizing the integral

$$\int_V \left[\left(\frac{\partial \psi}{\partial x} - v_y \right)^2 + \left(\frac{\partial \psi}{\partial y} + v_x \right)^2 \right] dV \tag{8.2}$$

Using equation (2.2) to approximate ψ one gets

$$\sum_{k=1}^{N_e}\int_{E_k}\left[\left(\sum_n \psi_n \frac{\partial N_n}{\partial x} - v_y\right)^2 + \left(\sum_n \psi_n \frac{\partial N_n}{\partial y} + v_x\right)^2\right] dV = \text{minimum} \qquad (8.3)$$

A minimum is obtained when

$$\sum_{k=1}^{N_e}\int_{E_k}\left[\sum_{m,n}\left(\frac{\partial N_m}{\partial x}\frac{\partial N_n}{\partial x} + \frac{\partial N_m}{\partial y}\frac{\partial N_n}{\partial y}\right)\psi_n - \sum_m\left(\frac{\partial N_m}{\partial x}v_y - \frac{\partial N_m}{\partial y}v_x\right)\right] dV = 0 \qquad (8.4)$$

so that the local matrix and local vector are

$$[K_E]_{mn} = \int_{E_k}\left(\frac{\partial N_m}{\partial x}\frac{\partial N_n}{\partial x} + \frac{\partial N_m}{\partial y}\frac{\partial N_n}{\partial y}\right) dV; \qquad \{f_E\} = \int_E\left(\frac{\partial N_m}{\partial x}v_y - \frac{\partial N_m}{\partial y}v_x\right) dV \qquad (8.5)$$

Notice that the local matrix is the same as the local matrix for the two-dimensional LaplaceElement. Hence, the SfncElement is defined as a specialization of the LaplaceElement, which inherits the contribution_to_lhs member function. The contribution_to_rhs member function is overloaded to calculate $\{f_E\}$ according to equation (8.5). An SfncElement can be constructed from any two-dimensional VolumeElement.

8.2 The FESfncSweep and FESfncSolver Classes

An FESfncSweep is a sweep for calculating the streamfunction which corresponds to a given incompressible velocity field. It inherits all its member functions from FESweep. An FESfncSweep is constructed from an FEBlock which contains the reference elements from which the SfncElements are constructed, a VelIOVar which contains the velocity field, and an SfncIOVar which is used to contain the values of the streamfunction:

```
FESfncSweep(FEBlock *b, VelIOVar *v, SfncIOVar *s);
```

An FESfncSolver is a Solver which has an FESfncSweep for its sweep. Since the calculation of the streamfunction is linear, the sweep need only be executed once. Therefore an NTimesStopTest is used for the stop test. It is set to execute only once. The following constructors are available for the FESfncSolver class:

```
FESfncSolver(FEBlock *b);
```

Makes an FESfncSolver using the block b to construct the sweep. A new VelIOVar is created to store the velocity field. The velocity values must be defined by reading them from an input file. A new SfncIOVar is created to store the values of the streamfunction.

```
FESfncSolver(FEBlock *b, const IOVar &v, const Node &n);
```

Makes an FESfncSolver using the block b to construct the sweep. If v contains a field called "Velocity", then this field, starting at node n, is used to obtain the values of the velocity; otherwise a new VelIOVar is created to store the velocity field. If v contains a field called "Streamfunction", then this field, starting at node n, is used to store the values of the streamfunction; otherwise a new SfncIOVar is

created to store the streamfunction.

9 Recommendations for Improvements

The finite element solvers in TRANSOM are newly written code. They can be improved in several areas, mostly to improve the efficiency.

To ease the programming burden, the element classes NS2dVolElement and NS2dBndElement have been written so that they can be constructed from arbitrary reference elements. However, in some cases the use of specialized elements would decrease the computation of the local matrix thus reducing the time taken to assemble the global matrix.

No attempt has been made to reduce the matrix bandwidth by implementing a node re-ordering scheme such as the reverse Cuthill-McKee algorithm[18]. Reduction of solution time would also be gained by using a more efficient matrix solver. The current skyline algorithm was used for compatibility with MEF. For the initial code development it was thought that this would reduce the likelihood of difficult-to-find bugs. To increase the efficiency of the solution of the global linear system, the use of a subroutine library such as FMSLIB should be investigated as should the use of parallel processors.

A reduction in the number of iterations to convergence may be obtained by using an improvement on the substitution method used to update the velocity field. The Newton-Raphson method and the semi-implicit over-relaxed time marching methods implemented in MEF should be included as options in TRANSOM.

The error reporting in TRANSOM should also be improved. TRANSOM was begun before a C++ compiler implementing exception handling was available. Consequently almost all errors are treated as fatal and only terse error messages are given. Use of the C++ exception handling should provide much better identification of errors and, in some cases, allow recovery from errors.

The k-ε turbulence model already implemented in the DREA version of MEF should be incorporated into TRANSOM. For compatibility with the next version of the TRANSOM pseudo-compressibility solver, the Baldwin-Barth turbulence model[9] should also be implemented. The solver should also be extended to three-dimensional flow.

10 Concluding Remarks

TRANSOM is a complex program which is still in the early stage of development. It is likely that some of the classes defined in this memorandum will be changed as that development continues.

The two-dimensional laminar finite element flow solver works well for Reynolds numbers up to about 1000. At higher Reynolds numbers convergence is sometimes not attained. Implementation of the improved iteration schemes described in the previous section may alleviate this problem.

For most two-dimensional problems the TRANSOM finite element solver is acceptably fast; however, before realistic three-dimensional flows are attempted, it will be necessary to make it faster.

The other major shortcoming of this solver is that there is no grid generator to ease the generation of input files. There are many commercial grid generators available, including some used by the Structural Mechanics Group at DREA. An interface between one of these codes and TRANSOM should be written.

As described in the introduction, it is intended that TRANSOM should be able to apply its pseudo-compressibility solver and its finite element solver concurrently on different flow blocks. Some early experimentation with this has been done, although convergence is at best slow and at worst not attained. Work in this area will continue over the coming year.

Appendices

A Vectors and Matrices

Many of the C++ prototypes given in the body of this memorandum make use of the classes Vec and Mtx. A brief description of these classes is given here.

A.1 The Vec Class

The Vec class represents numerical vectors: i.e. an array of floating point values. Vectors represented by the Vec class have fixed length which is specified when the Vec is constructed. The constructors and member functions of the Vec class are as follows.

Vec(unsigned n, double *d = 0);
: Constructs a Vec with length n with elements stored at d. If d is null, new memory will be allocated.

Vec(const Vec&);
: Copy constructor.

unsigned len() const;
: Returns the length of the vector.

void zero();
: Sets every element of the vector to zero.

void unit();
: Converts the vector to a unit vector

void operator=(const Vec &v);
: Copies the elements of the vector v. If the lengths of the vectors are not the same a fatal error will be generated.

Vec& operator-();
: Negates every element of the vector and returns it.

void operator+=(const Vec &v);
: Adds v to the vector.

void operator-=(const Vec &v);
: Subtracts v from the vector.

void operator*=(double d);
: Multiplies the vector by d;

void operator/=(double d);
: Divides the vector by d;

Vec operator*(const Mtx &m) const;
: Interprets the vector as a row vector and multiplies it on the right by m. The Vec returned is interpreted as a column vector. If the

number of columns of m is not the same as the length of the vector then a fatal error is generated.

```
double& operator[](unsigned n);
const double& operator[](unsigned n) const;
```
Return the n^{th} element of the vector. No subscript checking is done.

```
operator double *();
operator const double *() const;
```
Convert the vector to a pointer to a sequence of doubles. This can be used to get gains in efficiency in computationally intensive inner loops.

```
ostream& operator<<(ostream &out, const Vec &vc);
```
Writes all the vector elements to the output stream.

```
istream& operator>>(istream &in, Vec &vc);
```
Reads all the vector elements from the input stream.

```
int operator==(const Vec &vc1, const Vec &vc2);
```
Returns true if the lengths of vc1 and vc2 are the same and their elements have the same values.

```
int operator!=(const Vec &vc1, const Vec &vc2);
```
Returns !(vc1 == vc2).

```
double abs(const Vec &vc);
```
Returns the magnitude of the vector.

```
double min(const Vec &vc);
```
Returns the value of the element with smallest value.

```
double max(const Vec &vc);
```
Returns the value of the element with largest value.

```
double min_abs(const Vec &vc);
```
Returns the value of the element whose absolute value is smallest.

```
double max_abs(const Vec &vc);
```
Returns the value of the element whose absolute value is largest.

```
double dot(const Vec &vc1, const Vec &vc2);
```
Returns the dot product of vc1 and vc2.

A.2 The Mtx Class

The Mtx class represents a numerical matrix. It is implemented as a specialization of the Vec class so that many of the Vec member functions can be inherited. Matrices represented by the Mtx class have fixed dimensions which are specified when the Mtx is constructed. The constructors and member functions of the Mtx class, not including those inherited from the Vec class, are as follows.

```
Mtx(unsigned n, unsigned m, double *d = 0);
```
Constructs an n x m matrix with elements stored at d. If d is null, new memory is allocated.

`Mtx(const Mtx&);`
Copy constructor

`void operator=(const Mtx&);`
Assignment operator

`Vec row(unsigned n) const`
Returns a `Vec` containing the n^{th} row of the matrix.

`unsigned num_rows() const;`
Returns the number of rows in the matrix.

`unsigned num_cols() const;`
Returns the number of columns in the matrix.

`Mtx& operator-();`
Negates every element in the matrix and returns it.

`Vec operator*(const Vec &v) const;`
Returns the row vector result of multiplying the column vector `v` by the matrix.

`Mtx operator*(const Mtx&m) const;`
Returns the result of multiplying `m` by the matrix.

`double* operator[](unsigned n);`
`const double* operator[](unsigned n) const;`
Returns a pointer to the values in the n^{th} row of the matrix. Thus, if `m` is a matrix, its $(n,m)^{th}$ element is `m[n][m]`.

```
typedef double *dptr;
operator dptr *();
operator const dptr *() const;
```
Convert the matrix to a sequence of pointers to its rows.

`ostream& operator<<(ostream &out, const Mtx &m);`
Writes the elements of `m` to the output stream.

`int operator==(const Mtx &m1, const Mtx &m2);`
Returns true if `m1` and `m2` have the same dimensions and if their elements are the same.

`int operator!=(const Mtx &vc1, const Mtx &vc2);`
Returns `!(m1 == m2)`.

A.3 The SqMtx Class

The `SqMtx` class is a specialization of the `Mtx` class; it is used to represent square matrices. Member functions are provided to invert the matrix and to calculate its determinant. The constructors and member functions of the `SqMtx` class, not including those inherited from the `Mtx` class, are as follows:

`SqMtx(unsigned n, double *d = 0);`
Constructs an `n x n` matrix with elements stored at `d`. If `d` is null, new memory is allocated.

`void identity();`
Initialize matrix to the identity.

void lu_decomp();
Calculate the LU decomposition of the matrix. It is an error to LU decompose a matrix twice. After LU-decomposition the original matrix is destroyed.

void invert();
Invert the matrix using Gaussian elimination with partial pivoting.

double determinant();
Calculate the determinant. The matrix will be LU-decomposed if it has not already been; thus, calculating the determinant will destroy the original matrix.

void solve(const Vec &b, Vec &x);
Solve A*x = b, where A = *this. The matrix will be LU-decomposed if it has not already been; thus, using solve() will destroy the original matrix.

B Arrays

The Array class is a C++ template used to implement dynamic arrays of arbitrary objects. Arrays can have length zero, allowing arrays of arrays to be defined easily. To allow efficient copying of arrays and to reduce overhead in allocating memory, the length of allocated memory is stored separately from the length of the array. The constructors and member functions of the Array class are as follows (in the descriptions it is assumed that the array stores objects of type T):

Array();
Constructs an array of length zero. No memory is allocated.

Array(int s);
Constructs an array of length s. Enough memory is allocated to store s objects of type T.

Array(int s, int m);
Constructs an array of length s. Enough memory is allocated to store max(m,s) objects of type T.

Array(const Array&);
Copy constructor.

Array& operator=(const Array &v);
Makes the length of the array the same as the length of v, then copies the elements of v.

int len() const;
Returns the length of the array.

int max_len() const;
Returns the number of objects of type T which will fit in the memory currently allocated for the array. The array can be resized to length max_len() without reallocation of memory.

void set_len(int n);
Changes the length of the array to n. If there is insufficient memory available, new memory will be allocated and the existing elements of the array will be copied to the new memory.

void set_mem(int n);
Increases the amount of allocated memory so that n objects of type T will fit. If the current memory allocation exceeds n, nothing is done. If new memory is allocated, the elements of the array are copied to the new memory.

void shiftl(int shift, int lo, int hi);
Shifts elements lo to hi to the left by shift places.

void shiftr(int shift, int lo, int hi);
Shifts elements lo to hi to the right by shift places.

void shiftc(int shift);
Shifts all the elements of the array cyclically by shift places. Elements which are shifted off the end of the array reappear at the beginning, and vice versa.

void insert(const T& t, int n);
Inserts t into the array at element n. The current elements from n to the end of the array are shifted to the right by one place and the array length is increased by one.

void insert(const Array&v, int v);
Inserts v into the array at element n. The current elements from n to the end of the array are shifted to the right and the array length is increased by the length of v.

void remove(int n);
Removes element n reducing the size of the array by one.

void remove(int lo, int hi);
Removes all elements from lo to hi.

Array& append(const Array &v);
Appends v to the array.

T& operator[](int n);
const T& operator[](int n) const;
Return element n.

Array operator()(int lo, int hi) const;
Returns a new array equivalent to the sub-array from elements lo to hi.

operator T *();
operator const T *() const;
Convert the vector to a pointer to a sequence of objects of type T. This can be used to get gains in efficiency in computationally intensive inner loops or for automatic type conversions in function arguments.

References

1. D. Hally, "User's Guide for HLLFLO Version 2.0", DREA Technical Memorandum 93/309, 1993.
2. D. Hally, "Implementation of a Free Surface in Calculations of the Flow into the Propeller Plane of a Ship", in *Proceedings of CADMO92: Third International Conference on Computer Aided Design Manufacture and Operation in the Marine and Offshore Industries*, Madrid, Spain, October 1992.
3. D. J. Hall and D. W. Zingg and C. R. Ethier, "A Laminar Incompressible Navier-Stokes Flow Solver", DREA Contractor Report CR/93/462, 1993.
4. D. J. Hall and D. W. Zingg and C. R. Ethier, "A Two-Dimensional Incompressible Navier-Stokes Turbulent Flow Solver", DREA Contractor Report, CR/94/435, 1994.
5. D. Hally, "Revisions to NSI2D", DREA Technical Memorandum 94/303, 1994.
6. S. E. Rogers and D. Kwak, "An Upwind Differencing Scheme for the Incompressible Navier-Stokes Equations," NASA TN 101051, Nov. 1988.
7. B. S. Baldwin and H. Lomax, "Thin Layer Approximation and Algebraic Model for Separated Turbulent Flows," AIAA Paper 78-257, 1978.
8. H. C. Chen and V. C. Patel, "Near-Wall Turbulence Models for Complex Flows Including Separation," AIAA Journal Vol. 26, No. 6, pp. 641-648, June 1988.
9. B. S. Baldwin and T. J. Barth, "A One-Equation Turbulence Transport Model for High Reynolds Number Near Wall-Bounded Flows," NASA TM 102847, August 1990.
10. D. Hally, "Implementation of a Pseudo-Compressibility Navier-Stokes Solver in TRANSOM", DREA Technical communication, in preparation.
11. G. Dhatt and G. Touzot, *Une Présentation de la Méthode des Eléments Finis*, S. A. Maloine, ed., Paris, 1981.
12. F. Pineau, "A Finite Element Based Reynolds Averaged Navier-Stokes Solver for Modelling Three-Dimensional Turbulent Flows," DREA Report 93/106, 1993.
13. D. Hally, "TRANSOM: A Multi-Method Reynolds-Averaged Navier-Stokes Solver: Overall Design," DREA Technical Memorandum, in review.
14. D. Hally, "User's Guide for TRANSOM Version 1.0: A Two-Dimensional Multi-Method Reynolds-Averaged Navier-Stokes Solver," DREA Technical Memorandum, in preparation.

15. D. Hally, "OFFSRF: A System for Representing Ship Offset Data", DREA Technical Memorandum 89/305, 1989.
16. D. Hally, "C++ Classes for Reading and Writing files in OFFSRF Format", DREA Technical Memorandum 94/302, 1994.
17. M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions,* Applied Mathematics Series 55, National Bureau of Standards, 1965.
18. E. H. Cuthill, "Several Strategies for Reducing the Bandwidth of Matrices," in *Sparse Matrices and their Applications*, D. J. Rose and R. A Willoughby, eds., Plenum Press, New York, 1972, pp. 157-166.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence Research Establishment Atlantic
P.O. Box 1012, Dartmouth, N.S. B2Y 3Z7

2. SECURITY CLASSIFICATION (Overall security of the document including special warning terms if applicable.)

UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title.)

Implementation of a Finite Element Navier-Stokes Solver in TRANSOM

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

HALLY, David

5. DATE OF PUBLICATION (Month and year of publication of document.)

April 1997

6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)

57

6b. NO. OF REFS. (Total cited in document.)

18

6. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the reseach and development. include the address.)

Defence Research Establishment Atlantic
P.O. Box 1012, Dartmouth, N.S. B2Y 3Z7

9a. PROJECT OR GRANT NUMBER (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)

1.g.a

9b. CONTRACT NUMBER (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number mrst be unique to this document.)

DREA Technical Memorandum 97/225

10b. OTHER DOCUMENT NUMBERS (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; futher distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic annoucement of this document. This will normally correspond to the Document Availability (11). However, where futher distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

Unlimited

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

TRANSOM is a multi-block, multi-method Reynolds-Averaged Navier Stokes solver being developed at DREA to address problems associated with the flow around ships and submarines. It is multi-block because the flow is divided into several distinct regions. It is multi-method because a different solution method may be used on each of the flow regions. At present two different methods of solution can be chosen; a finite-volume solver based on the pseudo-compressibility method; and a finite element solver which uses the penalty function method to determine the pressure.

TRANSOM is written in C++ following principles of Object Oriented Programming. This document describes the design of the finite element flow solver in TRANSOM with emphasis on the class hierarchies used to represent elements finite element grids, degrees of freedom, and the solver itself. Two companion reports describe the overall design of TRANSOM and the design of the pseudo-compressibility solver.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

Fluid Flow
Turbulence
Reynolds-Averaged Navier-Stokes Equations
Hydrodynamics
Computer Programs
Object Oriented Design
TRANSOM

DREA

CRDA