NOTE TO USERS

Three Dimensional Symmetric Pyramidal Finite Elements

A thesis submitted to
Lakehead University
in partial fulfillment of the requirements
for the degree of
Master of Science

by
Kevin B. Davies
2004

Your file Votre référence
ISBN: 0-494-10653-0
Our file Notre référence
ISBN: 0-494-10653-0

Abstract

In this thesis we conduct an examination of pyramidal elements and their use as interface, or mortar, elements in the joining of tetrahedral and hexahedral elements in three dimensional finite element meshes. Several new sets of basis functions are developed and analyzed for pyramidal elements having five, thirteen and fourteen nodal points. These basis functions take advantage of the symmetries of the element in order to give improved accuracy over previously studied basis functions. Evidence of this improvement in accuracy is presented in the form of numerical experiments.

Acknowledgements

I would like to express my most sincere gratitude to my supervisor, Professor Liping Liu, for her advice, encouragement and generous support during the preparation of this thesis. I would also like to thank Professor Michal Křížek (Academy of Sciences, Czech Republic), Kewei Yuan and Professor Liu for their collaboration on the material in sections 3.1 and 3.2. I also wish to take this opportunity to express my gratitude to Mrs. Uta Hickins for her kindness and help during my years at Lakehead University.

My thanks go to Lynn Gollat and the rest of the staff at the office of graduate studies at Lakehead University for their assistance on numerous occasions. I would also like to extend my gratitude to the staff at the interlibrary loans department of the Chancellor Paterson Library for their resourcefulness and efficient delivery of research documents. I am also extremely grateful to the Natural Science and Engineering Research Council of Canada (NSERC) for providing me with financial support.

With my deepest appreciation I would like to acknowledge all the nurture, love and support my mother Freda Davies and grandmother Freda Kamstra have given me over all the years. Without their continued assistance this work would have never been completed. Finally, I would like to express my most heartfelt appreciation to my fiancée Li Fong Yong, to whom this thesis is dedicated, for all her patience, love and encouragement during my work on this project.

Kevin Davies
December 2004

Dedication:

For Li Fong

Table of Contents

1. Introduction

It is a common situation in three dimensional mesh discretizations that interior regions of the domain are better approximated by hexahedral elements, but tetrahedral elements are more suitable for geometrically complex areas, such as those near the domain boundary. Thus, it is desirable to construct a mesh that combines these element types so that we can achieve the best approximation possible. Unfortunately, hexahedral and tetrahedral elements cannot be joined together properly without the use of special interface elements. The Pyramidal finite element is one such interface, or mortar, element, and it is an extremely useful and flexible tool for joining tetrahedral and hexahedral portions of a three dimensional mesh.

This work is organized as follows: First we present a history of the finite element method, as well as a brief overview of how the method can be applied to problems in science and engineering. In section two we conduct a detailed examination of the finite element method and its applications, as well as a theoretical justification for its validity. We also present some mathematical concepts and discuss some related topics in numerical analysis, such as methods for solving large linear systems, that will be helpful to the understanding of material in later sections. In section three we proceed with our analysis of pyramidal mortar elements. Specifically, we examine three new symmetric elements, one five node element which has bilinear functions on its base and linear functions on its triangular faces, as well as thirteen and fourteen node pyramidal elements with biquadratic basis functions on their bases and quadratic functions on their triangular faces. The pyramidal element with five nodal points is most suitable for making connections between four node linear tetrahedral elements and eight node hexahedral elements, whereas the thirteen and fourteen node elements can be used for face-to-face connections between ten node quadratic tetrahedra and twenty node or twenty-seven node biquadratic hexahedral elements respectively. In section three, we also conduct a detailed examination of the basis functions developed for these new pyramidal elements, and prove the correctness of their construction. We also make the claim that these new basis functions, which take advantage of the symmetry of the element, reduce the discretization error of the mesh. In section four we outline the development of software tools to be used to conduct numerical experiments measuring the accuracy of the new elements. We follow this with a presentation of the test results from the numerical experiments showing that our claim is justified, and that the new symmetric pyramidal elements do in fact yield a better discretization error than other elements. Finally, we give some concluding remarks in section five, as well as considerations for possible future research in this area.

1.1 Preliminaries

History

Mathematical models for highly complex systems are regularly needed in science and engineering. Often these models take the form of differential or integral equations which are difficult, if not impossible to solve analytically. In the study of such highly complicated systems it has become increasingly important to employ computer technology to aid us in

developing and solving mathematical models. The rapid development of computer technology over the past few decades has provided us with increasingly powerful computers while the costs involved in computation have been reduced significantly. This environment of high performance but relatively cheap computational resources has further facilitated the use of computer-implemented mathematical models to simulate and analyze complicated processes in science and engineering.

Unfortunately, even with the availability of such computational power, it is still difficult to determine the exact analytical solutions for all but the simplest of model cases. In addition, numerical methods are required in order to implement mathematical models on a computer. Thus, we must generally rely on approximate solutions for model problems. However, improved numerical techniques and computational performance mean that we can find approximate solutions that are very close to the actual exact solution. The Finite Element Method (FEM) is one such technique for determining numerical solutions to the differential or integral equations that arise in models for systems in science and engineering.

The Finite Element Method was introduced in the early 1950s, with one of its earliest uses being for the stiffness analysis of delta airplane wings [Turner, et al., 1956]. At first the method was thought to be a generalization of previous methods used for structural analysis where the structure is subdivided into small parts, or so-called finite elements, with known simple behaviour. The actual term "Finite Element Method" was not adopted for the procedure until 1960 [Clough, 1960]. As the mathematics behind the method were studied further during the early 1960s it became clear that the method was in fact rooted in variational methods of mathematics introduced at the beginning of the 20th century, and was a general technique for determining numerical solutions of partial differential equations. Over the decades since, the method has been developed and refined into a general method for the numerical solution of partial differential equations and integral equations with applications in many areas of science and engineering. Today, finite element methods are still used extensively in structural engineering; however, they are also used for problems in fluid mechanics, nuclear engineering, electromagnetism, wave-propagation, heat conduction, convection-diffusion processes, reaction-diffusion processes, aerospace structures, integrated circuits, and many other areas. For a more detailed and interesting account of the early development of the method the reader is referred to [Clough, 1980].

The Method

In general terms, the idea behind any numerical method for solving differential equations is to discretize the given continuous problem to obtain a discrete problem. In other words, we must transform the continuous problem into a problem which consists of a system of equations with a finite number of unknowns which can then be solved computationally. The difference method is perhaps the most classical of numerical methods for partial differential equations. In the difference method the discrete problem is obtained by replacing derivatives with difference quotients involving the values of the unknowns at certain (finitely many) points. In the finite element method, the discretization process is somewhat different. The basic idea behind the finite element method is to reformulate the given differential equation as an equivalent variational problem. In order to better understand these ideas let us consider the following examples based on ones from [Cook, Malkus, and Plesha, 1989, pp.1-3].

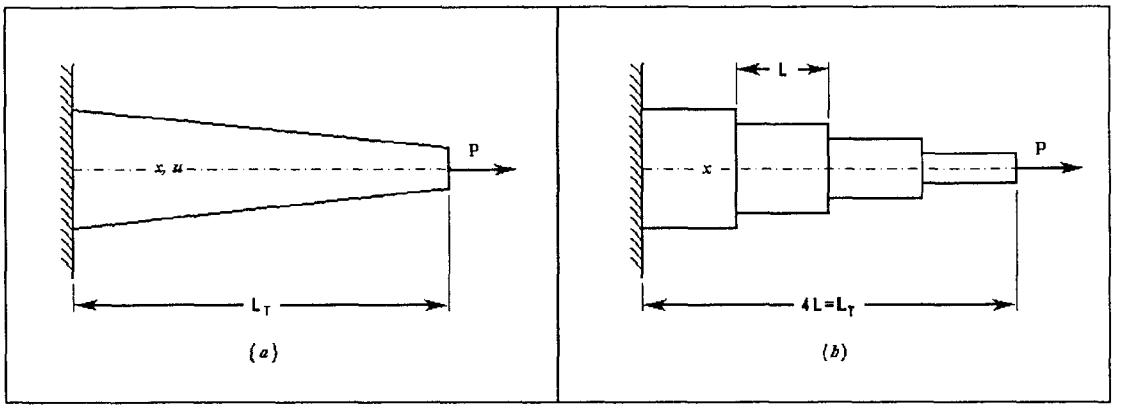


Figure 1.1.
(a) A tapered bar under end load P. (b) A model built of four uniform (non-tapered) elements of equal length.

Figure 1.1 gives us an example of how a continuous problem might be discretized. Suppose we wish to determine the displacement of the right end of the bar in Figure 1.1. To solve this problem using a more classical approach we would determine the differential equation of the continuous tapered bar, solve this equation for axial displacement u as a function of x and finally substitute $x = L_T$ to find the required end displacement. In the finite element method, rather than beginning with a differential equation, the bar is *discretized* by modeling it as a series of *finite elements*, each uniform but of a different cross-sectional area A (Figure 1.1b). In each element u varies linearly with x; therefore, for $0 < x < L_T$, u is a piecewise-smooth function of x. The elongation of each element can be determined from the elementary formula PL/AE. The end displacement, at $x = L_T$, is the sum of the element elongations. The accuracy of this model improves as more elements are used.

In general, the finite element method forms a model of a structure or system which consists of an assemblage of small parts or elements. If the geometry of each element is kept simple then it is much easier to analyze the model than the actual structure. In essence, we approximate a complicated solution by a model that consists of piecewise-continuous simple solutions. Elements are called "finite" to distinguish them from differential elements used in calculus.

In a heat transfer context, Figure 1.1 might represent a bar with insulated sides, prescribed temperature at the left end, and prescribed heat flow at the right end. One might ask for the temperature in the bar as a function of x and time.

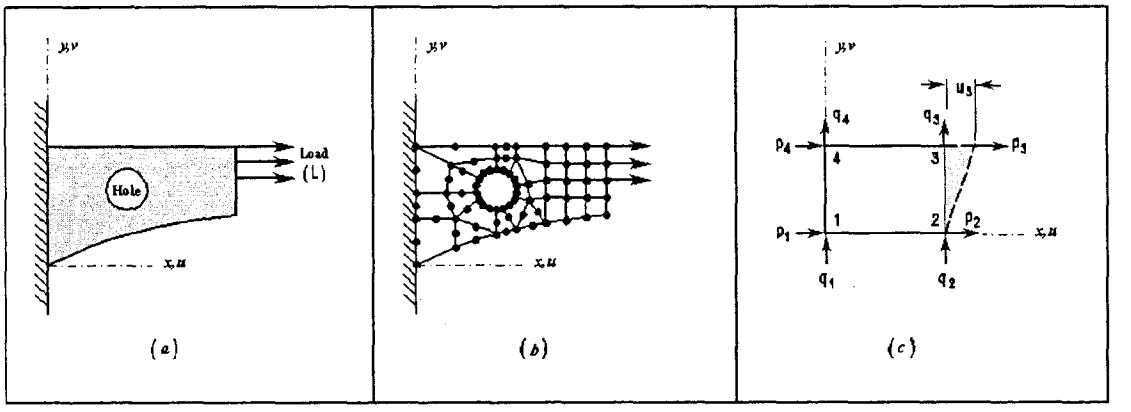


Figure 1.2.
(a) A plane structure of arbitrary shape. (b) a possible finite element model of the structure. (c) A plane rectangular element showing nodal forces p_i and q_i. The dashed line shows the deformation mode associated with x-direction displacement of node 3.

Figure 1.2 gives us a more complex example based on a plane structure. Suppose we wish to determine displacements and stresses caused by load L. Figure 1.2b gives a possible finite element model discretization of the problem. The model is constructed of elements that consist of plane areas, some triangular and some quadrilateral. As this example shows, it is often desirable to use a number of different types of elements in the discretization. This is particularly true when dealing with complicated models in three dimensional problems. If done properly, there is no difficulty in combining different element types; however, the areas where these different elements join do require special considerations, as well as the possible use of special mortar elements (see section 3). The black dots in the mesh of Figure 1.2b are called nodes or nodal points and they indicate where elements are connected to one another. In this model each node has two degrees of freedom (d.o.f.): that is, each node can displace in both the x and y direction. Thus, if there are n nodes in Figure 1.2b, there are $2n$ d.o.f. in the model. (In the real structure there are infinitely many d.o.f. because the structure has infinitely many particles.) Algebraic equations that describe the finite element model are solved to determine the d.o.f. Use of only $2n$ d.o.f. in analysis is similar to use of the first $2n$ terms of a convergent infinite series. (In heat transfer, each node has only one d.o.f. – the temperature of the node. Thus, a finite element model for heat transfer of n nodes has n d.o.f.) We see that in going from Figure 1.2a to 1.2b the distributed load L has been converted to concentrated forces at nodes. The analysis procedure gives a prescription for making this conversion, as will be shown subsequently.

From Figure 1.2 it may appear that discretization is accomplished simply by sawing the continuum into pieces and then pinning the pieces together again at node points. But such a model would not deform like the continuum. Under load, strain concentrations would appear at the nodes, and the elements would tend to overlap or separate along the saw cuts. Clearly, the actual structure does not behave this way, so the elements must be restricted in their deformation patterns. For example, if elements are allowed to have only such deformation modes as will keep edges straight (Figure 1.2c), then adjacent elements will neither overlap nor

separate. In this way we satisfy the basic requirement that deformations of a continuous medium must be compatible.

An important ingredient in a finite element analysis is the behaviour of the individual elements. A few good elements may produce better results than many poorer elements. We can see that several element types are possible by considering Figure 1.2b. A function ϕ, which might represent any of several physical quantities, varies smoothly in the actual structure. A finite element model typically yields a piecewise-smooth representation of ϕ. Between elements there may be jumps in the x and y derivatives of ϕ. Within each element ϕ is a smooth function that is usually represented by a simple polynomial.

2. Theoretical Background

2.1. Mathematical Concepts

Before proceeding further, let us review some of the underlying principles and concepts involved in the finite element method. We will restrict ourselves to a discussion of the finite element method as applied to boundary-value problems for linear elliptic equations. Also, for the sake of simplicity, we only consider second order problems. In particular, only Laplace's operator for two and three dimensional spaces will be considered, and proofs will only be made for the case where the domain has a Lipschitz-continuous boundary. In addition, in order to avoid unnecessary overcomplication, let us concentrate on functional spaces consisting of piecewise polynomial functions in two or three variables. For a more comprehensive discussion of finite element theory the reader is directed to [Kardestuncer, 1987], [Johnson, 1987], [Zienkiewicz, 1989].

We will begin by examining the functional spaces and related properties that are needed for developing finite element models for elliptic problems. After this we will present an important formula, Green's formula, that will be useful in our later analysis of the method. We will then turn our attention to a detailed examination of some of the fundamental concepts of the method.

Hilbert Spaces

In our development of variational formulations of boundary value problems for partial differential equations we need to specify the functional space, V, that we will be dealing with. In the cases we will be examining, it is most convenient to consider a space V that is somewhat larger (contains more functions) than the space of continuous functions with piecewise continuous derivatives. It will also be necessary to equip the space V with scalar products related to the boundary value problem. Specifically, V will be a Hilbert space.

Before giving a precise definition of a Hilbert space, let us review some basic principles from linear algebra:

For a linear space V, we say that L is a *linear form* on V if $L : V \to R$, i.e., $L(v) \in R$ for $v \in V$, and L is linear. L is considered to be linear if we have that for all $v, w \in V$ and $\beta, \theta \in R$:

$$L(\beta v + \theta w) = \beta L(v) + \theta L(w).$$

Also, we refer to $a(.,.)$ as a *bilinear form* on $V \times V$ if $a : V \times V \to R$, i.e., $a(v,w) \in R$ for $v, w \in V$, and a is linear in each argument. We consider a to be linear in each argument if we have for all $u, v, w \in V$ and $\beta, \theta \in R$:

$$a(u, \beta v + \theta w) = \beta a(u,v) + \theta a(u,w)$$

$$a(\beta u + \theta v, w) = \beta a(u,w) + \theta a(v,w).$$

The bilinear form $a(.,.)$ on $V \times V$ is said to be *symmetric* if

$$a(v,w) = a(w,v) \qquad \forall v, w \in V.$$

A symmetric bilinear form $a(.,.)$ on $V \times V$ is a *scalar product* on V if

$$a(v,v) > 0 \qquad \forall v \in V,\ v \neq 0.$$

The *norm* $\|\cdot\|_a$ associated with a scalar product $a(.,.)$ is defined by

$$\|v\|_a = (a(v,v))^{\frac{1}{2}}, \qquad \forall v \in V.$$

In general, if $\langle .,. \rangle$ is a scalar product with corresponding norm $\|\cdot\|$, then we have by Cauchy's inequality

$$|\langle v,w \rangle| \leq \|v\| \cdot \|w\|.$$

A space V is said to be a Hilbert space if V is a linear space with a scalar product and corresponding norm $\|\cdot\|$, and V is also complete. A space is considered to be complete if every Cauchy sequence with respect to the norm $\|\cdot\|$ converges.

We will now define some Hilbert spaces which are useful for variational formulations of the type of boundary value problems considered here. For simplicity, let us start with the one-dimensional case. Let $I = (a,b)$ be an interval, and define:

$$L_2(I) = \left\{ v : v \text{ is defined on } I \text{ and } \int_I v^2 dx < \infty \text{ (i.e., the integral exists)} \right\}.$$

In other words $L_2(I)$ is the space of square integrable functions on I. If we equip $L_2(I)$ with the scalar product

$$(v,w) = \int_I vw\, dx$$

and the corresponding norm (the L_2-norm):

$$\|v\|_{L_2(I)} = \sqrt{\int_I v^2\, dx} = (v,v)^{\frac{1}{2}}.$$

We have by Cauchy's inequality

$$|(v,w)| \leq \|v\|_{L_2(I)} \|w\|_{L_2(I)}.$$

Observe that the scalar product (v,w) is well defined since the integral (v,w) exists, if $v,w \in L_2(I)$. Thus, we have that the space $L_2(I)$ is a Hilbert space.

We will also define some other Hilbert spaces that will prove useful in the upcoming discussion. Define

$$H^1(I) = \left\{ v : v \text{ and } v' \text{ belong to } L_2(I) \right\}$$

and equip this space with the scalar product

$$(v,w)_{H^1(I)} = \int_I (vw + v'w')dx$$

and the corresponding norm

$$\|v\|_{H^1(I)} = \sqrt{\int_I (v^2 + (v')^2 dx}.$$

Note: here we use u' to denote the first derivative of u. Thus, the space $H^1(I)$ is comprised of the functions, v, which are defined on I and together with their first derivatives are square-integrable. In the consideration of boundary value problems of the form

$$-u'' = f \qquad \text{on } I = (a,b)$$

with boundary conditions

$$u(a) = u(b) = 0$$

it is convenient to define the space

$$H_0^1(I) = \{v \in H^1(I),\ v(a) = v(b) = 0\}$$

and equip it with the same scalar product and norm as $H^1(I)$.

These concepts are easily extended to cases of higher dimensionality. For instance, let Ω be a bounded domain in $R^d, d = 2$ or 3, and define

$$L_2(\Omega) = \{v : v \text{ is defined on } \Omega \text{ and } \int_\Omega v^2 dx < \infty\}$$

$$H^1(\Omega) = \{v \in L_2(\Omega) : \frac{\partial v}{\partial x_i} \in L_2(\Omega),\ i = 1,\ldots,d\}$$

with the corresponding scalar products and norms

$$(v, w) = \int_\Omega vw\, dx$$

$$\|v\|_{L_2(\Omega)} = \left(\int_\Omega v^2 dx\right)^{\frac{1}{2}}$$

$$(v, w)_{H^1(\Omega)} = \int_\Omega [vw + \nabla v \nabla w] dx$$

$$\|v\|_{H^1(\Omega)} = \left(\int_\Omega [v^2 + |\nabla v|^2] dx\right)^{\frac{1}{2}}$$

where ∇v denotes the gradient of v, i.e., $\nabla v = \left(\frac{\partial v}{\partial x_1}, \frac{\partial v}{\partial x_2}\right)$ for $d = 2$. Furthermore, we define the space

$$H_0^1(\Omega) = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma\}$$

where Γ is used to denote the boundary of Ω. We also equip $H_0^1(\Omega)$ with the same scalar product and norm as $H^1(\Omega)$.

Green's Formula

Let us now mention a certain Green's formula which will be useful in the sections that follow. Starting with divergence theorem (for two dimensional space) we have for a domain Ω with boundary Γ:

$$\int_\Omega div\, A\, dx = \int_\Gamma A \cdot n\, ds$$

where $A = (A_1, A_2)$ is a vector valued function defined on Ω,

$$div\, A = \frac{\partial A_1}{\partial x_1} + \frac{\partial A_2}{\partial x_2}$$

and $n = (n_1, n_2)$ is the outward unit normal to Γ. In this case dx denotes the element of area R^2 and ds the element of arc length along Γ. Applying the divergence theorem to $A = (vw, 0)$ and $A = (0, vw)$, we have:

$$\int_\Omega w\frac{\partial v}{\partial x_1} + v\frac{\partial w}{\partial x_1} dx = \int_\Gamma vwn_1 ds$$

and

$$\int_\Omega w\frac{\partial v}{\partial x_2} + v\frac{\partial w}{\partial x_2} dx = \int_\Gamma vwn_2 ds$$

respectively. Adding these two equations and rearranging terms, we have:

$$\int_\Omega w\frac{\partial v}{\partial x_1} + w\frac{\partial v}{\partial x_2}\, dx + \int_\Omega v\frac{\partial w}{\partial x_1} + v\frac{\partial w}{\partial x_2} dx = \int_\Gamma vwn_1 + vwn_2\, ds.$$

If we let $w = \frac{\partial w}{\partial x}$, and denote by ∇v the gradient of v, i.e., $\nabla v = \left(\frac{\partial v}{\partial x_1}, \frac{\partial v}{\partial x_2}\right)$, we then get the

following Green's formula:

$$\begin{aligned}\int_\Omega \frac{\partial w}{\partial x_1}\frac{\partial v}{\partial x_1} + \frac{\partial w}{\partial x_2}\frac{\partial v}{\partial x_2}\,dx &\equiv \int_\Omega \nabla v \cdot \nabla w\,dx \\ &= \int_\Gamma v\frac{\partial w}{\partial x_1}n_1 + v\frac{\partial w}{\partial x_2}n_2\,ds - \int_\Omega v\frac{\partial^2 w}{\partial x_1} + v\frac{\partial^2 w}{\partial x_2}\,dx \\ &= \int_\Gamma v\left(\frac{\partial w}{\partial x_1}n_1 + \frac{\partial w}{\partial x_2}n_2\right)ds - \int_\Omega v\left(\frac{\partial^2 w}{\partial x_1} + \frac{\partial^2 w}{\partial x_2}\right)dx \\ &= \int_\Gamma v\frac{\partial w}{\partial n}\,ds - \int_\Omega v\Delta w\,dx\end{aligned}$$

or,

$$\int_\Omega \nabla v \cdot \nabla w\,dx = \int_\Gamma v\frac{\partial w}{\partial n}\,ds - \int_\Omega v\Delta w\,dx \qquad (2.1.1)$$

where we have used the short form, $\frac{\partial w}{\partial n} = \frac{\partial w}{\partial x_1}n_1 + \frac{\partial w}{\partial x_2}n_2$ for the normal derivative, i.e., the derivative in the outward normal direction to the boundary Γ.

2.2. The Finite Element Model

In order to further aid our understanding and analysis of the method let us now consider some model problems. In the process we will also examine some well known results concerning the regularity of the solution, and some fundamental results concerning piecewise polynomial functions. For the time being we restrict ourselves to two dimensional space; however, these principals extend naturally to problems in three dimensional space. Also, unless specifically stated otherwise, it will be assumed throughout the following that the boundary Γ is Lipschitz-continuous.

Let Ω be a bounded convex plane domain with Lipschitz-continuous boundary Γ, and consider the boundary value problem:

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \Omega \\ u &= 0 \qquad \text{on } \Gamma \end{aligned} \qquad (2.2.1)$$

where Δ is the Laplacian operator, i.e., $\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}$. (This is the classical Dirichlet problem for Poisson's equation with homogeneous boundary conditions.) To help us in our discussion of the properties of the solution to this problem and others, we introduce some notation. Denote by $\|\cdot\|_0$ the L_2-norm over Ω and by $\|\cdot\|_k$ that in the Hilbert space $H^k(\Omega)$. Thus, for real-valued functions v,

$$\|v\|_0 = \left(\int_\Omega v^2 dx\right)^{\frac{1}{2}}$$

and for k a positive integer,

$$\|v\|_k = \left(\sum_{|\alpha|\le k}\|D^\alpha v\|_0^2\right)^{\frac{1}{2}}$$

where α is a multi-index (that is, $\alpha = (\alpha_1,\alpha_2)$, with α_1 and α_2 nonnegative integers), $|\alpha| = \alpha_1 + \alpha_2$, and $D^\alpha v \equiv \frac{\partial^{|\alpha|} v}{\partial x_1^{\alpha_1}\partial x_2^{\alpha_2}}$. For example,

$$\|v\|_2 = \left(\|v\|_0^2 + \left\|\frac{\partial v}{\partial x_1}\right\|_0^2 + \left\|\frac{\partial v}{\partial x_2}\right\|_0^2 + \left\|\frac{\partial^2 v}{\partial x_1^2}\right\|_0^2 + \left\|\frac{\partial^2 v}{\partial x_1 \partial x_2}\right\|_0^2 + \left\|\frac{\partial^2 v}{\partial x_2 \partial x_1}\right\|_0^2 + \left\|\frac{\partial^2 v}{\partial x_2^2}\right\|_0^2\right)^{\frac{1}{2}}$$

From a mathematical standpoint results for the finite element method are most easily derived using as our norms the $H^k(\Omega)$-norms. From a computational standpoint, pointwise estimates for the error are often more interesting, but they are much more difficult to prove than the average estimates provided by the $H^k(\Omega)$-norms.

The following lemma provides us with an important fact, which will be useful later in proving error estimates, it gives us a precise statement of what smoothness we may expect for the solution u of (2.2.1) given a certain degree of smoothness of f.

Lemma 2.1

Given any nonnegative integer k, there is a constant C such that for any $f \in H^k(\Omega)$ *with u the corresponding solution of* (2.2.1), *we have*

$$\|u\|_{k+2} \leq C\|f\|_k$$

Hence $u \in H^{k+2}(\Omega)$.

This fact can be found in [Johnson, 1987, pp.93], it states a well known regularity property associated with elliptic equations.

Also consider the following model Neumann problem:

$$\begin{aligned} -\Delta u + u &= f \qquad \text{in } \Omega \\ \tfrac{\partial u}{\partial n} &= 0 \qquad \text{on } \Gamma \end{aligned} \tag{2.2.2}$$

where $\frac{\partial u}{\partial n}$ is the outward normal derivative on Γ. Corresponding to this problem we have the following smoothing property [Kardestuncer, 1987, pp.1.157].

Lemma 2.2

Given any nonnegative integer k, there is a constant C such that for any $f \in H^k(\Omega)$ *with u the corresponding solution of* (2.2.2), *we have*

$$\|u\|_{k+2} \leq C\|f\|_k$$

Hence $u \in H^{k+2}(\Omega)$.

These fundamental facts concerning the regularity of solutions of elliptic boundary-value problems will provide us with the tools we need for analyzing the properties of the errors in finite element approximations.

In general, we want to approximate the solutions of (2.2.1) and (2.2.2) by certain piecewise polynomial functions defined on Ω. To avoid overcomplicating matters we will, for the time being, concentrate on piecewise linear functions.

Before determining the approximate solutions of (2.2.1) and (2.2.2), let us examine the problem of approximation of smooth functions on Ω. First we consider smooth functions which vanish on Γ, of which the solution u of (2.2.1) is a member. Let us proceed then by considering the construction of a finite dimensional subspace V_h of V, consisting of a triangulation of Ω. For simplicity we assume that Γ is a polygonal curve, in which case Ω is a polygonal domain. If this is not the case, i.e., if Γ is curved, we may first approximate Γ with a polygonal curve.

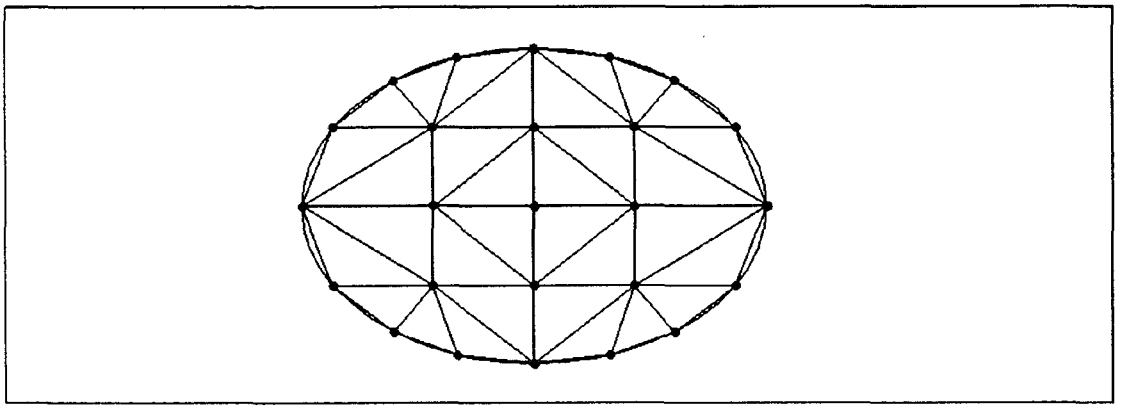


Figure 2.1.
Triangulation of a convex domain Ω.

Let $T_h = \{\tau_1, \ldots, \tau_m\}$ denote a partitioning of Ω into non-overlapping triangles τ_i, such that no vertex of any triangle lies on a side of another triangle and such that the union of the triangles determines a polygonal domain $\Omega_h \subset \Omega$ (where Ω is convex) whose boundary vertices lie on Γ (Figure 2.1). In other words, we have that

$$\Omega_h = \bigcup_{\tau \in T_h} \tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_m.$$

Let h denote the maximal length of a side in the triangulation T_h, i.e., the length of the longest side for the triangle τ with the longest side in T_h, or

$$h = \max_{\tau \in T_h}(diam(\tau))$$

where $diam(\tau)$ = diameter of τ = longest side of τ. Thus, h is a parameter which decreases as the triangulation is made finer. Let us assume that the angles of the triangulation are bounded below, independently of h and also that the triangulations are quasi-uniform in the sense that the triangles τ of T_h are of essentially the same size, which may be expressed by demanding that the area of any triangle τ in T_h is bounded below by ch^2 with $c > 0$ independent of h. Now define

$$V_h = \{v : v \text{ is continuous on } \Omega_h,\ v|_\tau \text{ is linear for } \tau \in T_h,\ v = 0 \text{ on } \Gamma\}$$

where $v|_\tau$ denotes the restriction of v to triangle τ, i.e., the function defined on τ agreeing with v on τ. The space V_h consists of all continuous functions that are linear on each triangle τ and vanish on Γ. Note that $V_h \subset V$. Let N_h be the number of interior vertices in the triangulation of Ω, and let $\{P_j\}_{j=1}^{N_h}$ be the interior vertices of T_h (we exclude the boundary nodes, since $v = 0$ on Γ). A function in $v \in V_h$ is uniquely determined by its value, $v(P_j)$, at the points P_j and thus depends on N_h parameters. Let ϕ_j be a function in V_h which takes the value 1 at P_j but vanishes at the other vertices, i.e.,

$$\phi_j(P_i) = \delta_{ij} \equiv \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \quad i, j = 1, \ldots, N_h \end{cases}$$

It can be seen that the support of ϕ_j (the set of points x for which $\phi_j(x) \neq 0$) consists of the

triangles with the common node P_j. Thus, $\{\phi_j\}_{j=1}^{N_h}$ forms a basis for V_h, and every χ in V_h admits the representation:

$$\chi(x) = \sum_{j=1}^{N_h} \alpha_j \phi_j(x) \qquad \text{with } \alpha_j = \chi(P_j),\ x \in \Omega_h \cup \Gamma.$$

Given a smooth function v on Ω which vanishes on Γ, we can interpolate v into V_h by defining the interpolant of v, $I_h v$ in V_h as that function in V_h satisfying

$$I_h v(P_j) = v(P_j) \qquad \text{for } j = 1, \ldots N_h$$

Hence

$$I_h v(x) = \sum_{j=1}^{N_h} v(P_j)\phi_j(x).$$

The following error estimates for the interpolant $I_h v$ are well known [Johnson, 1987], [Bramble and Zlamal, 1970]. For a triangulation T_h and piecewise polynomials of degree $r \geq 1$, we have

$$\|I_h v - v\|_0 \leq Ch^{r+1}\|v\|_{r+1} \qquad (2.2.3a)$$

and

$$\|\nabla I_h v - \nabla v\|_0 \leq Ch^r\|v\|_{r+1} \qquad (2.2.3b)$$

where,

$$\|\nabla v\|_0 = \left(\left\| \frac{\partial v}{\partial x_1} \right\|_0^2 + \left\| \frac{\partial v}{\partial x_2} \right\|_0^2 \right)^{\frac{1}{2}}$$

and we assume that v is sufficiently regular so that $\|v\|_2$ is finite. The orders of these estimates, $O(h^2)$ and $O(h)$, respectively are the optimal orders to which the functions and their gradients can be approximated in V_h in the sense that the powers of h in $(2.2.3a)$ cannot be increased with $(2.2.3b)$ remaining valid for a fixed C and all $v \in H^2(\Omega)$ vanishing on Γ. We will soon see that the piecewise linear Galerkin approximation to the solution of $(2.2.1)$ satisfies similar inequalities.

In considering the approximation of the solution of $(2.2.2)$ we must consider sets of functions V_h which do not necessarily vanish on Γ. However, details of such functions and their properties are beyond the scope of this paper. Please refer to [Zienkiewicz, 1989], [Kardestuncer, 1987] for a more complete discussion.

The Ritz Approximation for Dirichlet's Problem

Let us take note of the fact that in simple cases of elliptic equations the variational problem can also be considered as a minimization problem corresponding to the classical Ritz-Galerkin method that goes back to the beginning of the 20th century. In the case of more general formulations they correspond to Galerkin methods. In the case of elliptic equations, for example, this variational problem in basic cases is a minimization problem of the form:

(M) Find $u \in V$ such that $F(u) \leq F(v)$ for all $v \in V$

where V is a given set of admissible functions and $F : V \to R$ is a functional (i.e., $F(v) \in R$ for all $v \in V$ with R denoting the set of real numbers). As we have seen, the functions v in V often represent a continuously varying quantity such as displacement in an elastic body, a

temperature, etc. $F(v)$ is the total energy associated with v and (M) corresponds to an equivalent characterization of the solution of the differential equation as the function in V that minimizes the total energy of the considered system. In general the dimension of V is infinite (i.e., the functions in V cannot be described by a finite number of parameters) and thus in general the problem (M) cannot be solved exactly. To obtain a problem that can be solved computationally we replace the set V with a set V_h which consists of simple functions dependent on only finitely many parameters. This leads to a finite-dimensional minimization problem of the form:

(M_h) Find $u_h \in V_h$ such that $F(u_h) \leq F(v)$ for all $v \in V_h$.

This problem is equivalent to a large linear or nonlinear system of equations. The hope is now that the solution u_h of (M_h) is a sufficiently good approximation of the solution u of (M), the original partial differential equation. Generally V_h is chosen to be a subset of V, in other words $V_h \subset V$, i.e., if $v \in V_h$ then $v \in V$. In this case (M_h) corresponds to the classical Ritz-Galerkin method. The special feature of a finite element method as a particular Ritz-Galerkin method is the fact that the functions in V_h are chosen to be piecewise polynomial. One may also start from more general variational formulations than the minimization problem (M) and this corresponds to the so-called Galerkin methods.

To define a Ritz approximation of (2.2.1) we first multiply the equation by a smooth function ϕ which vanishes on Γ,

$$-\Delta u\phi = f\phi$$

now integrate over Ω,

$$-\int_\Omega \Delta u\phi \, dx = \int f\phi \, dx.$$

Next we apply Green's formula (2.1.1),

$$-\int_\Omega \Delta u\phi \, dx = \int \nabla u \cdot \nabla\phi \, dx = \int f\phi \, dx$$

since u and ϕ vanish on Γ. Thus, we obtain, for all such ϕ, with (v,w) denoting the inner product $\int_\Omega vw \, dx$ in $L_2(\Omega)$,

$$(\nabla u, \nabla\phi) = (f, \phi). \qquad (2.2.4)$$

We may then pose the approximate problem to find u_h in V_h such that

$$(\nabla u_h, \nabla\chi) = (f, \chi) \qquad \text{for all } \chi \in V_h. \qquad (2.2.4b)$$

In terms of the basis $\{\phi_j\}_{j=1}^{N_h}$ introduced above, we may restate the problem as follows:

Find the coefficients $\xi_1, \ldots, \xi_{N_h}$ defining

$$u_h(x) = \sum_{j=1}^{N_h} \xi_j \phi_j(x)$$

such that

$$\sum_{j=1}^{N_h} \xi_j (\nabla\phi_j, \nabla\phi_k) = (f, \phi_k) \qquad k = 1, \ldots, N_h \qquad (2.2.5)$$

which is a linear system with N_h equations in N_h unknowns, $\xi_1, \ldots, \xi_{N_h}$. In matrix form we

have

$$A\xi = b \qquad (2.2.6)$$

with symmetric $N_h \times N_h$ matrix $A = (a_{jk})$ with $a_{jk} = (\nabla\phi_j, \nabla\phi_k)$, vector $b = (b_k)$ with $b_k = (f, \phi_k)$, and vector $\xi = (\xi_k)$ with components $\xi_k = (u_h(P_k))$. The matrix A is commonly referred to as the stiffness matrix, and the vector b is termed the load vector. When the bilinear form is symmetric, as it is in our case, the matrix A is symmetric and positive-definite, and thus non-singular so that (2.2.6) admits a unique solution ξ (we will examine this further in a moment). (Contrast this to matrices arising from finite-difference methods over other than rectangular regions.) This is an advantage for the numerical solution of system (2.2.6). In the choice of the basis $\{\phi_j\}_{j=1}^{N_h}$, it is of paramount importance, again from a numerical standpoint, that the resulting matrix possess as many zeros as possible.

In practice the elements $a_{jk} = (\nabla\phi_j, \nabla\phi_k)$ of stiffness matrix A are computed by summing the contributions from the different triangles:

$$(\nabla\phi_j, \nabla\phi_k) = \sum_{\tau \in T_h} (\nabla\phi_j, \nabla\phi_k)_\tau$$

where

$$(\nabla\phi_j, \nabla\phi_k)_\tau = \int_\tau \nabla\varphi_j \cdot \nabla\varphi_k \, dx.$$

Note that $(\nabla\phi_j, \nabla\phi_k)_\tau = 0$ unless P_i and P_j are both vertices of τ. Thus, A is also a sparse matrix. Let P_i, P_j, P_k be the vertices of triangle τ. Then the 3×3 matrix:

$$\begin{bmatrix} (\nabla\phi_i, \nabla\phi_i)_\tau & (\nabla\phi_i, \nabla\phi_j)_\tau & (\nabla\phi_i, \nabla\phi_k)_\tau \\ & (\nabla\phi_j, \nabla\phi_j)_\tau & (\nabla\phi_j, \nabla\phi_k)_\tau \\ sym & & (\nabla\phi_k, \nabla\phi_k)_\tau \end{bmatrix},$$

is the element stiffness matrix for τ. Thus, the global stiffness matrix A can be computed by first computing each element stiffness matrix and summing the contributions for each triangle. To compute the element stiffness matrix we work with the restrictions of the basis functions ϕ_i, ϕ_j and ϕ_k to the triangle τ. Denote these restrictions on τ as ψ_i, ψ_j, ψ_k, so that each ψ is a linear function on τ that takes the value one at one vertex and zero at the other two vertices. The functions ψ_i, ψ_j, ψ_k then form the basis functions on triangle τ. Thus, a linear function w on τ has the representation

$$w(x) = w(P_i)\psi_i(x) + w(P_j)\psi_j(x) + w(P_k)\psi_k(x) \qquad x \in \tau.$$

Let us now prove that (2.2.4b) has a unique solution u_h in V_h. Consider two solutions u_1^h and u_2^h of (2.2.4b). Then we have

$$(\nabla u_1^h, \nabla\chi) = (f, \chi) \qquad \text{for all } \chi \in V_h$$

$$(\nabla u_2^h, \nabla\chi) = (f, \chi) \qquad \text{for all } \chi \in V_h$$

Subtracting these, and choosing $\chi = u_1^h - u_2^h \in V_h$, we have

$$\int_\Omega \nabla u_1^h \nabla\chi \, dx - \int_\Omega \nabla u_2^h \nabla\chi \, dx = (f, \chi) - (f, \chi)$$

$$\int_\Omega \nabla(u_1^h - u_2^h)\nabla(u_1^h - u_2^h) \, dx = 0.$$

It follows that $(u_1^h - u_2^h)(x)$ is constant on Ω. This, along with the boundary condition $u_1^h = u_2^h = 0$ on Γ gives us $u_1^h(x) = u_2^h(x)$, $\forall x \in \Omega$, and so the solution is unique. This further proves that the matrix A in (2.2.6) is nonsingular so that (2.2.6) is uniquely solvable, given any right-hand side b.

Since (2.2.6) is uniquely solvable we can compare the finite element solution

$$u_h = \sum_{i=1}^{N_h} u_h(P_i)\phi_i(x)$$

with the solution u of (2.2.1). From (2.2.6), we know that

$$\xi = \left[u_h(P_1)\, u_h(P_2) \ldots u_h(P_{N_h})\right]^T,$$

can be obtained by solving the $N_h \times N_h$ matrix equation (2.2.6). Thus, once the matrix equation is solved, u_h can easily be determined. The easiest comparison between u_h and u is given with respect to the norm $\|\cdot\|_1$. Before we make this comparison, we present the following theorem from [Kardestuncer, 1987, pp.1.160].

Theorem 2.1

$$\|\nabla(u_h - u)\|_0 \le Ch^{s-1}\|u\|_s \qquad for\ s = 1,2.$$

Two important facts that this theorem gives us are as follows:

1. The finite element solution u_h is the best approximation in V_h (in the sense of the norm $\|\nabla \cdot\|_0$ on V_h) of the function u; in other words, u_h is the orthogonal projection of u onto V_h (with norm $\|\nabla \cdot\|_0$).
2. The finite element solution u_h may be computed directly from the "data" f.

The above theorem shows that the Ritz approximation u_h imitates the optimality property of the interpolant (2.2.3b), which is, of course, natural in view of fact 1. Note here the fact that u minimizes the functional $\frac{1}{2}\|\nabla v\|_0^2 - (f, v)$ for all v in $H^1(\Omega)$ which vanish on Γ. This property is imitated by u_h in that u_h minimizes the same functional for functions in V_h. Now let us examine a result presented in [Kardestuncer, 1987, pp.1.161] which show that the Ritz approximation has an optimality property with respect to the $L_2(\Omega)$-norm. Thus, the Ritz approximation has a property analogous to (2.2.3a).

Theorem 2.2

$$\|u_h - u\|_0 \le Ch^s\|u\|_s \qquad for\ s = 1,2.$$

2.3. Finite Element Spaces

Let us now examine some common finite element spaces V_h. These spaces will consist of piecewise polynomial functions on subdivisions, or "triangulations", $T_h = \{\tau\}$ of a bounded domain $\Omega \subset R^d$, $d = 1,2,3$, into elements τ. For $d = 1$, the elements τ are intervals, for $d = 2$, triangles or quadrilaterals, and for $d = 3$, tetrahedra and hexahedra.

The finite element space will need to satisfy either $V_h \subset H^1(\Omega)$ or $V_h \subset H^2(\Omega)$ corresponding to either second order or fourth order boundary value problems, respectively.

Since the space V_h consists of piecewise polynomials, we have

$$V_h \subset H^1(\Omega) \Leftrightarrow V_h \subset C^0(\overline{\Omega}) \qquad (2.3.1)$$

$$V_h \subset H^2(\Omega) \Leftrightarrow V_h \subset C^1(\overline{\Omega}) \qquad (2.3.2)$$

where, $\overline{\Omega} = \Omega \cup \Gamma$ and

$$C^0(\overline{\Omega}) = \{v : v \text{ is a continuous function defined on } \overline{\Omega}\}$$

$$C^1(\overline{\Omega}) = \{v \in C^0(\overline{\Omega}) : D^\alpha v \in C^0(\overline{\Omega}), |\alpha| = 1$$
(i.e., first derivative is continuous on $\overline{\Omega}$)}.

Thus, $V_h \subset H^1(\Omega)$ if and only if the functions $v \in V_h$ are continuous, and $V_h \subset H^2(\Omega)$ if and only if the functions $v \in V_h$ and their first derivatives are continuous. The equivalence (2.3.1) depends on the fact that functions v in V_h are polynomials on each τ so that if v is continuous across the common boundary of adjoining elements, then the first derivatives, $D^\alpha v, |\alpha| = 1$, exist and are piecewise continuous so that $v \in H^1(\Omega)$. On the other hand, if v is not continuous across a certain inter-element boundary, i.e., $v \notin C^0(\overline{\Omega})$, then the derivatives $D^\alpha v, |\alpha| = 1$ do not exist as functions in $L_2(\Omega)$ and thus $v \notin H^1(\Omega)$. (If v is discontinuous across an element side S, then $D^\alpha v, |\alpha| = 1$, would be a δ-function supported by S which is not a square-integrable function.) In a similar way we see that (2.3.2) holds.

Until now we have mentioned only triangles when dealing with the subdivision of the domain, but there is no reason why we cannot use elements of other shapes as well. Although we will continue to deal with triangles in the following, we will be discussing other element types in future. Thus, we will now use K rather then τ to denote individual elements in the subdivision T_h of Ω. (K is used to denote the more general convex hull.) Let us now turn to the task of constructing a finite element space.

To define a finite element space V_h we must specify:

1. The subdivision (triangulation) $T_h = \{K\}$ of domain Ω.
2. The nature of the functions $v \in V_h$ on each K.
3. The parameters used to describe the functions in V_h.

n-Simplex Elements

As was the case previously, when triangles were used, the domain $\Omega \subset R^n$ with boundary Γ is subdivided by $T_h = \{K_1, \ldots, K_m\}$ of non-overlapping elements K_i, such that

$$\Omega_h = \bigcup_{K \in T_h} K = K_1 \cup K_2 \cup \ldots \cup K_m$$

with $\Omega_h \subset \Omega$.

Although other types of functions are possible it is advisable to restrict ourselves to using polynomials. There are two reasons for this: First, this fact was used previously in proving the convergence of the method, and secondly, this yields a simple calculation for the coefficients of the linear system. Also, up to this point, we have only discussed linear polynomials; however, there is no reason why we should restrict ourselves to polynomials of the first degree. Thus, we introduce the following notation:

$$P_r(K) = \{v : v \text{ is a polynomial of degree } \leq r \text{ on } K\}, \text{ for } r = 1, 2, \ldots$$

Thus, $P_1(K)$ is the now familiar space of linear functions on K, of the form

$$v(x) = a_{00} + a_{10}x_1 + a_{01}x_2,\ x \in K \text{ where } a_{ij} \in R.$$

Further we see that $P_2(K)$ is the space of quadratic functions on K. In general, we have:

$$P_r(K) = \left\{ v : v(x) = \sum_{0 \le i+j \le r} a_{ij} x_1^i x_2^j \text{ for } x \in K, \text{ where } a_{ij} \in R \right\}$$

Note that

$$\dim(P_k) = \binom{n+k}{k}$$

where $\binom{m}{r}$ is given by

$$\binom{m}{r} = \frac{m!}{r!(m-r)!},$$

and dim denotes the dimension of a linear space. In the linear case, we see that $\{\psi_1, \psi_2, \psi_3\}$, where $\psi_1(x) \equiv 1$, $\psi_2(x) = x_1$, $\psi_3(x) = x_2$, is a basis for $P_1(K)$, and that $\dim P_1(K) = 3$.

Recall that in R^n, a n-simplex is a convex hull $K$of $n+1$ points $a_j = (a_{ij})_{i=1}^n \in R^n$, which are then called the vertices of the n-simplex, provided the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n+1} \\ a_{21} & a_{22} & \cdots & a_{2n+1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn+1} \\ 1 & 1 & \cdots & 1 \end{bmatrix} \qquad (2.3.3)$$

is regular (equivalently, the $n+1$ points a_j are not contained in a hyperplane). Thus,

$$K = \left\{ x = \sum_{j=1}^{n+1} \lambda_j a_j;\ 0 \le \lambda_j \le 1,\ 1 \le j \le n+1,\ \sum_{j=1}^{n+1} \lambda_j = 1 \right\}.$$

Notice that a 2-simplex is a triangle and a 3-simplex is a tetrahedron. The barycentric coordinates $\lambda_j = \lambda_j(x)$, $1 \le j \le n+1$, of any point $x \in R^n$ with respect to the $n+1$ points a_j are the unique solutions of the linear system

$$\sum_{j=1}^{n+1} a_{ij}\lambda_j = x_i \qquad 1 \le i \le n$$

$$\sum_{j=1}^{n+1} \lambda_j = 1$$

whose matrix is precisely the matrix A of (2.3.3). By inverting this linear system, we see that the barycentric coordinates are affine functions of $x_1, x_2, \ldots, x_n$:

$$\lambda_i = \sum_{j=1}^{n} b_{ij}x_j + b_{in+1} \qquad 1 \le i \le n+1$$

where the matrix $B = (b_{ij})$ is the inverse of the matrix A in (2.3.3). Since $\lambda_i(a_j) = \delta_{ij}$, $1 \le i$,

$j \leq n+1$, we have the identity

$$p = \sum_{i=1}^{n+1} p(a_i)\lambda_i \qquad \text{for all } p \in P_1.$$

Therefore, a polynomial of degree ≤ 1 is uniquely determined by its values at the $n+1$ points a_j. This observation leads to the definition of the simplest finite element which we shall call n-simplex of type 1. In this case the space P_K is P_1, and the set Σ_K of degrees of freedom, i.e., those parameters which uniquely define a function in the space P_K, consists of the values at the vertices, which we write symbolically as $\Sigma_K = \{p(a_i),\ 1 \leq i \leq n+1\}$; see Figure 2.2 for the case where $n = 2$.

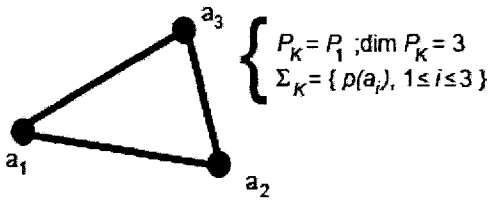


Figure 2.2. Triangle of type 1.

We now consider the case where the space of function is P_2. Let us define $a_{ij} = \frac{(a_i+a_j)}{2}$ as the midpoints of the edges of the n-simplex K. Since $\lambda_k(a_{ij}) = \frac{1}{2}(\delta_{ki} + \delta_{kj})$, we can establish the identity

$$p = \sum_{i=1}^{n+1} \lambda_i(2\lambda_i - 1)p(a_i) + \sum_{i<j} 4\lambda_i\lambda_j p(a_{ij}) \qquad \text{for all } p \in P_2.$$

This yields the definition of a finite element, called the n-simplex of type 2: the space P_K is P_2, and the set Σ_K consists of the values at the vertices and at the midpoints of the edges. See Figure 2.3 for the case where $n = 2$.

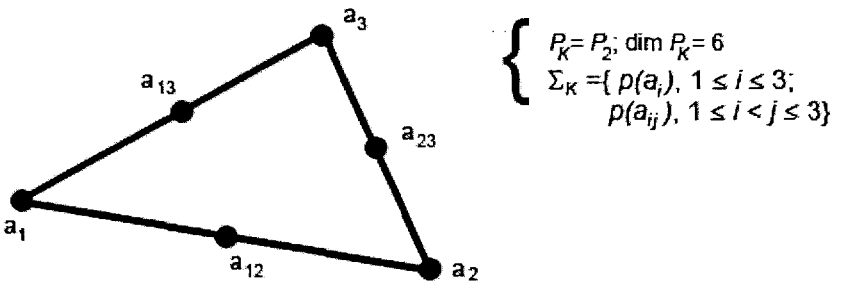


Figure 2.3. Triangle of type 2

Similarly we can deduce a definition for the n-simplex of type 3 having two points along each edge between the vertices, as well as a point in the centre of a face. In the case where $n = 2$ we have a triangle with a total of 10 points.

With a given triangulation, we now associate in a natural way a space V_h with each type of finite element. With triangles of type 1, a function of V_h is:

1. In the space $P_K = P_1$ for each $K \in T_h$.
2. Completely determined by its values at all the vertices of the triangulation, by definition.

Likewise, for triangles of type 2, a function of V_h is:

1. In the space $P_K = P_2$ for each $K \in T_h$.
2. Completely determined by its values at all the vertices and all the edge midpoint of the triangulation.

In other words, a function in V_h is specified by a set Σ_h of degrees of freedom, the function's values at all the vertices for triangles of type 1, vertices and edge midpoints for triangles of type 2, etc., in such a way that

$$\Sigma_h = \bigcup_{K \in T_h} \Sigma_K.$$

Let us now consider an example of how to determine the nodal basis functions for $P_1(K)$ associated with the degrees of freedom, i.e., the basis function $\phi_i \in P_1(K)$, $i = 1,2,3$, such that

$$\phi_i(a_j) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \quad i,j = 1,2,3. \end{cases}$$

A function $v \in P_1(K)$ then has the representation

$$v(x) = \sum_{i=1}^{3} v(a_i)\phi_i(x) \qquad x \in K.$$

Consider the following example from [Kwon and Bang, 1997, p86-88] with triangular elements defined on the x,y plane. The linear triangular element shown in Figure 2.4 has three nodes, one at each of the vertices of the triangle, and the variable interpolation within the element is linear in x and y, i.e.,

$$v = a_1 + a_2 x + a_3 y \qquad (2.3.4)$$

or, equivalently,

$$v = \begin{bmatrix} 1 & x & y \end{bmatrix} \begin{bmatrix} a_1 \\ a_3 \\ a_3 \end{bmatrix} \qquad (2.3.5)$$

where, a_i are constants to be determined. The interpolation function (2.3.4), should represent the nodal variables at the three nodal points. Substituting the x,y values at each nodal point gives:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_3 \\ a_3 \end{bmatrix} \qquad (2.3.6)$$

where, x_i and y_i are the coordinate values at the ith node and v_i is the nodal variable as seen in Figure 2.4.

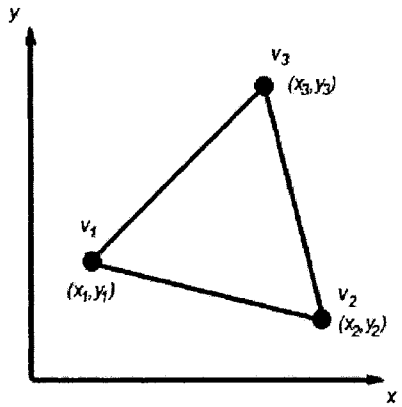


Figure 2.4.

Inverting the matrix and rewriting (2.3.6) gives

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \frac{1}{2A}\begin{bmatrix} x_2y_3 - x_3y_2 & x_3y_1 - x_1y_3 & x_1y_2 - x_2y_1 \\ y_2 - y_3 & y_3 - y_1 & y_1 - y_2 \\ x_3 - x_2 & x_1 - x_3 & x_2 - x_1 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \qquad (2.3.7)$$

where

$$A = \tfrac{1}{2}\det\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}$$

Thus, the magnitude of A is equal to the area of the triangular element. (Note that its value is positive if the node numbering is counter-clockwise and negative otherwise.) For the finite element computation, the element nodal sequence must be in the same direction for every element in the domain. Substitution of (2.3.7) into (2.3.5) gives

$$v = \phi_1(x,y)v_1 + \phi_2(x,y)v_2 + \phi_3(x,y)v_3$$

where $\phi_i(x,y)$, $i = 1,2,3$ are the shape functions for a linear triangular element:

$$\phi_1(x,y) = \frac{1}{2A}[(x_2y_3 - x_3y_2) + (y_2 - y_3)x + (x_3 - x_2)y]$$

$$\phi_2(x,y) = \frac{1}{2A}[(x_3y_1 - x_1y_3) + (y_3 - y_1)x + (x_1 - x_3)y]$$

$$\phi_1(x,y) = \frac{1}{2A}[(x_1y_2 - x_2y_1) + (y_1 - y_2)x + (x_2 - x_1)y]$$

The functions $\phi_i(x,y)$ then satisfy the conditions

$$\phi_i(x,y) = \delta_{ij}$$

and

$$\sum_{i=1}^{3}\phi_i = 1$$

For example, if the triangle K has vertices at: $(1,0),(0,1)$ and $(0,0)$ then

$$\phi_1 = x$$
$$\phi_2 = y$$
$$\phi_3 = 1 - x - y$$

since $A = \frac{1}{2}$.

Hypercube Elements

Another useful shape for finite elements in two dimensional space is the rectangle. In particular, if it so happens that the set $\overline{\Omega}$ is a rectangle then it may be conveniently "triangulated" by finite elements which are rectangles with sides parallel to the sides of $\overline{\Omega}$. In the case of three dimensional space, these ideas extend naturally to hexahedral elements. For simplicity we will restrict our discussion to hypercubes (squares in two dimensional space, cubes in three dimensional space).

In the following we let Q_k denote the space of all polynomials of degree $\leq k$ with respect to each of the n variables $x_1, x_2, \ldots, x_n$; i.e., a polynomial $p \in Q_k$ is of the form

$$p(x_1, x_2, \ldots, x_n) = \sum_{\alpha_i \leq k} a_{\alpha_1 \alpha_2 \ldots \alpha_n} x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n} \qquad 1 \leq i \leq n, \ \sum \alpha_i \leq k$$

Observe that $\dim(Q_k) = (k+1)^n$ and that we have the inclusions

$$P_k \subset Q_k$$

Note that, for example when $k = 1$, we call the functions in P_1 linear functions of $x_1, x_2, \ldots, x_n$ and the functions in Q_1 bilinear functions of $x_1, x_2, \ldots, x_n$. Denoting by K the unit hypercube $[0,1]^n$ of R^n, we define its subset

$$\Xi_k = \left\{ x = \left(\frac{i_1}{k}, \frac{i_2}{k}, \ldots, \frac{i_n}{k} \right) \in R^n;\ i_j \in \{0, 1, \ldots, k\},\ 1 \leq j \leq n \right\},$$

for any given integer $k \geq 1$. In view of the identity

$$p = \sum_{\substack{0 \leq i_j \leq k \\ 1 \leq j \leq n}} \prod_{j=1}^{n} \left(\prod_{\substack{i_j' = 0 \\ i_j' \neq i_j}}^{k} \frac{k x_j - i_j'}{i_j - i_j'} \right) p\left(\frac{i_1}{k}, \frac{i_2}{k}, \ldots, \frac{i_n}{k} \right) \qquad \text{for all } p \in Q_k$$

we deduce the definition of finite elements which we call hypercubes of type k. The following figures show the cases $k = 1, 2$ with $n = 2$, as well as the notation used for the points of the corresponding sets Ξ_k.

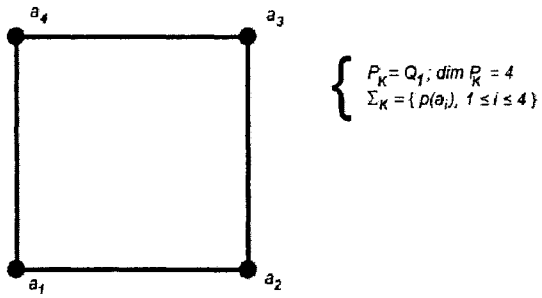


Figure 2.5. Square of type 1.

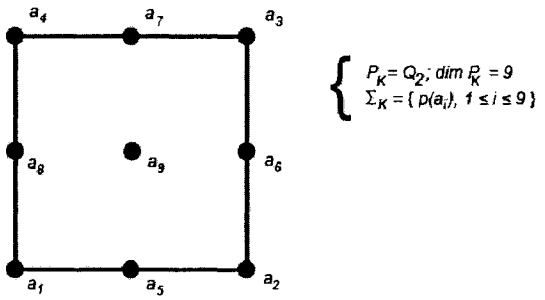


Figure 2.6. Square of type 2.

For example, let K be a square with vertices a_i, $i = 1,\ldots,4$, and define

$$Q_1(K) = \{v : v \text{ is bilinear on } K, \text{ i.e.,}$$
$$v(x) = a_{00} + a_{10}x_1 + a_{01}x_2 + a_{11}x_1x_2,\ x \in K, \text{ and } a_{ij} \in R\}$$

Then a function $v \in Q_1(K)$ is uniquely determined by the values $v(a_i)$, $i = 1,\ldots,4$. Further, if K_1 and K_2 are two squares in T_h with the common side S and the functions $v_1|_{K_1}$ and $v_2|_{K_2}$ agree at the end points of S then $v_1 - v_2 \equiv 0$ on S since $v_1 - v_2$ varies linearly on S. We may now define

$$V_h = \left\{v \in C^0(\overline{\Omega}) : v|_K \in Q_1(K),\ \forall K \in T_h\right\}$$

assuming that $T_h = \{K\}$ is a subdivision of Ω into non-overlapping squares such that no vertex of any square lies on a side of another square. Thus, values at the nodes may be used as global degrees of freedom.

General Properties of Finite Elements

With the previous examples in mind, we can now give a general definition of a finite element. A finite element in R^n is a triple (K,P,Σ) where the data K, P, and Σ have the following significant relationships:

1. K is a compact subset of R^n with a non-empty interior and a Lipschitz-continuous boundary.
2. P is a finite-dimensional space of real-valued functions defined over the set K, of dimension N.
3. Σ is a set of N linear forms $\varphi_i, 1 \leq i \leq N$, defined over the space P, in such a way that the set Σ is P-unisolvent in the sense that given any real scalars α_i, $1 \leq i \leq N$, there exists a unique function $p \in P$ which satisfies

   $$\varphi_i(p) = \alpha_i \qquad 1 \leq i \leq N$$

   Equivalently, there exist N functions $p_i \in P$, $1 \leq i \leq N$, which satisfy

   $$\varphi_j(p_i) = \delta_{ij} \qquad 1 \leq i \leq N$$

   which are called the basis functions of the finite element, since we have the identity

   $$p = \sum_{i=1}^{N} \varphi_i(p) p_i \qquad \text{for all } p \in P$$

In light of the definition of a finite element, let us go back over the examples seen previously. We have seen that K could be a n-simplex in R^n, i.e., a triangle in R^2 or a tetrahedron in R^3, or K could be a hypercube in R^n, i.e., a square in R^2 or a cube in R^3. These are special cases of straight finite elements, i.e., finite elements for which the set is a polyhedron in R^n. There exist also curved finite elements, i.e., those whose boundary is composed of curved surfaces; however such elements are beyond the scope of this work. Commonly, the sets Σ of degrees of freedom consist of the linear forms:

$$\varphi_{ij} : p \to p(a_{ij})$$

where the points a_{ij} are the points belonging to the finite element. For example, in the case of the linear triangle we have $1 \leq i \leq 3$, and the index j is dropped. For the quadratic triangle we again have $1 \leq i \leq 3$ for the vertices, and we have $1 \leq i < j \leq 3$ for the midpoints of edges. Similarly, for the rectangular elements we have $0 \leq i, j \leq 1$, and $0 \leq i, j \leq 2$, for the linear and quadratic cases respectively. Note that degrees of freedom can also be associated with partial derivatives on the element. Such elements are referred to as Hermite finite elements. However, in this work we confine ourselves to only Lagrange finite elements, elements not consisting of partial derivatives at the points. We refer to the points a_{ij} as the nodes of the finite element.

The Lagrange finite elements (K,P_K,Σ_K) which we have considered all share the following crucial property: Let $\varphi_i \in \Sigma_K$ be of the form $\varphi_i : p \to p(a_i)$. Then the associated basis function p_i is identically zero on any side of the finite element which does not contain the node a_i. This fact has the following three important consequences:

1. Let Δ denote a face of K. Then the restriction to Δ of a function in P_K solely depends upon the degrees of freedom whose associated nodes are on Δ.
2. Any basis function in V_h constructed from the basis functions of the finite elements is

automatically continuous over $\overline{\Omega}$.

3. If we are to construct a subspace of $H_0^1(\Omega)$, then it suffices that we equate to zero the degrees of freedom whose associated nodes are boundary nodes, i.e., those which lie on Γ. In other words, if we let V_h denote the finite element subspace "without boundary condition", i.e., V_h is an approximation of the space $H^1(\Omega)$, then the space

$$V_{0h} = \left\{ v \in V_h;\ \forall a \in \Sigma_{0h} = \Sigma_h \cap \Gamma,\ v(a) = 0 \right\}$$

is an approximation of the space $H_0^1(\Omega)$, where we have identified Σ_h with the set of all nodes of T_h. As an important consequence, the V_h-interpolate of a sufficiently smooth function v vanishing on the boundary Γ is also the V_{0h}-interpolate of v.

Continuity of Basis Functions in P_k

Let us show that the functions in V_h are continuous over the set $\overline{\Omega}$. We will consider the case of triangular meshes and quadratic polynomial basis functions, but a similar argument can be used to show continuity for other cases. Since the function $v \in V_h$ is already continuous in the interior of each element, it it is sufficient to check two functions $v|_{K_1}$ and $v|_{K_2}$ across a common inter-element side S of two adjacent triangles K_1 and K_2. Let t denote the abscissa along an axis containing the segment $S = \left[b_i,\ b_k \right]$. The two functions $v|_{K_1}$ and $v|_{K_2}$ along S are quadratic polynomials of t whose values coincide at the three points b_i, b_j, b_k; therefore, they are identical and we conclude that $v \in C^0(\overline{\Omega})$.

Affine Transformations and Reference Elements

We now come to an essential idea which we will apply to an example. Consider a family of triangular elements of type 2 (quadratic). Our aim is then to describe such a family as simply as possible. Let $\hat{K}$ be a triangle with vertices $\hat{a}_i$ and midpoints of the sides $\hat{a}_{ij} = \frac{(\hat{a}_i + \hat{a}_j)}{2}$, $1 \le i < j \le 3$, and let

$$\hat{\Sigma} = \left\{ p(\hat{a}_i),\ 1 \le i \le 3,\ p(\hat{a}_{ij}),\ 1 \le i < j \le 3 \right\},$$

so that the triple $\left(\hat{K}, \hat{P}, \hat{\Sigma} \right)$ with $\hat{P} = P_2$ is also a triangular element of type 2.

Given any finite element in the family, there exists a unique invertible affine mapping

$$F_K : \hat{x} \in R^2 \to F_K(\hat{x}) = B_K \hat{x} + b_K \in R^2,$$

i.e., with B_K an invertible matrix and b_K a vector in R^2, such that

$$F_K(\hat{a}_i) = a_i \qquad 1 \le i \le 3.$$

Then it automatically follows that

$$F_K(\hat{a}_{ij}) = a_{ij} \qquad 1 \le i < j \le 3.$$

This is so because the property for a point to be the midpoint of a segment is preserved by an affine mapping. Once we have established a bijection $\hat{x} \in \hat{K} \to F_K(\hat{x}) \in K$ between the points of the sets $\hat{K}$ and K, it is natural to associate the space

$$P_K = \left\{ p : K \to R;\ p = \hat{p} \circ F_K^{-1},\ \hat{p} \in \hat{P} \right\}$$

with the space $\hat{P}$. Then it follows that

$$P_K = P_2.$$

In other words, rather than prescribe such a family by the data (K, P_K, Σ_K), it suffices to give only *one* reference finite element $\left(\hat{K}, \hat{P}, \hat{\Sigma}\right)$ and the affine mappings F_K occurring in the family. Then we have the following relations

$$K = F_K\left(\hat{K}\right)$$

$$\Sigma_K = \left\{p(F_K(\hat{a}_i)),\ 1 \le i \le 3,\ p(F_K(\hat{a}_{ij})),\ 1 \le i < j \le 3\right\}$$

$$P_K = \left\{p : K \to R;\ p = \hat{p} \circ F_K^{-1},\ \hat{p} \in \hat{P}\right\}.$$

From this example, let us now derive the following general definition. Two finite elements $\left(\hat{K}, \hat{P}, \hat{\Sigma}\right)$ and (K, P, Σ) as defined in above are affine-equivalent if there exists an invertible affine mapping

$$F : \hat{x} \in R^n \to F(\hat{x}) = B\hat{x} + b \in R^n$$

such that the following relations hold

$$K = F\left(\hat{K}\right)$$

$$a_{ij} = F(\hat{a}_{ij})$$

whenever the nodes a_{ij} and $\hat{a}_{ij}$ occur in the definition of the set Σ or $\hat{\Sigma}$, respectively. Also,

$$P = \left\{p : K \to R;\ p = \hat{p} \circ F^{-1},\ \hat{p} \in \hat{P}\right\}.$$

We shall constantly use the bijections

$$\hat{x} \in \hat{K} \to x \in K \qquad \text{where} \qquad x = F(\hat{x})$$

$$\hat{p} \in \hat{P} \to p \in P \qquad \text{where} \qquad p = \hat{p} \circ F^{-1}$$

$$p(x) = \hat{p}(\hat{x}) \qquad \text{for all} \qquad x = F(\hat{x})$$

between the points and functions associated with two affine-equivalent finite elements $\left(\hat{K}, \hat{P}, \hat{\Sigma}\right)$ and (K, P, Σ). A family of finite elements is called an affine family if all its finite elements are affine-equivalent to a single representative finite element, which is then called the reference finite element of the family.

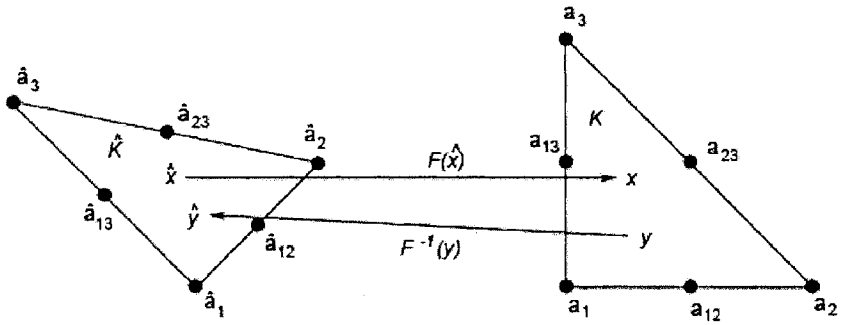


Figure 2.7. Two affine-equivalent triangles.

Let us again consider the construction of the subspace V_h. For the sake of brevity, our discussion will be essentially concerned with the two dimensional case, but all the subsequent considerations apply equally well to arbitrary dimensions. Consider a triangulation T_h of a polynomial domain $\overline{\Omega}$ in R^2, consisting of finite elements (K, P_K, Σ_K), $K \in T_h$, which are all of the same (affine-equivalent) type. For instance, consider a triangulation consisting of triangles of type 1. A space V_h is then associated with such a triangulation. Of course, if K_1 and K_2 are two adjacent finite elements, some compatibility conditions must be satisfied by the two sets Σ_{K_1} and Σ_{K_2}, if we are to define unambiguously a set Σ_h of degrees of freedom of the space V_h, which are now linear functions over the space V_h such that

$$\Sigma_h = \bigcup_{K \in T_h} \Sigma_K.$$

When the degrees of freedom are element nodal points, then the degrees of freedom of the space V_h are of the following form:

$$\varphi_{j,h} : v \to v(b_j),$$

where the points b_j are called the nodes of space V_h (to be distinguished from the vertices of the triangulation). If we write the set Σ_h as

$$\Sigma_h = \{\varphi_{i,h}\}_{i=1}^{M},$$

then the basis functions w_j, $1 \le j \le M$, of the space V_h are naturally defined by the relations

$$\varphi_{i,h}(w_j) = \delta_{ij} \qquad 1 \le i \le M.$$

The basis functions w_j of the space V_h can now be derived by "patching together" the basis functions of each finite element. And as we have seen, if the triangulation consists entirely of elements from the same affine family, only the basis functions for a single reference element are needed to accomplish this.

Three Dimensional Elements

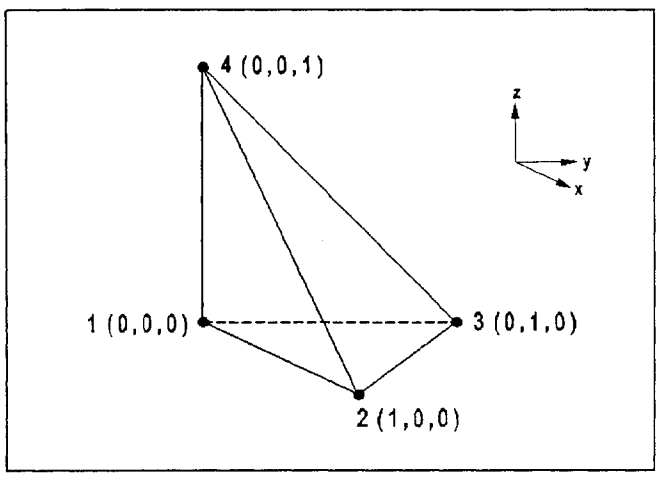
As stated previously, the concepts that have been presented for the two dimensional case can be easily extended to three dimensional space. Without restating the theory, we will now briefly examine two of the most common three dimensional shapes for finite elements, analogous to the two dimensional triangle and rectangle elements, the tetrahedron and the hexahedron respectively.

Mirroring the two dimensional case, we have that a finite element in R^3 is a triple (K, P_K, Σ_K) where the data K, P_K, and Σ_K are defined as follows:

1. K is a compact subset of R^3 with a non-empty interior and a Lipschitz-continuous boundary.
2. P_K is a finite-dimensional space of real-valued functions defined over the set K, of dimension N.
3. Σ_K is a set consisting of the degrees of freedom for the element. In our case, since we continue to work with only Lagrange elements, the degrees of freedom are function values at the nodes of the element.

Tetrahedral Elements

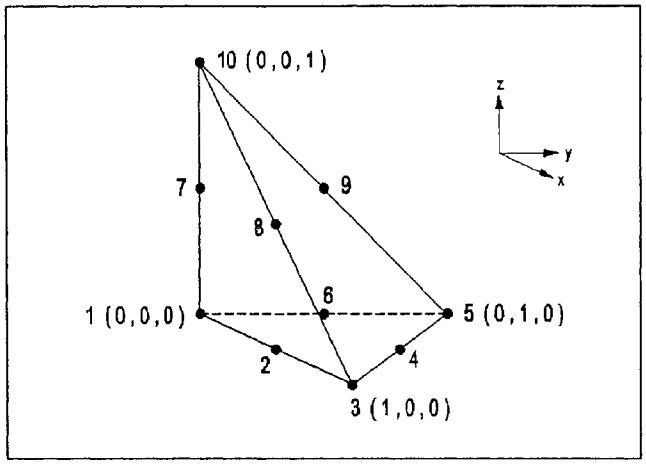
The extension of 2-simplex (triangular) elements to three dimensional space leads to 3-simplex (tetrahedral) elements. For the simplest of such elements, which we will call the 3-simplex of type 1 or 4-node linear tetrahedral element, we have that P_K is P_1, and Σ_K consists of the values at the four vertices of K, i.e., $p(a_i)$, $1 \leq i \leq 4$, (Figure 2.8).



$$\begin{cases} P_K = P_1; \ \dim(P_K) = 4 \\ \Sigma_K = \{p(a_i), \ 1 \leq i \leq 4\} \end{cases}$$

Figure 2.8.

If we also consider the nodes a_{ij} at the midpoints of edges between nodes a_i and a_j we come to the next type of tetrahedral element, the 3-simplex of type 2, or 10-node quadratic tetrahedral element. In this case P_K is P_2, and Σ_K consists of the values at the four vertices, i.e., $p(a_i)$, $1 \leq i \leq 4$, as well as the values $p(a_{ij})$, $1 \leq i < j \leq 4$ at the edge midpoints (Figure 2.9).



$$\begin{cases} P_K = P_2;\ \dim(P_K) = 10 \\ \Sigma_K = \{p(a_i), 1 \le i \le 4; \\ \qquad p(a_{ij}), 1 \le i < j \le 4\} \end{cases}$$

Figure 2.9.

Hexahedral Elements

We now turn our attention to the extension of two dimensional rectangular elements to the three dimensional hexahedral elements. For simplicity, we will only consider the case of hypercubes. For the simplest of such elements, which we will call the cube of type 1 or 8-node linear hexahedral element, we have that Q_K is Q_1, and Σ_K consists of the values at the eight vertices of K, i.e., $p(a_i)$, $1 \le i \le 8$, (Figure 2.10).

$$\begin{cases} Q_K = Q_1;\ \dim(Q_K) = 8 \\ \Sigma_K = \{p(a_i),\ 1 \le i \le 8\} \end{cases}$$

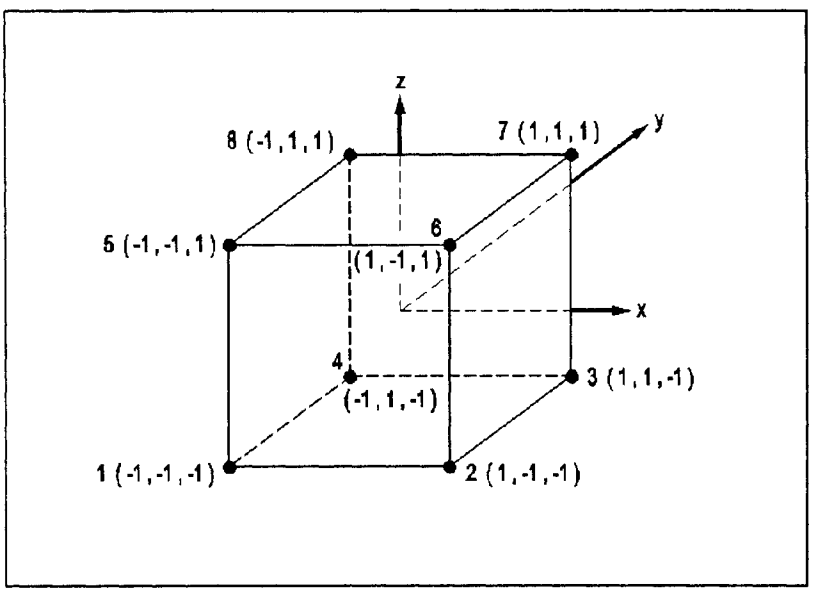


Figure 2.10.

In the case of the hypercube of type 2, or 27-node quadratic hypercube, we add not only nodes at the midpoints of edges, but also nodes at the middle of each of the six square faces of the element, as well as a node at the centre of gravity of the element (centre of the interior of the element). For simplicity we shall just number these nodes from 1 to 27, following a

counter-clockwise order starting on the base. In this case Q_K is Q_2, and Σ_K consists of the values at the twenty-seven nodes, i.e., $p(a_i), 1 \leq i \leq 27$ (Figure 2.11).

$$\begin{cases} Q_K = Q_2;\ \dim(Q_K) = 27 \\ \Sigma_K = \{p(a_i),\ 1 \leq i \leq 27\} \end{cases}$$

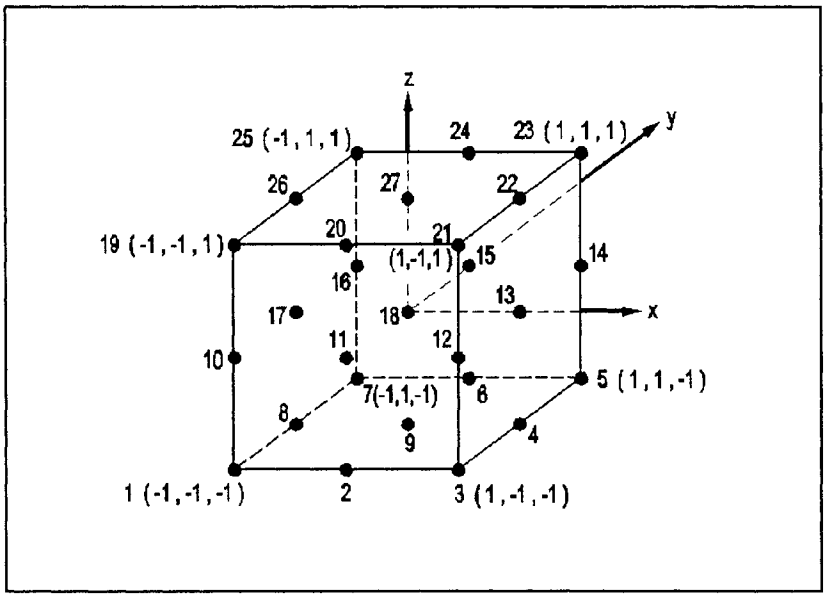


Figure 2.11.

2.4 Solving Linear Systems

For the problems we have discussed, the discretization of partial differential equations using the finite element method leads to a linear system of equations. In particular, the use of localized basis functions in the finite element approach results in a matrix for the system which is sparse, as well as positive definite. We now turn our attention to methods for solving such systems. We also note that sparsity of the system can be exploited to reduce the storage and work required for solving the systems for two and three dimensional problems to much less than the $O(n^2)$ and $O(n^3)$, respectively, that might be expected in a more naive approach. We will begin our discussion of solvers with an examination of direct methods, specifically we will look at Gaussian elimination and Cholesky factorization. Following this, we will turn our attention to iterative methods, the Gauss-Seidel and conjugate gradient methods in particular. During our discussion we will also point out special considerations for improving efficiency based on the properties of the linear systems involved. Before proceeding, we mention that other methods for solving linear systems, such as Multigrid [McCormick, 1989], [Douglas, 1997] and Fourier methods [Henrici, 1979], [Swarztrauber, 1984], are also applicable to solving the systems in question. Although these methods can achieve near optimal computational efficiency, a detailed discussion of their theory and implementation is beyond the scope of this thesis. For a further examination of methods for solving linear systems the reader is directed to [Johnson, 1987], or general texts on numerical methods or scientific computing such as [Rao, 2002], and [Heath, 2002].

Direct Methods - Sparse Factorization Methods

First we briefly consider direct methods for solving large sparse linear systems. Gaussian elimination and its variants such as Cholesky factorization for symmetric positive definite matrices are applicable in our case, but special considerations should be made in order to make the solution process as efficient as possible. In particular, since the matrix is sparse, we should take care to only store and operate on the nonzero entries of the matrix. Thus, the standard two-dimensional array, often used for dense matrices, is not suitable in our case. Instead, special data structures are needed for efficient storage of the system matrix.

For one-dimensional problems, the equations and unknowns can usually be ordered so that the nonzeros are concentrated in a relatively narrow band, which can be stored efficiently in a rectangular two-dimensional array by diagonals. Algorithms are available for reducing the bandwidth, if necessary, by reordering the rows and columns of the matrix. But for problems in higher dimensional spaces, even the narrowest possible band often contains mostly zeros, and hence any type of two-dimensional array storage would be prohibitively wasteful. In general, sparse systems require data structures in which individual nonzero entries are stored, along with the indices required to identify their locations in the matrix. Explicitly storing their indices not only incurs additional storage overhead but also makes arithmetic operations on the nonzeros less efficient due to the indirect addressing required to access the operands. Thus, such a representation is worthwhile only if the matrix is sufficiently sparse, which is often the case for very large problems arising from partial differential equations (PDEs) and many other applications.

When applied to a sparse matrix, LU or Cholesky factorization can be carried out in the usual manner, but taking linear combinations of rows and columns to annihilate nonzero entries can introduce new nonzeros in locations in the matrix that were initially zero. Such new nonzeros, called *fill*, must then be stored and, depending on their locations, may eventually be annihilated themselves in order to obtain the triangular factors. In any case, the resulting triangular factors can be expected to contain at least as many nonzeros as the original matrix and usually a significant amount of fill as well. The amount of fill incurred is very sensitive to the order in which the rows and columns of the matrix are processed. Different enumerations of the nodes may give different degrees of fill. Thus, one of the central problems in sparse factorization is to reorder the original matrix to limit the amount of fill that the matrix suffers during the factorization. Exact minimization of the fill turns out to be a very hard combinatorial problem (NP-complete), but heuristic algorithms are available that do a good job of limiting fill for many types of problems. To illustrate sparse factorization, consider the following simple example. Suppose a two-dimensional problem is discretized over a 3×3 grid consisting of four square elements (Figure 2.12). If a pair of nodes in the mesh are neighbours (are connected by a line or edge), then both appear in the same equation in the system.

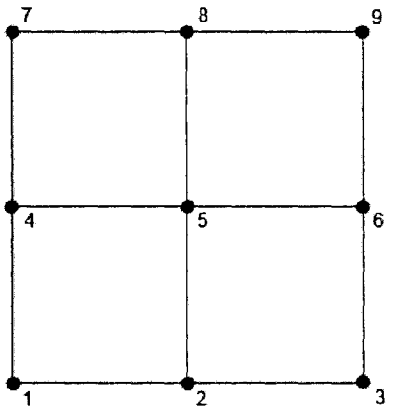


Figure 2.12. 3 × 3 grid mesh

$$A = \begin{bmatrix} \times & \times & & \times & & & & & \\ \times & \times & \times & & \times & & & & \\ & \times & \times & & & \times & & & \\ \times & & & \times & \times & & \times & & \\ & \times & & \times & \times & \times & & \times & \\ & & \times & & \times & \times & & & \times \\ & & & \times & & & \times & \times & \\ & & & & \times & & \times & \times & \times \\ & & & & & \times & & \times & \times \end{bmatrix}, \quad L = \begin{bmatrix} \times & & & & & & & & \\ \times & \times & & & & & & & \\ & \times & \times & & & & & & \\ \times & + & + & \times & & & & & \\ & \times & + & \times & \times & & & & \\ & & \times & + & \times & \times & & & \\ & & & \times & + & + & \times & & \\ & & & & \times & + & \times & \times & \\ & & & & & \times & + & \times & \times \end{bmatrix}$$

Figure 2.13. Example of nonzero patterns for sparse matrix, A, and fill patterns for its Cholesky factor L.

This connection can be seen by the nonzero pattern in the matrix A in Figure 2.13, where × represents a nonzero value. The diagonal entries of the matrix correspond to the nodes in the mesh, and the nonzero off-diagonal entries correspond to the edges in the mesh (i.e., $a_{ij} \neq 0$ iff nodes i and j are neighbours). Note that the matrix is banded, but it also has many zero entries inside the band. Specifically, the matrix is block tridiagonal, with each nonzero block being either tridiagonal or diagonal, as expected for a row- or column-wise ordering of a two-dimensional grid. Cholesky factorization of the matrix in this ordering fills in the band almost completely, as shown by matrix L in the above figure, where fill entries (new nonzeros) are indicated by +.

Gaussian Elimination and Cholesky's Method

We begin our examination of direct methods by recalling that using Gaussian elimination to solve a linear system produces a LU-factorization of A, i.e.,

$$A = LU, \qquad (2.4.1)$$

where $L = (l_{ij})$ is a lower triangular $M \times M$ matrix (i.e., $l_{ij} = 0$ if $j > i$), and $U = (u_{ij})$ is an upper triangular matrix (i.e., $u_{ij} = 0$ if $j < i$).

From the factorization (2.4.1) it is easy to solve the system $A\xi = b$ by using forward and backward substitution to solve the triangular systems:

$$L\eta = b,$$
$$U\xi = \eta.$$

Recall that for Gaussian elimination, $U = A^{(M)}$ where the matrices $A^{(k)}$, $k = 1,\ldots M$, are successively computed as follows:

(i) $A^{(1)} = A$,

(ii) Given $A^{(k)}$ of the form

$$A^{(k)} = \begin{bmatrix} a_{11}^{(k)} & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(k)} \\ & \ddots & & & & \vdots \\ & & \ddots & & & \vdots \\ & & & a_{kk}^{(k)} & & a_{kn}^{(k)} \\ & 0's & & \vdots & \ddots & \vdots \\ & & & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}$$

determine $A^{(k+1)} = \left(a_{ij}^{(k+1)}\right)$ as follows

$$a_{ij}^{(k+1)} = a_{ij}^{(k)}, \qquad \begin{matrix} i = 1,\ldots,k, \text{ or} \\ j = 1,\ldots,k-1, \end{matrix}$$

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} a_{kj}^{(k)} \qquad \begin{matrix} i = k+1,\ldots,M, \text{ and} \\ j = k,\ldots,M, \end{matrix}$$

under the assumption that $a_{kk}^{(k)} \neq 0$.

Also recall that $L = (l_{ij})$, where

$$l_{ii} = 1, \qquad i = 1,\ldots,M,$$
$$l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \qquad i = k+1,\ldots,M, \quad k = 1,\ldots,M,$$
$$l_{ik} = 0, \qquad \text{if } i < k.$$

It can be shown, [Wendroff, 1966, pp.124-125], that if A is a symmetric positive definite matrix then A has a triangular decomposition which can be obtained by Gaussian elimination

and $a_{kk}^{(k)} > 0$, $k = 1, \ldots, M$. Thus, Gaussian elimination can be performed without pivoting. In addition, it is not necessary to perform pivoting to prevent numerical instability due to small pivot elements $a_{kk}^{(k)}$. Also, we may perform the Gaussian elimination in any desired order. We will see that different direct methods for solving the linear system of equation (2.2.6) essentially differ in the choice of the order of the elimination. Alternatively, in the case where we perform the elimination according to the ordering of the nodes, these methods only differ in their choice of enumeration of the node.

Since A is symmetric positive definite we may alternatively factor A as

$$A = BB^T$$

with $B = DL$ and where D is a diagonal matrix with diagonal elements

$$d_{kk} = \sqrt{a_{kk}^{(k)}}, \qquad k = 1, \ldots, M$$

and L and $a_{kk}^{(k)}$ are obtained through Gaussian elimination as above. Here B^T denotes the transpose of the matrix B. The elements b_{ij} of matrix B can alternatively be determined using Cholesky's method as follows:

$$b_{11} = \sqrt{a_{11}},$$

$$b_{i1} = \frac{a_{i1}}{b_{11}} \qquad i = 2, \ldots, M,$$

and for $j = 2, \ldots, M$,

$$b_{jj} = a_{jj} - \sum_{k=1}^{j-1} b_{jk}^2$$

$$b_{ij} = \frac{\left(a_{ij} - \sum_{k=1}^{j-1} b_{ik} b_{jk} \right)}{b_{jj}}, \qquad i = j+1, \ldots, M.$$

Efficiency Considerations - Operation Counts and Band Matrices

The number of arithmetic operations to obtain an LU-factorization of a dense $M \times M$ matrix is of the order $\frac{M^3}{3}$. To see this, let us analyze a somewhat more straightforward version of the algorithm for Gaussian elimination:

Algorithm *LU Factorization by Gaussian Elimination*

```
(1)   for k = 1 to m − 1                (loop over columns)
(2)       if a_kk = 0 then stop         (halt algorithm if pivot is zero)
(3)       for i = k + 1 to m              (compute multipliers for
(4)           l_ik = a_ik/a_kk             current column)
(5)       end-for
(6)       for j = k + 1 to m
(7)           for i = k + 1 to m            (apply transformation to
(8)               a_ij = a_ij − l_ik a_kj     remaining submatrix)
(9)           end-for
(10)      end-for
(11)  end-for
```

Since the time required for a computer to carry out floating-point multiplications and divisions is much greater than that needed for additions and subtractions we will confine ourselves to counting only the number of multiplicative operations. Firstly, let us analyze the division that occurs at line (4). The first time we go through the for-loop (at line (3)) k will equal 1, so line (4) will be executed $m-1$ times. The next time through the algorithm k will be 2, so line (4) will be executed $m-2$ times, and so on until the last time when it is only executed once. Thus, the total number of divisions is:

$$(m-1)+(m-2)+\ldots+2+1=\sum_{i=1}^{m-1} i=\frac{(m-1)m}{2}.$$

Now, let us consider the number of times the multiplication on line (8) is done. Again we note that the first time we get to the for-loop at line (6) k will equal 1, so this loop will execute $m-1$ times. In addition, the loop at line (7) will also be executed $m-1$ times the first time it is encountered. However, since the loops are nested, the division at line (8) will in fact be executed a total of $(m-1)(m-1)$ times. Following the same argument as above, we see that the total number of multiplications is:

$$(m-1)^2+(m-2)^2+\ldots+2^2+1^2=\sum_{i=1}^{m-1} i^2=\frac{m(m-1)(2(m-1)+1)}{6}.$$

Finally, adding the two equations we have

$$\frac{(m-1)m}{2}+\frac{m(m-1)(2(m-1)+1)}{6}=\frac{m(m-1)(m+1)}{3}=\frac{O(m^3)}{3},$$

giving the expected order of operations.

If the matrix is sparse, then it is possible to greatly reduce the number of operations by taking advantage of the sparsity. This is particularly easy to do if the matrix A is a band matrix (Figure 2.14), i.e., there is a number d, the band width, such that

$$a_{ij}=0 \text{ if } |i-j|>d.$$

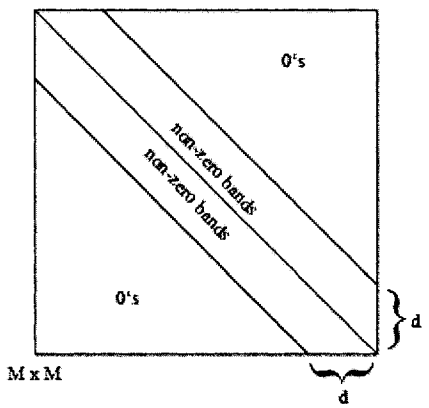


Figure 2.14. Sparse matrix with bandwidth d.

To factor an $M \times M$ matrix with band width d one only needs of order Md^2 operations.

When $a_{ij} = a(\varphi_i, \varphi_j)$, where $a(.,.)$ is a bilinear form and $\{\varphi_1, \ldots, \varphi_M\}$ is a basis for a finite element space V_h, we have that

$d = \max\{|i-j|$: φ_i and φ_j are associated with degrees of freedom

belonging to the same element$\}$.

Thus, we see that the bandwidth depends on the chosen enumeration of the nodes in the finite element mesh. In order to make the Gaussian elimination as efficient as possible the nodes should be enumerated in such a way as to minimize the bandwidth of the system matrix.

For example, consider the node enumeration for the mesh in Figure 2.15:

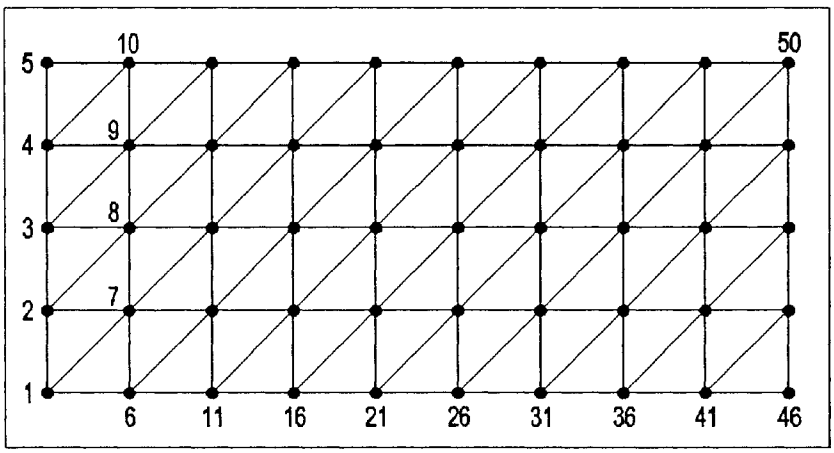


Figure 2.15

With nodes enumerated in this manner we have a bandwidth of $d = 6$ (which is the minimal bandwidth for this case, assuming only one degree of freedom per node). If, on the other hand, we were to enumerate the nodes in a horizontal manner, we would have a bandwidth of $d = 11$. A number of methods, such as the Frontal method [Duff, 1996], [Irons, 1970] and the Nested Dissection method [George, 1973], are available which attempt to enumerate the nodes and perform the elimination process as efficiently as possible.

Efficiency Considerations - Sparse Matrix Storage

Also note that if the matrix A is a symmetric matrix then only the upper or lower triangular portion of A needs to be stored. Thus, if $A = (a_{ij})$ is a symmetric band matrix with a bandwidth of, for example, 2, then one method of storage is to store A as a vector $a = (a_i)$ with the entries of the columns in the band stored in consecutive order. For example, consider the following case where the entries of A have been renumbered according to the sequence in which they would be stored as a vector.

$$A = \begin{bmatrix} a_1 & a_2 & a_4 & 0 & 0 & 0 \\ & a_3 & a_5 & a_7 & 0 & 0 \\ & & a_6 & a_8 & \ddots & 0 \\ & & & a_9 & \ddots & a_{n-2} \\ & sym & & & \ddots & a_{n-1} \\ & & & & & a_n \end{bmatrix}.$$

Of course, in order to properly reconstruct the matrix A, we would also need to store information regarding where the columns of A begin and end. One way to accomplish this is to store a second vector $c = (c_i)$ which consists of indices indicating the entries of a which correspond to the first entries of a column in A. For the above example we would have:

$$c = \begin{bmatrix} 1 & 2 & 4 & 7 & \ldots & n-2 \end{bmatrix}.$$

We would also need to store some additional information, such as the bandwidth number, in order to properly reconstruct the columns of A. Also note that this method would not be suitable if there were a large number of zero entries within the band. If this were the case, then we would store only the nonzero entries in a, and we would require a third vector $r = (r_i)$ to store the row indices of the nonzero elements. This more general case corresponds to the Harwell-Boeing format for sparse matrix storage [Duff, Grimes, and Lewis, 1992].

Iterative Methods For Linear Systems

The direct methods considered above compute the exact solution of the problem, subject only to the effects of rounding error, in a finite number of steps. This seems a desirable property, but the price paid in work and storage can be prohibitive for very large linear systems. By contrast, iterative methods for solving linear systems begin with an initial estimate for the solution, $x^{(0)}$, and successively improve on it until the solution is as accurate as desired. (Note that we will use parenthesized superscripts for the iteration index.) In theory, an infinite number of iterations might be required to converge to the exact solution, but in practice the iteration terminates when some measure of the error, typically some norm of the residual, is as small as desired. Providing they converge rapidly enough, iterative methods have several significant advantages over direct methods.

The simplest type of iterative method for solving a linear system $Ax = b$ has the form

$$x^{(k+1)} = Gx^{(k)} + c$$

where the matrix G and vector c are chosen so that a fixed point of the equation $x = Gx + c$ is a solution of $Ax = b$. Such a method is said to be stationary if G and c are constant over all iterations.

One way to obtain a suitable matrix G is by splitting, in which the matrix A is written as

$$A = M - N$$

with M nonsingular. We can then take $G = M^{-1}N$ and $c = M^{-1}b$, so that the iteration scheme becomes

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b$$

or equivalently,

$$Mx^{(k+1)} = Nx^{(k)} + b$$

so that we solve a linear system with matrix M at each iteration. Formally this splitting scheme is a fixed-point iteration with iteration function

$$g(x) = M^{-1}Nx + M^{-1}b$$

whose Jacobian matrix is

$$G(x) = M^{-1}N.$$

Thus, the iteration scheme is convergent if the spectral radius

$$\rho(G) = \rho(M^{-1}N) < 1,$$

and the smaller $\rho(G)$, the faster the convergence.

For rapid convergence, we should choose M and N so that $\rho(M^{-1}N)$ is as small as possible. There is a trade-off, however, as the cost per iteration is determined by the cost of solving a linear system with matrix M. As an extreme example, if $M = A$, then the scheme converges in a single iteration, but that one iteration may be prohibitively expensive. In particular, M is chosen to approximate A in some sense, but is usually constrained to have some simple form, such as diagonal or triangular, so that the linear system at each iteration is easy to solve.

Gauss-Seidel Method

The Gauss-Seidel method has an advantage over some other iterative methods, e.g., Jacobi method, in that, as each new component of the solution $x_i^{(k+1)}$ is computed for the Gauss-Seidel method, it is immediately used in the next equation to determine additional $x^{(k+1)}$ values. Thus, if the solution is converging, the best estimate will always be employed. The process starts by choosing an initial guess for x^0. Generally, the initial guess will simply be the zero vector, unless some approximation of x is known a priori. With the initial vector chosen, subsequent x's are computed by:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}}{a_{ii}} \qquad i = 1,\ldots,n.$$

or, using matrix notation for the system

$$Ax = b$$

we have:

$$\begin{aligned} x^{(k+1)} &= D^{-1}(b - Lx^{(k+1)} - Ux^{(k)}) \\ &= (D+L)^{-1}(b - Ux^{(k)}) \end{aligned}$$

where,

D is the matrix consisting of the diagonal entries of A.

L is the matrix consisting strictly of the lower triangular portion of A.

U is the matrix consisting strictly of the upper triangular portion of A.

This process for calculating $x^{(k)}$ is repeated until the solution converges to a prespecified

tolerance percentage ϵ_s. i.e., the process repeats until

$$|\varepsilon_{a,i}| = \left| \frac{x_i^{(j)} - x_i^{(j-1)}}{x_i^{(j)}} \right| 100\% < \varepsilon_s$$

for all i, where j and $j-1$ are the present and previous iterations respectively.

In addition to a fast rate of convergence, the Gauss-Seidel method has the added benefit that duplicate storage is not needed for the vector x, since the newly computed component values can overwrite the old ones immediately.

Convergence Criterion for Gauss-Seidel Method

Note that the Gauss-Seidel Method can sometimes suffer the problem of nonconvergence, or sometimes it will only converge very slowly. A sufficient but not necessary condition for convergence is that the system be diagonally dominant, i.e.,

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|$$

That is, the diagonal element must be greater than the off-diagonal elements for each row.

Conjugate Gradient Method

We now turn from stationary iterative methods to methods based on optimization. If A is an $n \times n$ symmetric positive definite matrix, then the quadratic function (where we use v^T to denote the transpose of vector v)

$$\phi(x) = \tfrac{1}{2} x^T A x - x^T b$$

attains a minimum precisely when $Ax = b$. Thus, we can apply optimization methods to obtain a solution of the corresponding linear system. In general, optimization methods progress from one iteration to the next by performing a one-dimensional search along some search direction $s^{(k)}$, so that

$$x_{k+1} = x_k + \alpha s_k$$

where α is a linear search parameter that is chosen to minimize the objective function $\phi(x_k + \alpha s_k)$ along s_k.

We note some special features of such a quadratic optimization problem. First, the negative gradient is simply the residual vector:

$$-\nabla\phi(x) = b - Ax = r.$$

Second, for any search direction s_k, we need not perform a line search, because the optimal choice for α can be determined analytically. Specifically, the minimum over α occurs when the new residual is orthogonal to the search direction:

$$0 = \tfrac{d}{d\alpha}\phi(x_{k+1}) = \nabla\phi x_{k+1}^T \tfrac{d}{d\alpha} x_{k+1} = (Ax_{k+1} - b)^T \tfrac{d}{d\alpha}(x_k + \alpha s_k) = -r_{k+1}^T s_k.$$

Since the new residual can be expressed in terms of the old residual and the search direction,

$$r_{k+1} = b - Ax_{k+1} = b - A(x_k + \alpha s_k) = (b - Ax_k) - \alpha A s_k = r_k - \alpha A s_k,$$

we can solve for

$$\alpha = \frac{r_k^T s_k}{s_k^T A s_k}.$$

Taking these properties into account, we obtain the conjugate gradient method, or CG method, for solving symmetric positive definite linear systems:

Algorithm *Conjugate Gradient Method*

$x_0 = initial\ guess$

$r_0 = b - Ax_0$

$s_0 = r_0$

for $k = 0, 1, 2, \ldots$

$\alpha_k = \frac{r_k^T r_k}{s_k^T A s_k}$ *{compute search parameter}*

$x_{k+1} = x_k + \alpha_k s_k$ *{update solution}*

$r_{k+1} = r_k - \alpha_k A s_k$ *{compute new residual}*

$\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$

$s_{k+1} = r_{k+1} + \beta_{k+1} s_k$ *{compute new search direction}*

end-for

It turns out that in the quadratic case, the error at each step of CG is minimal (with respect to the norm induced by A) over the space spanned by the search directions generated so far. Since the search directions are A-orthogonal (vectors y and z are A-orthogonal if $y^T Az = 0$), or conjugate, and hence linearly independent, this property implies that after, at most, n steps, the solution is exact, because the n search directions must span the whole space. Thus, in theory, CG is a direct method, but in practice rounding, error causes a loss of orthogonality, thus spoiling its finite termination property. As a result, CG is usually used in an iterative manner and halted when the residual, or some other measure of the error, is sufficiently small. In practice, the method often converges in far fewer than n iterations.

Preconditioning

Unfortunately, the conjugate gradient method can still converge very slowly if the matrix A is ill-conditioned. However, convergence can often be substantially accelerated by preconditioning, which can be thought of as implicitly multiplying A by M^{-1}, where M is a matrix for which systems of the form $Mz = y$ are easily solved, and whose inverse approximates A, so that $M^{-1}A$ is relatively well-conditioned. Technically, to preserve symmetry, we should apply CG to $L^{-1}AL^{-T}$ instead of $M^{-1}A$, where $M = LL^T$. However, the algorithm can be suitably rearranged so that only M is used and the corresponding matrix L is not required explicitly. The resulting preconditioned conjugate gradient method is given in the following algorithm.

Algorithm *Conjugate Gradient Method with Preconditioning*

$x_0 = initial\ guess$

$r_0 = b - Ax_0$

$s_0 = M^{-1}r_0$

for $k = 0, 1, 2, \ldots$

$$\alpha_k = \frac{r_k^T M^{-1} r_k}{s_k^T A s_k} \qquad \{compute\ search\ parameter\}$$

$$x_{k+1} = x_k + \alpha_k s_k \qquad \{update\ solution\}$$

$$r_{k+1} = r_k - \alpha_k A s_k \qquad \{compute\ new\ residual\}$$

$$\beta_{k+1} = \frac{r_{k+1}^T M^{-1} r_{k+1}}{r_k^T M^{-1} r_k}$$

$$s_{k+1} = M^{-1} r_{k+1} + \beta_{k+1} s_k \qquad \{compute\ new\ search\ direction\}$$

end-for

The choice of an appropriate preconditioner depends on the usual trade-off between the gain in the convergence rate and the increased cost per iteration that results from applying the preconditioner. A wide variety of preconditioners have been proposed, and this topic remains an active area of research. Some of the most common preconditioners are:

- Diagonal (also known as Jacobi): M is taken to be a diagonal matrix with diagonal entries equal to those of A.
- Block Diagonal (or block Jacobi): If the indices $1, \ldots, n$ are partitioned into mutually disjoint subsets, then $m_{ij} = a_{ij}$ if i and j are in the same subset, and $m_{ij} = 0$ otherwise. Natural choices include partitioning along lines or planes in two- or three-dimensional grids, respectively, or grouping together physical variables that correspond to a common node, as in many finite element problems.
- Symmetric Successive Over-Relaxation (SSOR): Using a matrix splitting of the form $A = L + D + L^T$, we can take $M = (D + L)D^{-1}(D + L)^T$, or, introducing the SSOR relaxation parameter ω,

  $$M(\omega) = \tfrac{1}{2-\omega}(\tfrac{1}{\omega}D + L)(\tfrac{1}{\omega}D)^{-1}(\tfrac{1}{\omega}D + L)^T.$$

  With the optimal choice of ω, the SSOR preconditioner is capable of reducing the condition number to $cond(M^{-1}A) = O\left(\sqrt{cond(A)}\right)$, but determining this optimal value may be impractical.
- Incomplete factorization: Ideally, one would like to solve the linear system directly using the Cholesky factorization $A = LL^T$, but as noted earlier this may incur unacceptable fill. One may instead compute an approximate factorization $A \approx \hat{L}\hat{L}^T$ that allows little or no fill (e.g., restricting the nonzero entries of $\hat{L}$ to be in the same positions as those of the lower triangle of A), and then use $M = \hat{L}\hat{L}^T$ as a preconditioner.
- Polynomial: In this case M^{-1} is taken to be a polynomial in A that approximates A^{-1}. One way to obtain a suitable polynomial is to use a fixed number of steps of a stationary iterative method to solve the preconditioning system $Mz_k = r_k$ at each conjugate gradient iteration.

- Approximate inverse: M^{-1} is determined by using an optimization algorithm to minimize the residual

$$\| I - AM^{-1} \| \text{ or } \| I - M^{-1}A \|$$

in some norm, with M^{-1} restricted to have a prescribed pattern of nonzero entries.

A significant amount of work is required to compute some of these preconditioners, and this work must also be included in the cost trade-off mentioned earlier. The conjugate gradient method is rarely used without some form of preconditioning. Since diagonal preconditioning requires almost no extra work or storage, at least this much preconditioning is generally advisable, and more sophisticated preconditioners are often worthwhile.

The conjugate gradient method is generally applicable only to symmetric positive definite systems. If the matrix A is indefinite or nonsymmetric, then the algorithm may break down both theoretically (i.e., the corresponding optimization problem may not have a minimum) and practically (i.e., the formula for α may fail). We also note that first order hyperbolic problems typically lead to non-symmetric linear systems of equations. In this case there is no associated minimization problem (unless a least-squares formulation is used) and it is not clear how to construct efficient iterative methods for general classes of non-symmetric problems. Thus, for such problems Gaussian elimination (with pivoting) is often used.

3. Pyramidal Elements

As we have seen, three dimensional problems can be discretized over a mesh constructed from hexahedral or tetrahedral elements. Both of these element types have a number of advantages and disadvantages, particularly when it comes to adaptive mesh generation.

The basic idea behind a mesh adaptation scheme is to carry out a process whereby elements at crucial regions are subdivided into smaller elements to create a finer mesh in these regions and improve the accuracy of the overall model. However, if we confine ourselves to only one element type, then the subdivision process can lead to problems. For instance, repeated anisotropic subdivision of tetrahedral elements can cause serious grid deficiency. This loss of mesh quality can lead to inaccurate solutions when directional flowfield features are present. In fact, it has been shown that in order to maintain mesh quality for arbitrary refinement levels, isotropic subdivision is required for tetrahedral meshes [Biswas, Strawn, 1996]. Hexahedral meshes, on the other hand, do not suffer from this problem as a hexahedron can be subdivided anisotropically in any of the three directions and yield child elements whose face angles are similar to their parent. Another potential drawback to tetrahedral meshes is their storage requirements. In general, tetrahedral meshes require more than twice the amount of storage as hexahedral meshes due to the greater number of edges involved [Aftosmis, Gaitonde, and Travares, 1994]. Despite having more edges in the mesh, tetrahedral meshes do not appear to give more accurate solutions than their hexahedral counterparts. In fact, under certain circumstances, such as displacement and stress calculations in elastic and elastic plastic analysis, hexahedra perform substantially better [Benzley, et al., 1995]. Although hexahedral meshes may appear to be superior, they too have their shortcomings. Hexahedral adaptation schemes tend to generate "hanging" vertices when a hexahedron cannot be split into smaller hexahedra without continuously propagating the mesh refinement into regions where it is not desired.

In order to overcome some of these deficiencies, we can construct a mesh using both hexahedral and tetrahedral elements. Combining the element types allows us to maximize the advantages of both while minimizing their disadvantages at the same time. For example, hexahedra can be used to fill in geometrically simple regions of the domain where no sharp corners or curves exist. Hexahedra may also be better suited for critical regions where more accurate results are sought, such as boundary layers or regions of high stress. Tetrahedral elements can then be used to fill in remaining, more geometrically complex, regions that are not suitable for hexahedral refinement. Unfortunately, combining tetrahedral and hexahedral elements in a mesh can lead to another problem, as illustrated by the following example.

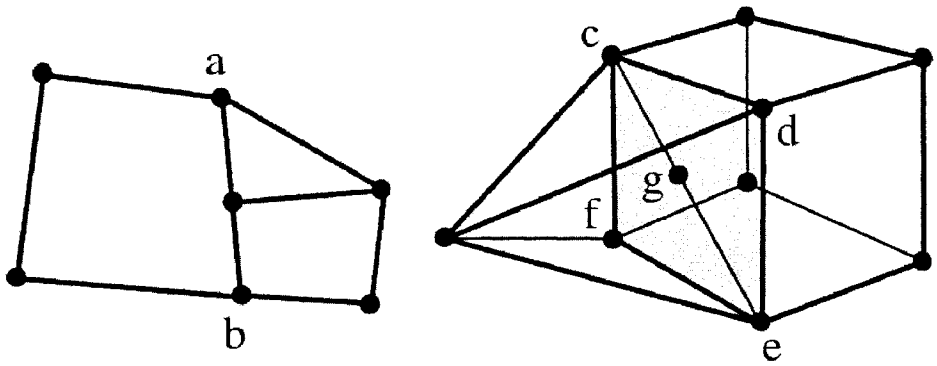


Figure 3.1. Two- and three-dimensional non-conforming meshes
[source: Owen, and Saigal, 2001]

> Geometrically, since two tetrahedra faces are required to interface with a single hexahedron, discontinuities will arise at the boundary between the two element types. Traditional uses of the finite element method require that elements conform. In two-dimensions, this principal implies that no single element edge will have more than two elements adjacent. In three-dimensions, no single face will have more than two adjacent elements. (In general, the two-dimensional case is easily avoidable, but here it serves to better illustrate the point.) Edge $a-b$ and face $c-d-e-f$ in (Figure 3.1) have more than the maximum two adjacent elements, thus rendering the mesh mathematically deficient for finite element analysis. [Owen, and Saigal, 2001]

Although techniques for interfacing hexahedra and tetrahedra directly have been proposed and implemented [Bretl, 1984], a more promising solution is the formation of pyramid and prism elements at the interface between hexahedra and tetrahedra. Indeed, the use of such elements can eliminate the need for tetrahedra altogether, as they can be used to fill in regions of a hexahedral mesh without unnecessarily propagating the grid refinement [Biswas, Strawn, 1998]. In other words, pyramids and prisms can be used to "connect up" the hanging vertices that might be created during a hexahedral mesh refinement. When used as interface elements between hexahedral and tetrahedral portions of a mesh, pyramidal and prismatic elements are often referred to as "mortar" elements, since they act like a glue joining the other element types together.

Although pyramids are ideal shapes for interfacing between tetrahedra and hexahedra, the development of basis functions for pyramidal elements has proven to be problematic. In particular, it can be shown [Wieners, 1997] that no polynomial shape functions exist for pyramids. The solution presented in [Wieners, 1997] is to split the pyramid in half and develop piecewise polynomial basis functions on the composite element. Unfortunately, this composition introduces an artificial anisotropy in solving isotropic problems. Another method for constructing pyramidal elements is to form a degenerate hexahedron where the nodes on one face are collapsed down to a single point [Owen, and Saigal, 2001], [Gradinaru and Hiptmair, 1999]. Although this form of construction is commonly used in commercial finite element codes, it is hardly an ideal solution to the problem. In addition, the quadratic pyramidal elements studied to date are 13-node quadratic elements which lack a node at the centre point of the base, and are not suited for face-to-face connections with 27-node hexahedra.

These issues will be addressed in the upcoming sections of this work. In sections 3.1 and 3.2 we present work already published [Liu, Davies, Yuan, and Křížek, 2004] which addresses the problem of artificial anisotropy in the composite pyramidal elements introduced by [Wieners, 1997]. In particular, section 3.1 will deal with the problem of artificial anisotropy by developing a new linear pyramidal element which possesses greater symmetry than the previous elements. In section 3.2 these ideas are extended in order to develop a more symmetric 13-node quadratic pyramidal element. The development of a new, highly symmetric,14-node quadratic pyramidal element will be the focus of section 3.3.

3.1 The Five Node Pyramidal Element

As we have seen above, pyramidal elements can be used for a face-to-face connection of tetrahedral finite elements with hexahedral elements (Figure 3.2). These elements provide us with a very useful tool for joining three dimensional tetrahedral meshes with hexahedral meshes.

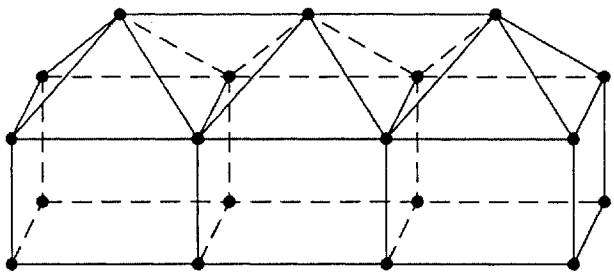


Figure 3.2.
Pyramidal elements as interface elements between hexahedra and tetrahedra.

This has many practical applications in conforming finite element discretizations of domains where only part of the domain can be decomposed into blocks by hexahedra (Figure 3.3).

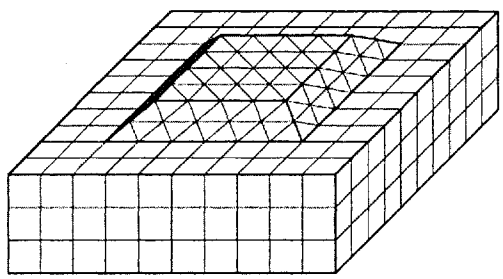


Figure 3.3.
Example domain decomposed by tetrahedra, hexahedra and pyramids.

In [Wieners, 1997], Christian Wieners presents a special family of mortar pyramidal finite elements (see also [Zgainski, et al., 1996]). Wieners proves that it is impossible to define

polynomial finite element shape functions, which would attain five given values at five vertices and which would be linear on all triangular faces and bilinear on the base of the pyramidal element. However, he shows that it is possible to define piecewise polynomial basis functions on the composite pyramidal element (see Figure 3.4) having the above-mentioned properties. According to [Křížek, Liu, and Neittaanmäki, 2001], these functions are piecewise harmonic. This gives us some advantages in practical computation [Hlaváček, Křížek, 2001]. Wieners' pyramidal elements, composed of two tetrahedra, cause an artificial anisotropy in solving isotropic problems (compare with [Křížek, and Neittaanmäki, pp. 38]). The aim of the remainder of this section, and of section 3.2, is to derive new pyramidal elements, which are composed of four tetrahedra and which have more symmetries and the same number of degrees of freedom as elements from [Wieners, 1997]. The degrees of freedom for the first type of pyramidal element are function values at vertices, and for the second type, are values at vertices and midpoints of edges.

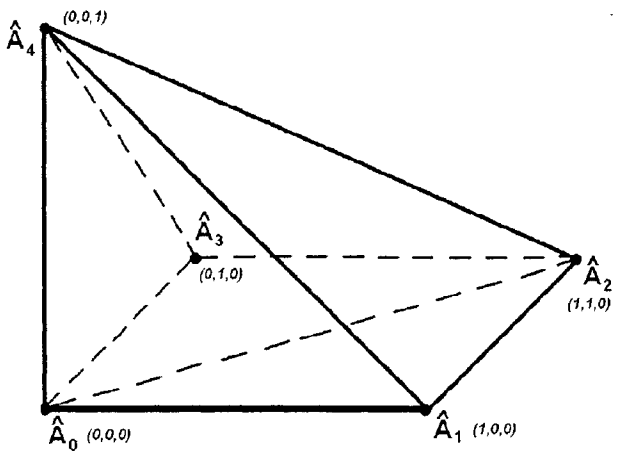


Figure 3.4. Reference element $\hat{K}$.

5-node Pyramidal Finite Element Basis Functions

Let *conv* stand for the convex hull. Then Wieners' trilinear finite element basis functions are defined on the reference pyramid,

$$\hat{K} = conv\left\{\hat{A}_0, \hat{A}_1, \hat{A}_2, \hat{A}_3, \hat{A}_4\right\} = conv\{(0,0,0),(1,0,0),(1,1,0),(0,1,0),(0,0,1),\},$$

as follows:

$$\hat{p}_0(\hat{x},\hat{y},\hat{z}) = \begin{cases} (1-\hat{x})(1-\hat{y}) + \hat{z}(\hat{y}-1) & \text{for } \hat{x} > \hat{y} \\ (1-\hat{x})(1-\hat{y}) + \hat{z}(\hat{x}-1) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{p}_1(\hat{x},\hat{y},\hat{z}) = \begin{cases} \hat{x}(1-\hat{y}) - \hat{z}\hat{y} & \text{for } \hat{x} > \hat{y} \\ \hat{x}(1-\hat{y}) - \hat{z}\hat{x} & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{p}_2(\hat{x},\hat{y},\hat{z}) = \begin{cases} \hat{x}\hat{y} + \hat{z}\hat{y} & \text{for } \hat{x} > \hat{y} \\ \hat{x}\hat{y} + \hat{z}\hat{x} & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{p}_3(\hat{x},\hat{y},\hat{z}) = \begin{cases} (1-\hat{x})\hat{y} - \hat{z}\hat{y} & \text{for } \hat{x} > \hat{y} \\ (1-\hat{x})\hat{y} - \hat{z}\hat{x} & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{p}_4(\hat{x},\hat{y},\hat{z}) = \hat{z}.$$

Observe that $\hat{p}_i(\hat{A}_j) = \delta_{ij}$ for $i, j = 0,\ldots,4$.

In order to derive the new pyramidal finite element basis functions, we make use of the following mapping:

$$F_K\left(\hat{X}\right) : (\hat{x},\hat{y},\hat{z}) \to (x,y,z)$$

where

$$F_K\left(\hat{X}\right) = B\hat{X} + b.$$

This represents a linear affine mapping from the reference pyramid $\hat{K}$ into the new reference pyramid (see Figure 3.5 and compare with any pyramid of Figure 3.2):

$$\begin{aligned} K &= conv\{A_0, A_1, A_2, A_3, A_4\} \\ &= conv\{(-1,-1,0),(1,-1,0),(1,1,0),(-1,1,0),(0,0,1)\}. \end{aligned}$$

We easily find that

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = F_K\left(\hat{X}\right) = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} \qquad (3.1.1)$$

i.e.,

$$\begin{aligned} x &= 2\hat{x} + \hat{z} - 1 \\ y &= 2\hat{y} + \hat{z} - 1 \\ z &= \hat{z}. \end{aligned}$$

It can be directly checked that $F_K\left(\hat{A}_j\right) = A_j$ for $j = 1,2,3,4$. From (3.1.1) we see

$$\begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{bmatrix} = B^{-1}\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} - \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} \frac{1}{2} & 0 & -\frac{1}{2} \\ 0 & \frac{1}{2} & -\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x+1 \\ y+1 \\ z \end{bmatrix}$$

i.e.,

$$\begin{aligned} \hat{x} &= \tfrac{1}{2}(x - z + 1) \\ \hat{y} &= \tfrac{1}{2}(y - z + 1) \\ \hat{z} &= z. \end{aligned}$$

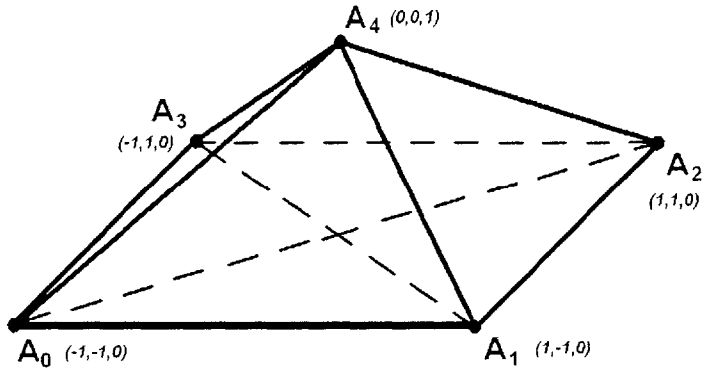


Figure 3.5. Reference element K.

From the linear affine transformation we find that the basis functions on the new reference pyramid K are of the form:

$$p_0(x,y,z) = \begin{cases} \frac{1}{4}(x-z-1)(y-z-1)+\frac{1}{2}z(y-z-1) & \text{for } x > y \\ \frac{1}{4}(x-z-1)(y-z-1)+\frac{1}{2}z(x-z-1) & \text{for } x \le y \end{cases}$$

$$p_1(x,y,z) = \begin{cases} \frac{1}{4}(x-z+1)(-y+z+1)-\frac{1}{2}z(y-z+1) & \text{for } x > y \\ \frac{1}{4}(x-z+1)(-y+z+1)-\frac{1}{2}z(x-z+1) & \text{for } x \le y \end{cases}$$

$$p_2(x,y,z) = \begin{cases} \frac{1}{4}(x-z+1)(y-z+1)+\frac{1}{2}z(y-z+1) & \text{for } x > y \\ \frac{1}{4}(x-z+1)(y-z+1)+\frac{1}{2}z(x-z+1) & \text{for } x \le y \end{cases} \qquad (3.1.2)$$

$$p_3(x,y,z) = \begin{cases} \frac{1}{4}(-x+z+1)(y-z+1)-\frac{1}{2}z(y-z+1) & \text{for } x > y \\ \frac{1}{4}(-x+z+1)(y-z+1)-\frac{1}{2}z(x-z+1) & \text{for } x \le y \end{cases}$$

$$p_4(x,y,z) = z.$$

Lemma *3.1.1*

The basis functions $p_0,\ldots,p_4$ satisfy the following conditions:

1. $p_i(A_j) = \delta_{ij}$, $i, j = 0,1,2,3,4$.
2. Each basis function is bilinear on the base of K (contained in the plane $z = 0$).
3. Each basis function is linear on all triangular faces of K.
4. Each basis function is continuous on the interelement boundary of K (contained in the plane $x = y$).
5. The sum of the basis functions is unity at any point in the pyramid (i.e., $\sum p_i(x,y,z) \equiv 1$, $i = 0,\ldots,4$).
6. The basis functions p_i, $i = 0,\ldots,4$ vanish on all faces not containing node i.

Proof :

1. This can be easily verified by direct calculation.
2. Setting $z = 0$, we immediately see that p_i, $i = 0,\ldots,4$ are bilinear, for instance, $p_0(x,y,0) = \frac{1}{4}(x-1)(y-1)$.

3. On the face $A_0A_1A_4$, which is contained in the plane $y = z - 1$, we have

$$\begin{aligned} p_0(x,y)|_{\{x>y\}} &= -\tfrac{1}{2}(x+z-1) \\ p_1(x,y)|_{\{x>y\}} &= \tfrac{1}{2}(x-z+1) \\ p_2(x,y)|_{\{x>y\}} &= p_3(x,y)|_{\{x>y\}} = 0 \\ p_4(x,y)|_{\{x>y\}} &= z. \end{aligned}$$

On the faces $A_1A_2A_4$, $A_2A_3A_4$, and $A_0A_3A_4$ this can be done similarly.

4. By setting $x = y$, we can easily see that the functions p_i, $i = 0,\ldots,4$ are continuous in the $x = y$ plane.

5. This can be verified by summing the basis functions. For instance, in the case of $x > y$ we have:

$$\begin{aligned} \sum_{i=0}^{4} p_i(x,y)|_{\{x>y\}} &= \tfrac{1}{4}(x-z-1)(y-z-1) + \tfrac{1}{2}z(y-z-1) \\ &\quad + \tfrac{1}{4}(x-z+1)(-y+z+1) - \tfrac{1}{2}z(y-z+1) \\ &\quad + \tfrac{1}{4}(x-z+1)(y-z+1) + \tfrac{1}{2}z(y-z+1) \\ &\quad + \tfrac{1}{4}(-x+z+1)(y-z+1) - \tfrac{1}{2}z(y-z+1) \\ &\quad + z \\ &\equiv 1. \end{aligned}$$

Similarly, we find that $\sum p_i(x,y)|_{\{x\le y\}} \equiv 1$, $i = 0,\ldots,4$.

6. This can be verified by a quick calculation. For instance the two faces that do not contain node A_0 are the face $A_1A_2A_4$ contained in the plane $z = 1 - x$, and the face $A_2A_3A_4$ contained in the plane $z = 1 - y$. In the case of $A_1A_2A_4$ we have:

$$p_0(x,y)|_{\{x>y\}} = \tfrac{1}{4}(x-(1-x)-1)(y-(1-x)-1) + \tfrac{1}{2}(1-x)(y-(1-x)-1) = 0.$$

Similarly, on the face $A_2A_3A_4$:

$$p_0(x,y)|_{\{x\le y\}} = \tfrac{1}{4}(x-(1-y)-1)(y-(1-y)-1) + \tfrac{1}{2}(1-y)(x-(1-y)-1) = 0.$$

Similar calculations for the remaining basis functions and appropriate faces yield the same result. □

To get more symmetries, we now take the average of the basis functions p_i, $i = 0,1,2,3$, with their mirror images. Namely, we consider the mirror image mapping

$$M : (x,y,z) \to (-x,y,z)$$

and define

$$\begin{aligned} \overline{p}_0(x,y,z) &= p_1(-x,y,z) \\ \overline{p}_1(x,y,z) &= p_0(-x,y,z) \\ \overline{p}_2(x,y,z) &= p_3(-x,y,z) \\ \overline{p}_3(x,y,z) &= p_2(-x,y,z). \end{aligned} \qquad (3.1.3)$$

Setting

$$\begin{aligned} p_i^{sym} &= \tfrac{1}{2}(p_i + \overline{p}_i) \qquad \text{for } i = 0,1,2,3 \\ p_4^{sym} &= p_4 \end{aligned} \qquad (3.1.4)$$

we find that

$$p_0^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x-1)(y-z-1) - \frac{1}{2}z & \text{for } |x| \geq y \\ \frac{1}{4}(x-1)(y-z-1) + \frac{1}{4}z(x+y-2) & \text{for } x \geq |y| \\ \frac{1}{4}(x-1)(y+z-1) & \text{for } |x| \leq y \\ \frac{1}{4}(x-1)(y-z-1) + \frac{1}{4}z(x-y-2) & \text{for } x \leq |y| \end{cases}$$

$$p_1^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x+1)(-y+z+1) - \frac{1}{2}z & \text{for } |x| \geq y \\ \frac{1}{4}(x+1)(-y+z+1) - \frac{1}{4}z(x+y+2) & \text{for } x \geq |y| \\ \frac{1}{4}(x+1)(-y-z+1) & \text{for } |x| \leq y \\ \frac{1}{4}(x+1)(-y+z+1) - \frac{1}{4}z(x-y+2) & \text{for } x \leq |y| \end{cases}$$

$$p_2^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x+1)(y-z+1) & \text{for } |x| \geq y \\ \frac{1}{4}(x+1)(y-z+1) + \frac{1}{4}z(x+y) & \text{for } x \geq |y| \\ \frac{1}{4}(x+1)(y-z+1) + \frac{1}{2}xz & \text{for } |x| \leq y \\ \frac{1}{4}(x+1)(y-z+1) + \frac{1}{4}z(x-y) & \text{for } x \leq |y| \end{cases}$$

$$p_3^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(-x+1)(y-z+1) & \text{for } |x| \geq y \\ \frac{1}{4}(-x+1)(y-z+1) - \frac{1}{4}z(x+y) & \text{for } x \geq |y| \\ \frac{1}{4}(-x+1)(y-z+1) - \frac{1}{2}xz & \text{for } |x| \leq y \\ \frac{1}{4}(-x+1)(y-z+1) - \frac{1}{4}z(x-y) & \text{for } x \leq |y| \end{cases}$$

$$p_4^{sym}(x,y,z) = z.$$

Theorem *3.1.1*

The basis functions $p_i^{sym}, i = 0,\ldots,4$, *satisfy the following conditions:*

1. $p_i^{sym}(A_j) = \delta_{ij}$, $i, j = 0,1,2,3,4$.
2. Each basis function is bilinear on the base of K (contained in the plane $z = 0$).
3. Each basis function is linear on all triangular faces of K.
4. Each basis function is continuous on the interelement boundaries of K (contained in the planes $x = y$ and $x = -y$).
5. The sum of the basis functions is unity at any point in the pyramid (i.e., $\sum p_i^{sym}(x,y,z) \equiv 1$, $i = 0,\ldots,4$).
6. The basis functions p_i^{sym}, $i = 0,\ldots,4$ vanish on all faces not containing node i.

Proof :

The proof is an immediate consequence of (3.1.3), (3.1.4), and Lemma 3.1.1. □

This averaging helps us in reducing the discretization error coming from anisotropy of Wieners' composite elements. The new basis functions are now more symmetric on K and the number of degrees of freedom is five as for the simplest Wieners' element.

3.2 The Thirteen Node Pyramidal Element

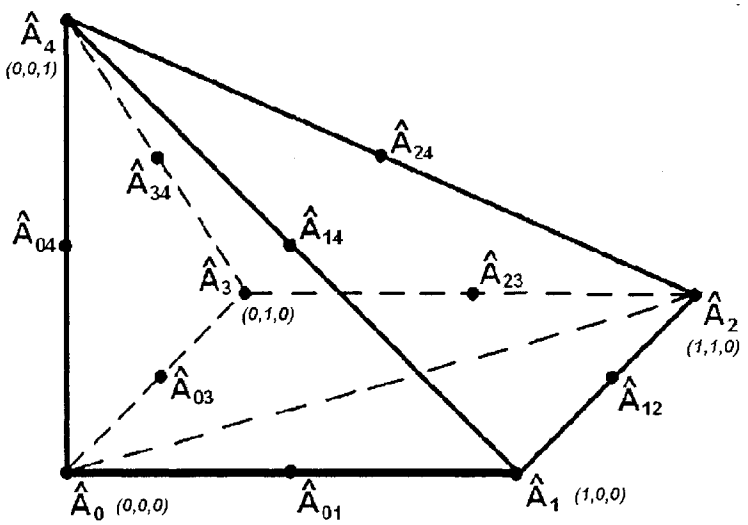


Figure 3.6. 13-node quadratic reference element $\hat{K}$.

13-node Pyramidal Finite Element Basis Functions

We will now extend the ideas presented in the previous section to the case of quadratic basis functions and construct a more complicated pyramidal finite element with 13 degrees of freedom. The triquadratic finite element basis functions from [Wieners, 1997] are defined on the reference pyramid $\hat{K}$ (see Figure 3.6) as follows:

$$\hat{q}_0(\hat{x},\hat{y},\hat{z}) = \begin{cases} ((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{y}-1))(1-2\hat{x}-2\hat{y}-2\hat{z}) & \text{for } \hat{x} > \hat{y} \\ ((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{x}-1))(1-2\hat{x}-2\hat{y}-2\hat{z}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_1(\hat{x},\hat{y},\hat{z}) = \begin{cases} (\hat{x}(1-\hat{y})-\hat{z}\hat{y})(2\hat{x}-2\hat{y}-1) & \text{for } \hat{x} > \hat{y} \\ (\hat{x}(1-\hat{y})-\hat{z}\hat{x})(2\hat{x}-2\hat{y}-1) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_2(\hat{x},\hat{y},\hat{z}) = \begin{cases} (\hat{x}\hat{y}+\hat{z}\hat{y})(2\hat{x}+2\hat{y}+2\hat{z}-3) & \text{for } \hat{x} > \hat{y} \\ (\hat{x}\hat{y}+\hat{z}\hat{x})(2\hat{x}+2\hat{y}+2\hat{z}-3) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_3(\hat{x},\hat{y},\hat{z}) = \begin{cases} ((1-\hat{x})\hat{y}-\hat{z}\hat{y})(2\hat{y}-2\hat{x}-1) & \text{for } \hat{x} > \hat{y} \\ ((1-\hat{x})\hat{y}-\hat{z}\hat{x})(2\hat{y}-2\hat{x}-1) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_4(\hat{x},\hat{y},\hat{z}) = \hat{z}(2\hat{z}-1)$$

$$\hat{q}_{01}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{x}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{y}-1)) - 2\hat{y}\hat{z}(1-\hat{x}-\hat{z}) & \text{for } \hat{x} > \hat{y} \\ 4\hat{x}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{x}-1)) - 2\hat{x}\hat{z}(1-\hat{y}-\hat{z}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_{12}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{y}(\hat{x}(1-\hat{y})-\hat{z}\hat{y})+2\hat{y}\hat{z}(1-\hat{x}-\hat{z}) & \text{for } \hat{x} > \hat{y} \\ 4\hat{y}(\hat{x}(1-\hat{y})-\hat{z}\hat{x})+2\hat{x}\hat{z}(1-\hat{y}-\hat{z}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_{23}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{x}((1-\hat{x})\hat{y}-\hat{z}\hat{y})+2\hat{y}\hat{z}(1-\hat{x}-\hat{z}) & \text{for } \hat{x} > \hat{y} \\ 4\hat{x}((1-\hat{x})\hat{y}-\hat{z}\hat{x})+2\hat{x}\hat{z}(1-\hat{y}-\hat{z}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_{03}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{y}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{y}-1)) - 2\hat{y}\hat{z}(1-\hat{x}-\hat{z}) & \text{for } \hat{x} > \hat{y} \\ 4\hat{y}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{x}-1)) - 2\hat{x}\hat{z}(1-\hat{y}-\hat{z}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_{04}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{y}-1)) & \text{for } \hat{x} > \hat{y} \\ 4\hat{z}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{x}-1)) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_{14}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}(\hat{x}(1-\hat{y})-\hat{z}\hat{y}) & \text{for } \hat{x} > \hat{y} \\ 4\hat{z}(\hat{x}(1-\hat{y})-\hat{z}\hat{x}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_{24}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}(\hat{x}\hat{y}+\hat{z}\hat{y}) & \text{for } \hat{x} > \hat{y} \\ 4\hat{z}(\hat{x}\hat{y}+\hat{z}\hat{x}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{q}_{34}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}((1-\hat{x})\hat{y}-\hat{z}\hat{y}) & \text{for } \hat{x} > \hat{y} \\ 4\hat{z}((1-\hat{x})\hat{y}-\hat{z}\hat{x}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

where $\hat{q}_{ij}$ corresponds to node $\hat{A}_{ij}$ located at the midpoint of the edge $\left(\hat{A}_i,\hat{A}_j\right)$.

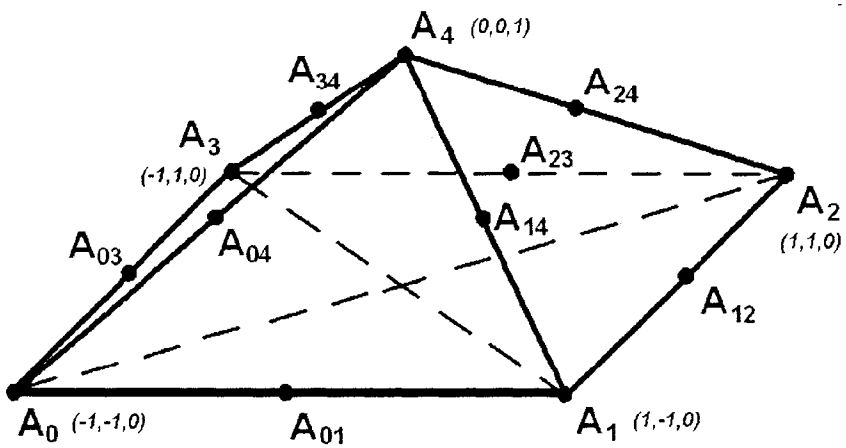


Figure 3.7. 13-node reference element K.

By again applying the linear affine transformation defined by (3.1.1) we find that piecewise basis functions on the new reference pyramid K (see Figure 3.7) are of the form

$$q_0(x,y,z) = \begin{cases} \frac{1}{4}(x+y+1)(x+z-1)(-y+z+1) & \text{for } x > y \\ \frac{1}{4}(x+y+1)(-x+z+1)(y+z-1) & \text{for } x \le y \end{cases}$$

$$q_1(x,y,z) = \begin{cases} \frac{1}{4}(x-y-1)[(x+z+1)(-y+z+1)-4z] & \text{for } x > y \\ \frac{1}{4}(x-y-1)(x-z+1)(-y-z+1) & \text{for } x \le y \end{cases}$$

$$q_2(x,y,z) = \begin{cases} \frac{1}{4}(x+y-1)(x+z+1)(y-z+1) & \text{for } x > y \\ \frac{1}{4}(x+y-1)(x-z+1)(y+z+1) & \text{for } x \le y \end{cases}$$

$$q_3(x,y,z) = \begin{cases} \frac{1}{4}(x-y+1)(x+z-1)(y-z+1) & \text{for } x > y \\ \frac{1}{4}(x-y+1)[(x-z-1)(y+z+1)+4z] & \text{for } x \le y \end{cases}$$

$$q_4(x,y,z) = z(2z-1)$$

$$q_{01}(x,y,z) = \begin{cases} \frac{1}{2}(x+z-1)[(y-z-1)(x+1)+2z] & \text{for } x > y \\ \frac{1}{2}(x-z+1)(y+z-1)(x-1) & \text{for } x \le y \end{cases}$$

$$q_{12}(x,y,z) = \begin{cases} -\frac{1}{2}(y-z+1)[(x+z+1)(y-1)+2z] & \text{for } x > y \\ -\frac{1}{2}(x-z+1)(y+z-1)(y+1) & \text{for } x \le y \end{cases} \qquad (3.2.1)$$

$$q_{23}(x,y,z) = \begin{cases} -\frac{1}{2}(y-z+1)(x+z-1)(x+1) & \text{for } x > y \\ -\frac{1}{2}(x-z+1)[(y+z+1)(x-1)+2z] & \text{for } x \le y \end{cases}$$

$$q_{03}(x,y,z) = \begin{cases} \frac{1}{2}(y-z+1)(x+z-1)(y-1) & \text{for } x > y \\ \frac{1}{2}(y+z-1)[(x-z-1)(y+1)+2z] & \text{for } x \le y \end{cases}$$

$$q_{04}(x,y,z) = \begin{cases} z(y-z-1)(x+z-1) & \text{for } x > y \\ z(x-z-1)(y+z-1) & \text{for } x \le y \end{cases}$$

$$q_{14}(x,y,z) = \begin{cases} -z[(x+z+1)(y-z-1)+4z] & \text{for } x > y \\ -z(x-z+1)(y+z-1) & \text{for } x \le y \end{cases}$$

$$q_{24}(x,y,z) = \begin{cases} z(y-z+1)(x+z+1) & \text{for } x > y \\ z(x-z+1)(y+z+1) & \text{for } x \le y \end{cases}$$

$$q_{34}(x,y,z) = \begin{cases} -z(y-z+1)(x+z-1) & \text{for } x > y \\ -z[(y+z+1)(x-z-1)+4z] & \text{for } x \le y \end{cases}$$

Lemma *3.2.1*

The basis functions $q_0, \ldots, q_4, q_{01}, \ldots, q_{34}$ *satisfy the following conditions:*

1. $q_i(A_j) = \delta_{ij}$, $i, j \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$.
2. Each basis function is biquadratic on the base of K (contained in the plane $z = 0$).
3. Each basis function is quadratic on all triangular faces of K.
4. Each basis function is continuous on the interelement boundary of K (contained in the plane $x = y$).
5. The sum of the basis functions is unity at any point in the pyramid (i.e., $\sum q_i(x,y,z) \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$).
6. The basis functions q_i, $i \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$ vanish on all faces not containing node i.

Proof :

1. This can be easily verified by direct calculation.
2. Setting $z = 0$, we immediately find that $q_0, \ldots, q_4, q_{01}, \ldots, q_{34}$ are biquadratic, for instance, $q_0(x,y,0) = (x+y+1)(x-1)(-y+1)/4$.
3. On the face $A_0A_1A_4$, which is contained in the plane $z = 1 + y$:

$$q_2(x,y)|_{\{x>y\}} = q_3(x,y)|_{\{x>y\}} = q_{12}(x,y)|_{\{x>y\}} = q_{23}(x,y)|_{\{x>y\}} = 0$$
$$q_{03}(x,y)|_{\{x>y\}} = q_{24}(x,y)|_{\{x>y\}} = q_{34}(x,y)|_{\{x>y\}} = 0$$
$$q_0(x,y)|_{\{x>y\}} = \tfrac{1}{2}(x+y+1)(x+y)$$
$$q_1(x,y)|_{\{x>y\}} = \tfrac{1}{2}(x-y-1)(x-y)$$
$$q_4(x,y)|_{\{x>y\}} = (1+y)(1+2y)$$
$$q_{01}(x,y)|_{\{x>y\}} = (x+y)(-x+y)$$
$$q_{04}(x,y)|_{\{x>y\}} = -2(1+y)(x+y)$$
$$q_{14}(x,y)|_{\{x>y\}} = 2(1+y)(x-y).$$

   On the faces $A_1A_2A_4$, $A_2A_3A_4$ and $A_0A_3A_4$ this can be done similarly.
4. By setting $x = y$, we can easily see that the functions $q_0, \ldots, q_4, q_{01}, \ldots, q_{34}$ are continuous in the plane $x = y$.
5. This can be verified by summing the basis functions. For instance, in the case of $x > y$ we have:

$$\begin{aligned}\sum q_i(x,y)|_{\{x>y\}} = {} & \tfrac{1}{4}(x+y+1)(x+z-1)(-y+z+1) \\ & + \tfrac{1}{4}(x-y-1)[(x+z+1)(-y+z+1) - 4z] \\ & + \tfrac{1}{4}(x+y-1)(x+z+1)(y-z+1) \\ & + \tfrac{1}{4}(x-y+1)(x+z-1)(y-z+1) \\ & + z(2z-1) \\ & + \tfrac{1}{2}(x+z-1)[(y-z-1)(x+1) + 2z] \\ & - \tfrac{1}{2}(y-z+1)[(x+z+1)(y-1) + 2z] \\ & - \tfrac{1}{2}(y-z+1)(x+z-1)(x+1) \\ & + \tfrac{1}{2}(y-z+1)(x+z-1)(y-1) \\ & + z(y-z-1)(x+z-1)\end{aligned}$$

$$
\begin{aligned}
&- z[(x+z+1)(y-z-1)+4z] \\
&+ z(y-z+1)(x+z+1) \\
&- z(y-z+1)(x+z-1) \\
&\equiv 1.
\end{aligned}
$$

Similarly, we find that $\sum q_i(x,y)|_{\{x\le y\}} \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$.

6. This can be verified by a quick calculation. For instance, the two faces that do not contain node A_0 are the face $A_1A_2A_4$ contained in the plane $z = 1-x$, and the face $A_2A_3A_4$ contained in the plane $z = 1-y$. In the case of $A_1A_2A_4$ we have:

$$q_0(x,y)|_{\{x>y\}} = \tfrac{1}{4}(x+y+1)(x+(1-x)-1)(-y+(1-x)+1) = 0$$

Similarly, on the face $A_2A_3A_4$:

$$q_0(x,y)|_{\{x\le y\}} = \tfrac{1}{4}(x+y+1)(-x+(1-y)+1)(y+(1-y)-1) = 0.$$

Similar calculations for the remaining basis functions and appropriate faces yield the same result. □

In order to get more symmetries, we again take the averages of the basis functions, q_i, $i \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$, with their mirror images. In other words, we again consider a mirror image mapping

$$M : (x,y,z) \to (-x,y,z)$$

and define

$$
\begin{aligned}
\overline{q}_0(x,y,z) &= q_1(-x,y,z) \\
\overline{q}_1(x,y,z) &= q_0(-x,y,z) \\
\overline{q}_2(x,y,z) &= q_3(-x,y,z) \\
\overline{q}_3(x,y,z) &= q_2(-x,y,z) \\
\overline{q}_{01}(x,y,z) &= q_{01}(-x,y,z) \\
\overline{q}_{12}(x,y,z) &= q_{03}(-x,y,z) \\
\overline{q}_{23}(x,y,z) &= q_{23}(-x,y,z) \\
\overline{q}_{03}(x,y,z) &= q_{12}(-x,y,z) \\
\overline{q}_{04}(x,y,z) &= q_{14}(-x,y,z) \\
\overline{q}_{14}(x,y,z) &= q_{04}(-x,y,z) \\
\overline{q}_{24}(x,y,z) &= q_{34}(-x,y,z) \\
\overline{q}_{34}(x,y,z) &= q_{24}(-x,y,z).
\end{aligned}
\qquad (3.2.2)
$$

Setting

$$
\begin{aligned}
q_i^{sym} &= \tfrac{1}{2}(q_i + \overline{q}_i) \quad \text{for } i \in \{0,1,2,3,01,12,23,03,04,14,24,34\} \\
q_i^{sym} &= q_4
\end{aligned}
\qquad (3.2.3)
$$

we find that

$$q_0^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x+y+1)[(-y+z+1)(x-1)+2z] & \text{for } |x| \geq y \\ \frac{1}{4}(x+y+1)(x+z-1)(-y+1) & \text{for } x \geq |y| \\ \frac{1}{4}(x+y+1)(y+z-1)(-x+1) & \text{for } |x| \leq y \\ \frac{1}{4}(x+y+1)[(-x+z+1)(y-1)+2z] & \text{for } x \leq |y| \end{cases}$$

$$q_1^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x-y-1)[(-y+z+1)(x+1)-2z] & \text{for } |x| \geq y \\ \frac{1}{4}(x-y-1)[(x+z+1)(-y+1)-2z] & \text{for } x \geq |y| \\ \frac{1}{4}(x-y-1)(-y-z+1)(x+1) & \text{for } |x| \leq y \\ \frac{1}{4}(x-y-1)(x-z+1)(-y+1) & \text{for } x \leq |y| \end{cases}$$

$$q_2^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x+y-1)(y-z+1)(x+1) & \text{for } |x| \geq y \\ \frac{1}{4}(x+y-1)[(x+z+1)(y+1)-2z] & \text{for } x \geq |y| \\ \frac{1}{4}(x+y-1)[(y+z+1)(x+1)-2z] & \text{for } |x| \leq y \\ \frac{1}{4}(x+y-1)(x-z+1)(y+1) & \text{for } x \leq |y| \end{cases}$$

$$q_3^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x-y+1)(y-z+1)(x-1) & \text{for } |x| \geq y \\ \frac{1}{4}(x-y+1)(x+z-1)(y+1) & \text{for } x \geq |y| \\ \frac{1}{4}(x-y+1)[(y+z+1)(x-1)+2z] & \text{for } |x| \leq y \\ \frac{1}{4}(x-y+1)[(x-z-1)(y+1)+2z] & \text{for } x \leq |y| \end{cases}$$

$$q_4^{sym}(x,y,z) = z(2z-1)$$

$$q_{01}^{sym}(x,y,z) = \begin{cases} \frac{1}{2}(y+z-1)[(x+1)(x-1)+z]-zx^2 & \text{for } |x| \geq y \\ \frac{1}{2}(x+z-1)[(x+1)(y-1)+z] & \text{for } x \geq |y| \\ \frac{1}{2}(y+z-1)[(x+1)(x-1)+z] & \text{for } |x| \leq y \\ \frac{1}{2}(x-z+1)[(x-1)(y-1)-z] & \text{for } x \leq |y| \end{cases}$$

$$q_{12}^{sym}(x,y,z) = \begin{cases} -\frac{1}{2}(y-z+1)[(x+1)(y-1)+z] & \text{for } |x| \geq y \\ -\frac{1}{2}(x-z+1)[(y+1)(y-1)+z]-zy^2 & \text{for } x \geq |y| \\ -\frac{1}{2}(y+z-1)[(x+1)(y+1)-z] & \text{for } |x| \leq y \\ -\frac{1}{2}(x-z+1)[(y+1)(y-1)+z] & \text{for } x \leq |y| \end{cases}$$

$$q_{23}^{sym}(x,y,z) = \begin{cases} -\frac{1}{2}(y-z+1)[(x+1)(x-1)+z] & \text{for } |x| \geq y \\ -\frac{1}{2}(x+z-1)[(x+1)(y+1)-z] & \text{for } x \geq |y| \\ -\frac{1}{2}(y-z+1)[(x+1)(x-1)+z]-zx^2 & \text{for } |x| \leq y \\ -\frac{1}{2}(x-z+1)[(x-1)(y+1)+z] & \text{for } x \leq |y| \end{cases}$$

$$q_{03}^{sym}(x,y,z) = \begin{cases} \frac{1}{2}(y-z+1)[(x-1)(y-1)-z] & \text{for } |x| \geq y \\ \frac{1}{2}(x+z-1)[(y+1)(y-1)+z] & \text{for } x \geq |y| \\ \frac{1}{2}(y+z-1)[(x-1)(y+1)+z] & \text{for } |x| \leq y \\ \frac{1}{2}(x+z-1)[(y+1)(y-1)+z]-zy^2 & \text{for } x \leq |y| \end{cases}$$

$$q_{04}^{sym}(x,y,z) = \begin{cases} z[(y-z-1)(x-1)-2z] & \text{for } |x| \geq y \\ z(x+z-1)(y-1) & \text{for } x \geq |y| \\ z(y+z-1)(x-1) & \text{for } |x| \leq y \\ z[(x-z-1)(y-1)-2z] & \text{for } x \leq |y| \end{cases}$$

$$q_{14}^{sym}(x,y,z) = \begin{cases} -z[(y-z-1)(x+1)+2z] & \text{for } |x| \geq y \\ -z[(x+z+1)(y-1)+2z] & \text{for } x \geq |y| \\ -z(y+z-1)(x+1) & \text{for } |x| \leq y \\ -z(x-z+1)(y-1) & \text{for } x \leq |y| \end{cases}$$

$$q_{24}^{sym}(x,y,z) = \begin{cases} z(y-z+1)(x+1) & \text{for } |x| \geq y \\ z[(x+z+1)(y+1)-2z] & \text{for } x \geq |y| \\ z[(y+z+1)(x+1)-2z] & \text{for } |x| \leq y \\ z(x-z+1)(y+1) & \text{for } x \leq |y| \end{cases}$$

$$q_{34}^{sym}(x,y,z) = \begin{cases} z(y-z+1)(-x+1) & \text{for } |x| \geq y \\ z(-x-z+1)(y+1) & \text{for } x \geq |y| \\ z[(y+z+1)(-x+1)-2z] & \text{for } |x| \leq y \\ z[(-x+z+1)(y+1)-2z] & \text{for } x \leq |y|. \end{cases}$$

Theorem *3.2.1*

The basis functions $q_0^{sym}, \ldots, q_4^{sym}, q_{01}^{sym}, \ldots, q_{34}^{sym}$ *satisfy the following conditions:*

1. $q_i^{sym}(A_j) = \delta_{ij}$, $i, j \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$.
2. Each basis function is biquadratic on the base of K (contained in the plane $z = 0$).
3. Each basis function is quadratic on all triangular faces of K.
4. Each basis function is continuous on the interelement boundaries of K (contained in the planes $x = y$ and $x = -y$).

5. The sum of the basis functions is unity at any point in the pyramid (i.e., $\sum q_i^{sym}(x,y,z) \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$).
6. The basis functions q_i^{sym}, $i \in \{0,1,2,3,4,01,12,23,03,04,14,24,34\}$ vanish on all faces not containing node i.

Proof :
The proof is an immediate consequence of (3.2.1), (3.2.2), and Lemma 3.2.1. □

3.3 The Fourteen Node Pyramidal Element

In this section we will present the basis functions for a 14-node pyramidal finite element. These elements, with nine nodes on the square base and six nodes on each triangular face, provide us with a useful tool for interfacing between ten node tetrahedral elements and twenty-seven node hexahedral elements. This is in contrast to the thirteen node pyramidal elements discussed in the previous section which are not suitable for a face-to-face connection with a twenty-seven node hexahedron.

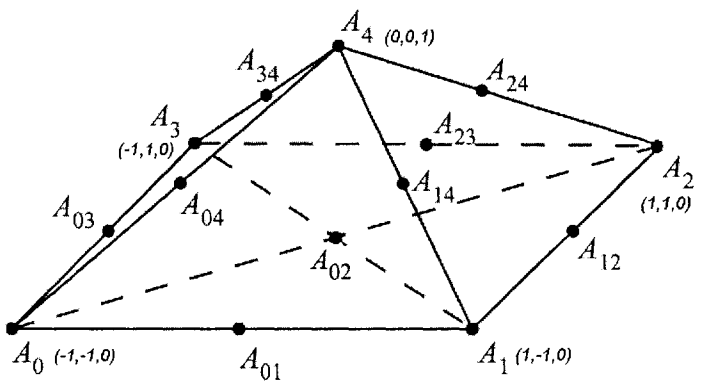


Figure 3.8. 14-node reference element K.

Development of Basis Functions

In the case of the fourteen node element we consider a total of four sets of basis functions. The first set is developed on a reference element with the same coordinates as $\hat{K}$ introduced in section 3.1, and then mapped to a reference element K (Figure 3.8) using the affine transformation (3.1.1). The second set of basis functions are developed directly on the element K in order to reduce any unnecessary complexity in the equations. However, we shall see that several of the functions are common between the two sets, as the choice is limited due to the constraints involved. In particular, the functions at the peak of the pyramid as well as those at the midpoints of the edge between the base and the peak are identical, and are also the same as those used in the case of the thirteen node element. Note, in fact that these five particular functions are similar to the five functions defining a linear element, only differing by a constant and a factor of z (the equation for the base of the pyramid). The process introduced in section 3.1 of averaging the functions with their mirror images is then applied to each of these sets in order to develop two more sets of highly symmetric basis functions. During the development of the different cases of basis functions for the 14-node element, great care was

taken to ensure that the functions would produce the most accurate results. The two cases presented here represent the best two sets of functions out of several sets of functions that were considered. They also serve to better illustrate the fact that changing the basis functions of an element alters the accuracy of the element.

14-node Pyramidal Finite Element Basis Functions

Case I Basis Functions:

We define the following piecewise basis functions on the composite reference pyramidal element $\hat{K}$:

$$\hat{r}_0(\hat{x},\hat{y},\hat{z}) = \begin{cases} (\hat{x}+\hat{z}-1)(\hat{y}-1)(2\hat{y}-1)(2\hat{x}+2\hat{z}-1) & \text{for } \hat{x} > \hat{y} \\ (\hat{y}+\hat{z}-1)(\hat{x}-1)(2\hat{y}+2\hat{z}-1)(2\hat{x}-1) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{r}_1(\hat{x},\hat{y},\hat{z}) = \begin{cases} -(2\hat{x}+2\hat{z}-1)(2\hat{y}-1)(\hat{x}(1-\hat{y})-\hat{z}\hat{y})-2\hat{z}(\hat{x}-\hat{y}) & \text{for } \hat{x} > \hat{y} \\ -(2\hat{y}+2\hat{z}-1)(2\hat{x}-1)(\hat{x}(1-\hat{y})-\hat{z}\hat{x}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{r}_2(\hat{x},\hat{y},\hat{z}) = \begin{cases} (\hat{x}+\hat{z})(2\hat{x}+2\hat{z}-1)\hat{y}(2\hat{y}-1) & \text{for } x > y \\ (\hat{y}+\hat{z})(2\hat{y}+2\hat{z}-1)\hat{x}(2\hat{x}-1) & \text{for } x \le y \end{cases}$$

$$\hat{r}_3(\hat{x},\hat{y},\hat{z}) = \begin{cases} -(2\hat{y}-1)(2\hat{x}+2\hat{z}-1)((1-\hat{x})\hat{y}-\hat{z}\hat{y}) & \text{for } \hat{x} > \hat{y} \\ -(2\hat{x}-1)(2\hat{y}+2\hat{z}-1)((1-\hat{x})\hat{y}-\hat{z}\hat{x})+2\hat{z}(\hat{x}-\hat{y}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{r}_4(\hat{x},\hat{y},\hat{z}) = \hat{z}(2\hat{z}-1)$$

$$\hat{r}_{01}(\hat{x},\hat{y},\hat{z}) = \begin{cases} -4\hat{x}((1-\hat{x})(1-\hat{y})(2\hat{y}-1)+\hat{z}(\hat{y}-1)(2\hat{y}-1)) & \text{for } \hat{x} > \hat{y} \\ -4\hat{x}((1-\hat{x})(1-\hat{y})(2\hat{y}-1)+\hat{z}(\hat{x}-1)(2\hat{y}-1)) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{r}_{12}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{y}(\hat{x}(1-\hat{y})(2\hat{x}+2\hat{z}-1)-\hat{z}\hat{y}(2\hat{x}+2\hat{z}-1)) & \text{for } \hat{x} > \hat{y} \\ 4\hat{y}(\hat{x}(1-\hat{y})(2\hat{x}+2\hat{z}-1)-\hat{z}\hat{x}(2\hat{x}+2\hat{z}-1)) & \text{for } \hat{x} \le \hat{y} \end{cases} \qquad (3.3.1)$$

$$\hat{r}_{23}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{x}((1-\hat{x})(2\hat{y}+2\hat{z}-1)\hat{y}-\hat{z}\hat{y}(2\hat{y}+2\hat{z}-1)) & \text{for } \hat{x} > \hat{y} \\ 4\hat{x}((1-\hat{x})(2\hat{y}+2\hat{z}-1)\hat{y}-\hat{z}\hat{x}(2\hat{y}+2\hat{z}-1)) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{r}_{03}(\hat{x},\hat{y},\hat{z}) = \begin{cases} -4\hat{y}((1-\hat{x})(1-\hat{y})(2\hat{x}-1)+\hat{z}(\hat{y}-1)(2\hat{x}-1)) & \text{for } \hat{x} > \hat{y} \\ -4\hat{y}((1-\hat{x})(1-\hat{y})(2\hat{x}-1)+\hat{z}(\hat{x}-1)(2\hat{x}-1)) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{r}_{02}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 16\hat{x}(\hat{x}+\hat{z}-1)\hat{y}(\hat{y}-1)+4\hat{y}\hat{z}(\hat{x}+\hat{z}-1)(2\hat{x}+3-2\hat{y}) & \text{for } \hat{x} > \hat{y} \\ 16\hat{x}(\hat{x}-1)y(\hat{y}+\hat{z}-1)+4\hat{x}\hat{z}(\hat{y}+\hat{z}-1)(2\hat{y}+3-2\hat{x}) & \text{for } \hat{x} \le \hat{y} \end{cases}$$

$$\hat{r}_{04}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{y}-1)) & \text{for } x > y \\ 4\hat{z}((1-\hat{x})(1-\hat{y})+\hat{z}(\hat{x}-1)) & \text{for } x \le y \end{cases}$$

$$\hat{r}_{14}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}(\hat{x}(1-\hat{y})-\hat{z}\hat{y}) & \text{for } x > y \\ 4\hat{z}(\hat{x}(1-\hat{y})-\hat{z}\hat{x}) & \text{for } x \le y \end{cases}$$

$$\hat{r}_{24}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}(\hat{x}\hat{y}+\hat{z}\hat{y}) & \text{for } x > y \\ 4\hat{z}(\hat{x}\hat{y}+\hat{z}\hat{x}) & \text{for } x \le y \end{cases}$$

$$\hat{r}_{34}(\hat{x},\hat{y},\hat{z}) = \begin{cases} 4\hat{z}((1-\hat{x})\hat{y}-\hat{z}\hat{y}) & \text{for } x > y \\ 4\hat{z}((1-\hat{x})\hat{y}-\hat{z}\hat{x}) & \text{for } x \le y \end{cases}$$

By again applying the affine transformation $F_K\left(\hat{X}\right) = B\hat{X} + b$ defined by (3.2.1), we find that the basis functions on the reference element K take the form:

$$r_0(x,y,z) = \begin{cases} \frac{1}{4}(x+z-1)(y-z-1)(y-z)(x+z) & \text{if } x > y \\ \frac{1}{4}(y+z-1)(x-z-1)(y+z)(x-z) & \text{if } x \le y \end{cases}$$

$$r_1(x,y,z) = \begin{cases} \frac{1}{4}(z-y)(x+z)(x(1-y)+z(z+x-y-2)-y+1)-z(x-y) & \text{if } x > y \\ \frac{1}{4}(y+z)(x-z)(-1+y+z)(x-z+1) & \text{if } x \le y \end{cases}$$

$$r_2(x,y,z) = \begin{cases} \frac{1}{4}(1+x+z)(x+z)(-y+z-1)(z-y) & \text{if } x > y \\ \frac{1}{4}(y+z+1)(y+z)(x-z+1)(x-z) & \text{if } x \le y \end{cases}$$

$$r_3(x,y,z) = \begin{cases} \frac{1}{4}(z-y)(x+z)(-y+z-1)(x+z-1) & \text{if } x > y \\ \frac{1}{4}(y+z)(x-z)(z(-z+x-y+2)+x(y+1)-y-1)+z(x-y) & \text{if } x \le y \end{cases}$$

$$r_4(x,y,z) = z(2z-1)$$

$$r_{01}(x,y,z) = \begin{cases} -\frac{1}{2}(x-z+1)(y-z)(y-z-1)(x+z-1) & \text{if } x > y \\ -\frac{1}{2}(x-z+1)(y+z-1)(y-z)(x-z-1) & \text{if } x \le y \end{cases}$$

$$r_{12}(x,y,z) = \begin{cases} -\frac{1}{2}(-y+z-1)(x+z)((z+x-y-2)z+x(1-y)-y+1) & \text{if } x > y \\ \frac{1}{2}(-y+z-1)(x+z)(x-z+1)(-1+y+z) & \text{if } x \le y \end{cases}$$

$$r_{23}(x,y,z) = \begin{cases} \frac{1}{2}(x-z+1)(y+z)(-y+z-1)(x+z-1) & \text{if } x > y \\ -\frac{1}{2}(x-z+1)(y+z)((-z+x-y+2)z+x(y+1)-y-1) & \text{if } x \le y \end{cases}$$

$$r_{03}(x,y,z) = \begin{cases} -\frac{1}{2}(-y+z-1)(x+z-1)(x-z)(-y+1+z) & \text{if } x > y \\ \frac{1}{2}(-y+z-1)(x-z)(x-z-1)(-1+y+z) & \text{if } x \le y \end{cases}$$

$$r_{02}(x,y,z) = \begin{cases} -(-y+z-1)(x+z-1)((y-1)x+z(z-2y+3)+y-1) & \text{if } x > y \\ -(-1+y+z)(x-z+1)(z(-z+2x-3)-x(1+y)+y+1) & \text{if } x \le y \end{cases}$$

$$r_{04}(x,y,z) = \begin{cases} z(y-z-1)(x+z-1) & \text{for } x > y \\ z(x-z-1)(y+z-1) & \text{for } x \le y \end{cases}$$

$$r_{14}(x,y,z) = \begin{cases} -z((x+z+1)(y-z-1)+4z) & \text{for } x > y \\ -z(x-z+1)(y+z-1) & \text{for } x \le y \end{cases}$$

$$r_{24}(x,y,z) = \begin{cases} z(y-z+1)(x+z+1) & \text{for } x > y \\ z(x-z+1)(y+z+1) & \text{for } x \le y \end{cases}$$

$$r_{34}(x,y,z) = \begin{cases} -z(y-z+1)(x+z-1) & \text{for } x > y \\ -z((y+z+1)(x-z-1)+4z) & \text{for } x \le y \end{cases}$$

Lemma *3.3.1*

The basis functions $r_0,\ldots,r_4,r_{01},\ldots,r_{34}$ *satisfy the following conditions:*

1. $r_i(A_j) = \delta_{ij}$, $i, j \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$.
2. Each basis function is biquadratic on the base of K (contained in the plane $z = 0$).
3. Each basis function is quadratic on all triangular faces of K.
4. Each basis function is continuous on the interelement boundary of K (contained in the plane $x = y$).
5. The sum of the basis functions is unity at any point in the pyramid (i.e., $\sum r_i(x,y,z) \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$).
6. The basis functions r_i, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$ vanish on all faces not containing node i.

Proof :

1. This can be easily verified by direct calculation.
2. Setting $z = 0$, we immediately see that r_i, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$ are biquadratic, for instance, $r_0(x,y,0) = \frac{1}{4}(x-1)(y-1)yx$.
3. On the face $A_0A_1A_4$, which is contained in the plane $y = z - 1$, we have

$$\begin{aligned}
r_0(x,y)|_{\{x>y\}} &= \tfrac{1}{2}(x+z-1)(x+z) \\
r_1(x,y)|_{\{x>y\}} &= \tfrac{1}{2}(z-x-1)(z-x) \\
r_2(x,y)|_{\{x>y\}} &= r_3(x,y)|_{\{x>y\}} = 0, \\
r_{01}(x,y)|_{\{x>y\}} &= (z-x-1)(x+z-1) \\
r_{04}(x,y)|_{\{x>y\}} &= -2z(x+z-1) \\
r_{14}(x,y)|_{\{x>y\}} &= 2z(x-z+1) \\
r_{12}(x,y)|_{\{x>y\}} &= r_{23}(x,y)|_{\{x>y\}} = r_{03}(x,y)|_{\{x>y\}} \\
&= r_{02}(x,y)|_{\{x>y\}} = r_{24}(x,y)|_{\{x>y\}} \\
&= r_{34}(x,y)|_{\{x>y\}} = 0 \\
r_4(x,y)|_{\{x>y\}} &= z(2z-1).
\end{aligned}$$

4. By setting $x = y$, we can easily see that the functions $r_i, i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$ are continuous in the $x = y$ plane.

5. This can be verified by summing the basis functions. For instance, in the case of $x > y$ we have:

$$\begin{aligned}\sum r_i(x,y)|_{\{x>y\}} &= (\tfrac{1}{4}(x+z-1)(y-z-1)(y-z)(x+z))\\ &+ (\tfrac{1}{4}(z-y)(x+z)(x(1-y)+z(z+x-y-2)-y+1)-z(x-y))\\ &+ (\tfrac{1}{4}(1+x+z)(x+z)(-y+z-1)(z-y))\\ &+ (\tfrac{1}{4}(z-y)(x+z)(-y+z-1)(x+z-1))\\ &+ (z(2z-1))\\ &+ (-\tfrac{1}{2}(x-z+1)(y-z)(y-z-1)(x+z-1))\\ &+ (-\tfrac{1}{2}(-y+z-1)(x+z)((z+x-y-2)z+x(1-y)-y+1))\\ &+ (\tfrac{1}{2}(x-z+1)(y+z)(-y+z-1)(x+z-1))\\ &+ (-\tfrac{1}{2}(-y+z-1)(x+z-1)(x-z)(-y+1+z))\\ &+ (-(-y+z-1)(x+z-1)((y-1)x+z(z-2y+3)+y-1))\\ &+ (z(y-z-1)(x+z-1))\\ &+ (-z((x+z+1)(y-z-1)+4z))\\ &+ (z(y-z+1)(x+z+1))\\ &+ (-z(y-z+1)(x+z-1))\\ &\equiv 1.\end{aligned}$$

Similarly, we find that $\sum r_i(x,y)|_{\{x\le y\}} \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$.

6. This can be verified by quick calculation. For instance, the two faces that do not contain node A_0 are the face $A_1A_2A_4$ contained in the plane $z = 1 - x$, and the face $A_2A_3A_4$ contained in the plane $z = 1 - y$. In the case of $A_1A_2A_4$ we have:

$$r_0(x,y)|_{\{x>y\}} = \tfrac{1}{4}(x+(1-x)-1)(y-(1-x)-1)(y-(1-x))(x+(1-x)) = 0$$

Similarly, on the face $A_2A_3A_4$:

$$r_0(x,y)|_{\{x\le y\}} = \tfrac{1}{4}(y+(1-y)-1)(x-(1-y)-1)(y+(1-y))(x-(1-y)) = 0.$$

Similar calculations for the remaining basis functions and appropriate faces yield the same results. □

In order to develop a more symmetric element, we again apply the process of taking the averages of the basis functions r_i, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$, with their mirror images. Specifically, we consider the mirror image mapping

$$M : (x,y,z) \to (-x,y,z)$$

and define

$$
\begin{aligned}
\overline{r}_0(x,y,z) &= r_1(-x,y,z)\\
\overline{r}_1(x,y,z) &= r_0(-x,y,z)\\
\overline{r}_2(x,y,z) &= r_3(-x,y,z)\\
\overline{r}_3(x,y,z) &= r_2(-x,y,z)\\
\overline{r}_{01}(x,y,z) &= r_{01}(-x,y,z)\\
\overline{r}_{12}(x,y,z) &= r_{03}(-x,y,z)\\
\overline{r}_{23}(x,y,z) &= r_{23}(-x,y,z)\\
\overline{r}_{03}(x,y,z) &= r_{12}(-x,y,z)\\
\overline{r}_{02}(x,y,z) &= r_{02}(-x,y,z)\\
\overline{r}_{04}(x,y,z) &= r_{14}(-x,y,z)\\
\overline{r}_{14}(x,y,z) &= r_{04}(-x,y,z)\\
\overline{r}_{24}(x,y,z) &= r_{34}(-x,y,z)\\
\overline{r}_{34}(x,y,z) &= r_{24}(-x,y,z)
\end{aligned}
\qquad (3.3.2)
$$

Setting

$$
\begin{aligned}
r_i^{sym} &= \tfrac{1}{2}(r_i + \overline{r}_i) \quad \text{for } i \in \{0,1,2,3,01,12,23,03,02,04,14,24,34\}\\
r_i^{sym} &= q_4
\end{aligned}
\qquad (3.3.3)
$$

we find that

$$
r_0^{sym}(x,y,z) = \begin{cases}
\tfrac{1}{4}(z-y)(z(z(z-y)-z+x(1+x))-x(xy-x+1-y))-\tfrac{1}{2}z(-x-y) & \text{for } |x|>y\\
\tfrac{1}{4}(x+z)(x+z-1)(y^2+z^2-y) & \text{for } x \geq |y|\\
\tfrac{1}{4}(y+z)(y+z-1)(x^2+z^2-x) & \text{for } |x| \leq y\\
\tfrac{1}{4}(x-z)(y(-x+xy-y+1)+z(xz-y^2+z(1-z)-y))-\tfrac{1}{2}z(-x-y) & \text{for } x \leq |y|
\end{cases}
$$

$$
r_1^{sym}(x,y,z) = \begin{cases}
\tfrac{1}{4}(z-y)(z(z(z-1-y)+x(x-1))+x(x+1-xy-y))-\tfrac{1}{2}z(x-y) & \text{for } |x|>y\\
\tfrac{1}{4}(x+z)(z(xz+y(y+1)+z(z-1))+y(xy-x-1+y))-\tfrac{1}{2}z(x-y) & \text{for } x \geq |y|\\
\tfrac{1}{4}(y+z)(y+z-1)(x+x^2+z^2) & \text{for } |x| \leq y\\
\tfrac{1}{4}(x-z+1)(x-z)(y^2+z^2-y) & \text{for } x \leq |y|
\end{cases}
$$

$$
r_2^{sym}(x,y,z) = \begin{cases}
\tfrac{1}{4}(z-y)(-y+z-1)(z^2+x^2+x) & \text{for } |x|>y\\
\tfrac{1}{4}(x+z)(y(y+x+xy+1)+z(z(z+x-1)+y(y-1)))+\tfrac{1}{2}z(-x-y) & \text{for } x \geq |y|\\
\tfrac{1}{4}(y+z)(z(z(z+y-1)+x(x-1))+x(x(1+y)+1+y))+\tfrac{1}{2}z(-x-y) & \text{for } |x| \leq y\\
\tfrac{1}{4}(y^2+y+z^2)(x-z+1)(x-z) & \text{for } x \leq |y|
\end{cases}
$$

$$r_3^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(z-y)(-y+z-1)(z^2-x+x^2) & \text{for } |x| > y \\ \frac{1}{4}(y^2+y+z^2)(x+z)(x+z-1) & \text{for } x \geq |y| \\ \frac{1}{4}(y+z)(z(z(z+y-1)+x(1+x))+x(x(y+1)-1-y))+\frac{1}{2}z(x-y) & \text{for } |x| \leq y \\ \frac{1}{4}(x-z)(z(z(x-z+1)+y)+y(y(x-z-1)+x-1))+\frac{1}{2}z(x-y) & \text{for } x \leq |y| \end{cases}$$

$$r_4^{sym}(x,y,z) = z(2z-1)$$

$$r_{01}^{sym}(x,y,z) = \begin{cases} -\frac{1}{2}(x-z+1)(z-y)(-y+z+1)(x+z-1) & \text{for } |x| > y \\ \frac{1}{2}(z-y)(x+z-1)(xy-x+z^2-1+y) & \text{for } x \geq |y| \\ \frac{1}{2}(y+z-1)(z-y)(z^2+x^2-1) & \text{for } |x| \leq y \\ \frac{1}{2}(z-y)(x-z+1)(xy-x-z^2+1-y) & \text{for } x \leq |y| \end{cases}$$

$$r_{12}^{sym}(x,y,z) = \begin{cases} -\frac{1}{2}(-y+z-1)(x+z)(x(-y+z+1)-y-z+1) & \text{for } |x| > y \\ \frac{1}{2}(-y+z-1)(x+z)(y(x+z+1)-x-1+z) & \text{for } x \geq |y| \\ \frac{1}{2}(y+z-1)(-y+z-1)(x+1)(x+z) & \text{for } |x| \leq y \\ \frac{1}{2}(y-1)(-y+z-1)(x+z)(x-z+1) & \text{for } x \leq |y| \end{cases}$$

$$r_{23}^{sym}(x,y,z) = \begin{cases} \frac{1}{2}(x-z+1)(y+z)(-y+z-1)(x+z-1) & \text{for } |x| > y \\ -\frac{1}{2}(y+z)(x+z-1)(x(y+1)+z(z-2)+y+1) & \text{for } x \geq |y| \\ -\frac{1}{2}(y+z)(z(z(z-3+y)+x^2+3)-y-1+x(x+xy)) & \text{for } |x| \leq y \\ -\frac{1}{2}(y+z)(x-z+1)(x(y+1)+z(2-z)-y-1) & \text{for } x \leq |y| \end{cases}$$

$$r_{03}^{sym}(x,y,z) = \begin{cases} -\frac{1}{2}(-y+z-1)(x-z)(xz+z-xy+x+y-1) & \text{for } |x| > y \\ \frac{1}{2}(y-1)(-y+z-1)(x+z-1)(x-z) & \text{for } x \geq |y| \\ \frac{1}{2}(z+y-1)(-y+z-1)(x-1)(x-z) & \text{for } |x| \leq y \\ \frac{1}{2}(-y+z-1)(x-z)(-z-zy+xy-x-y+1) & \text{for } x \leq |y| \end{cases}$$

$$r_{02}^{sym}(x,y,z) = \begin{cases} -(-y+z-1)(z(z(z+2-2y)-4+3y)+1+x(x(y-1))-y) & \text{for } |x| > y \\ -(x+z-1)(x+z(z(x+z-y+2)-4+x(y-2))+1+y(y(z-1-x)+z)) & \text{for } x \geq |y| \\ -(z+y-1)(z(z(z+2)-4-y)+1+y+x(x(2z-y-1))) & \text{for } |x| \leq y \\ -(x+1-z)(x+z(z(y+x-z-2)+x(y-2)+4)-1+y(y(1-z-x)-z)) & \text{for } x \leq |y| \end{cases}$$

$$r_{04}^{sym}(x,y,z) = \begin{cases} z[(y-z-1)(x-1)-2z] & \text{for } |x| \geq y \\ z(x+z-1)(y-1) & \text{for } x \geq |y| \\ z(y+z-1)(x-1) & \text{for } |x| \leq y \\ z[(x-z-1)(y-1)-2z] & \text{for } x \leq |y| \end{cases}$$

$$r_{14}^{sym}(x,y,z) = \begin{cases} -z[(y-z-1)(x+1)+2z] & \text{for } |x| \geq y \\ -z[(x+z+1)(y-1)+2z] & \text{for } x \geq |y| \\ -z(y+z-1)(x+1) & \text{for } |x| \leq y \\ -z(x-z+1)(y-1) & \text{for } x \leq |y| \end{cases}$$

$$r_{24}^{sym}(x,y,z) = \begin{cases} z(y-z+1)(x+1) & \text{for } |x| \geq y \\ z[(x+z+1)(y+1)-2z] & \text{for } x \geq |y| \\ z[(y+z+1)(x+1)-2z] & \text{for } |x| \leq y \\ z(x-z+1)(y+1) & \text{for } x \leq |y| \end{cases}$$

$$r_{34}^{sym}(x,y,z) = \begin{cases} z(y-z+1)(-x+1) & \text{for } |x| \geq y \\ z(-x-z+1)(y+1) & \text{for } x \geq |y| \\ z[(y+z+1)(-x+1)-2z] & \text{for } |x| \leq y \\ z[(-x+z+1)(y+1)-2z] & \text{for } x \leq |y|. \end{cases}$$

Theorem *3.3.1*

The basis functions $r_0^{sym},\ldots,r_4^{sym},r_{01}^{sym},\ldots,r_{34}^{sym}$ *satisfy the following conditions:*

1. $r_i^{sym}(A_j) = \delta_{ij}$, $i, j \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$.
2. Each basis function is biquadratic on the base of K (contained in the plane $z = 0$).
3. Each basis function is quadratic on all triangular faces of K.
4. Each basis function is continuous on the interelement boundaries of K (contained in the planes $x = y$ and $x = -y$).
5. The sum of the basis functions is unity at any point in the pyramid (i.e., $\sum r_i^{sym}(x,y,z) \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$).
6. The basis functions r_i^{sym}, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$ vanish on all faces not containing node i.

Proof :

The proof is an immediate consequence of (3.3.1), (3.3.2), and Lemma 3.3.1.

Case II Basis Functions:

We now examine an alternate set of basis functions defined on pyramid element K. These function are of the form

$$s_0(x,y,z) = \begin{cases} \frac{1}{4}(x+z-1)(y-z-1)(x+z)(y-z) & \text{for } x > y \\ \frac{1}{4}(y+z-1)(x-z-1)(y+z)(x-z) & \text{for } x \leq y \end{cases}$$

$$s_1(x,y,z) = \begin{cases} \frac{-1}{4}(x+z)(y-z)((x+z+1)(-y+z+1)-4z)-z(x-y) & \text{for } x > y \\ \frac{-1}{4}(x-z)(y+z)(x-z+1)(-y-z+1) & \text{for } x \leq y \end{cases}$$

$$s_2(x,y,z) = \begin{cases} \frac{1}{4}(y-z+1)(x+z+1)(x+z)(y-z) & \text{for } x > y \\ \frac{1}{4}(x-z+1)(y+z+1)(x-z)(y+z) & \text{for } x \leq y \end{cases}$$

$$s_3(x,y,z) = \begin{cases} \frac{1}{4}(x+z)(y-z)(x+z-1)(y-z+1) & \text{for } x > y \\ \frac{1}{4}(x-z)(y+z)((x-z-1)(y+z+1)+4z)+z(x-y) & \text{for } x \leq y \end{cases}$$

$$s_4(x,y,z) = z(2z-1)$$

$$s_{01}(x,y,z) = \begin{cases} \frac{-1}{2}(x+z-1)(((y-z-1)(x+1)y-z)+z(2x+1)) & \text{for } x > y \\ \frac{-1}{2}(x-z+1)(y+z-1)(x-1)y & \text{for } x \le y \end{cases}$$

$$s_{12}(x,y,z) = \begin{cases} \frac{-1}{2}(y-z+1)(((x+z+1)(y-1)x-z)+z(2y+1)) & \text{for } x > y \\ \frac{-1}{2}(x-z+1)(y+z-1)(y+1)x & \text{for } x \le y \end{cases} \qquad (3.3.4)$$

$$s_{23}(x,y,z) = \begin{cases} \frac{-1}{2}(y-z+1)(x+z-1)(x+1)y & \text{for } x > y \\ \frac{-1}{2}(x-z+1)(((y+z+1)(x-1)y-z)+z(2x+1)) & \text{for } x \le y \end{cases}$$

$$s_{03}(x,y,z) = \begin{cases} \frac{-1}{2}(y-z+1)(x+z-1)(y-1)x & \text{for } x > y \\ \frac{-1}{2}(y+z-1)(((x-z-1)(y+1)x-z)+z(2y+1)) & \text{for } x \le y \end{cases}$$

$$s_{02}(x,y,z) = \begin{cases} (y-z+1)(x+z-1)((y-1)(x+1)+z(x-y+z+1)) & \text{for } x > y \\ (x-z+1)(y+z-1)((y+1)(x-1)-z(x-z-y-1)) & \text{for } x \le y \end{cases}$$

$$s_{04}(x,y,z) = \begin{cases} z(y-z-1)(x+z-1) & \text{for } x > y \\ z(x-z-1)(y+z-1) & \text{for } x \le y \end{cases}$$

$$s_{14}(x,y,z) = \begin{cases} -z((x+z+1)(y-z-1)+4z) & \text{for } x > y \\ -z(x-z+1)(y+z-1) & \text{for } x \le y \end{cases}$$

$$s_{24}(x,y,z) = \begin{cases} z(y-z+1)(x+z+1) & \text{for } x > y \\ z(x-z+1)(y+z+1) & \text{for } x \le y \end{cases}$$

$$s_{34}(x,y,z) = \begin{cases} -z(y-z+1)(x+z-1) & \text{for } x > y \\ -z((y+z+1)(x-z-1)+4z) & \text{for } x \le y \end{cases}$$

Lemma *3.3.2*

The basis functions $s_0,\ldots,s_4,s_{01},\ldots,s_{34}$ *satisfy the following conditions:*

1. $s_i(A_j) = \delta_{ij}$, $i,j \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$.
2. Each basis function is biquadratic on the base of K (contained in the plane $z = 0$).
3. Each basis function is quadratic on all triangular faces of K.
4. Each basis function is continuous on the interelement boundary of K (contained in the plane $x = y$).
5. The sum of the basis functions is unity (i.e., $\sum s_i(x,y,z) \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$).
6. The basis functions s_i, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$ vanish on all faces not containing node i.

Proof :

The method of the proof is identical to that for Lemma 3.3.1.

Following the same procedure as in Case I, we can develop a more symmetric element by taking the averages of the basis functions s_i, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$, with their mirror images. For the sake of simplicity, we shall omit the redundant information and only consider those functions that differ from Case I above

$$s_1^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(z-y)(x(x(1-y+z)+1-y)+z(z(z-1-y)-x))-\frac{1}{2}z(x-y) & \text{for } |x| \geq y \\ \frac{1}{4}(x+z)(y(y(z+x+1)-x-1)+z(z(z-1+x)+y))-\frac{1}{2}z(x-y) & \text{for } x \geq |y| \\ \frac{1}{4}(x+x^2+z^2)(y+z-1)(y+z) & \text{for } |x| \leq y \\ \frac{1}{4}(z-x-1)(z-x)(y^2-y+z^2) & \text{for } x \leq |y| \end{cases}$$

$$s_3^{sym}(x,y,z) = \begin{cases} \frac{1}{4}(x^2-x+z^2)(z-y-1)(z-y) & \text{for } |x| \geq y \\ \frac{1}{4}(x+z-1)(x+z)(y+y^2+z^2) & \text{for } x \geq |y| \\ \frac{1}{4}(y+z)(x(x(y+z+1)-y-1)+z(z(z-1+y)+x))+\frac{1}{2}z(x-y) & \text{for } |x| \leq y \\ \frac{1}{4}(z-x)(y(y(z+1-x)+1-x)+z(z(z-1-x)-y))+\frac{1}{2}z(x-y) & \text{for } x \leq |y| \end{cases}$$

$$s_{01}^{sym}(x,y,z) = \begin{cases} \frac{1}{2}(y(y-1+z(z-y))+x(x(y-2z+y(z-y)))) & \text{for } |x| \geq y \\ -\frac{1}{2}(x+z-1)(xz+y(y(x+1)-x-1)) & \text{for } x \geq |y| \\ -\frac{1}{2}y(x^2+z-1)(y+z-1) & \text{for } |x| \leq y \\ \frac{1}{2}(z-x-1)(xz+y(1-x+y(x-1))) & \text{for } x \leq |y| \end{cases}$$

$$s_{12}^{sym}(x,y,z) = \begin{cases} \frac{1}{2}(y-z+1)(x(1-y+x(1-y))-yz) & \text{for } |x| \geq y \\ -\frac{1}{2}(x(-x-1+z(z+x))+y(y(2z+x(x+z+1)))) & \text{for } x \geq |y| \\ -\frac{1}{2}(y+z-1)(x(x+xy+1+y)+yz) & \text{for } |x| \leq y \\ \frac{1}{2}x(z+y^2-1)(z-x-1) & \text{for } x \leq |y| \end{cases}$$

$$s_{23}^{sym}(x,y,z) = \begin{cases} \frac{1}{2}y(z+x^2-1)(z-y-1) & \text{for } |x| \geq y \\ -\frac{1}{2}(x+z-1)(y(y(x+1)+1+x)+xz) & \text{for } x \geq |y| \\ -\frac{1}{2}(x(x((z+y+1)y+2z))+y(z(z+y)-y-1)) & \text{for } |x| \leq y \\ \frac{1}{2}(z-x-1)(y(y(x-1)+x-1)+xz) & \text{for } x \leq |y| \end{cases}$$

$$s_{03}^{sym}(x,y,z) = \begin{cases} \frac{1}{2}(y-z+1)(x(x(1-y)+y-1)-yz) & \text{for } |x| \geq y \\ -\frac{1}{2}x(z+y^2-1)(x+z-1) & \text{for } x \geq |y| \\ -\frac{1}{2}(y+z-1)(yz+x(x+xy-y-1)) & \text{for } |x| \leq y \\ \frac{1}{2}(x(x(1-y)(1+y)-1+z(z-x))+y(y(z(x-2)+x))) & \text{for } x \leq |y| \end{cases}$$

$$s_{02}^{sym}(x,y,z) = \begin{cases} (z-y-1)(y-z(z(z-y)-2+2y)+x(x(1-y-z))-1) & \text{for } |x| \geq y \\ (x+z-1)(z(2-z(z+x)+2x)+y(y(1+x-z))-1-x) & \text{for } x \geq |y| \\ (y+z-1)(z(2-z(z+y)+2y)+x(x(1+y-z))-1-y) & \text{for } |x| \leq y \\ (z-x-1)(x+z(2-2x-z(z-x))+y(y(1-x-z))-1) & \text{for } x \leq |y| \end{cases}$$

For the remaining cases we have

$$s_0^{sym} = r_0^{sym}$$
$$s_2^{sym} = r_2^{sym}$$

$$s_4^{sym} = r_4^{sym}$$
$$s_{04}^{sym} = r_{04}^{sym}$$
$$s_{14}^{sym} = r_{14}^{sym}$$
$$s_{24}^{sym} = r_{24}^{sym}$$
$$s_{34}^{sym} = r_{34}^{sym}.$$

Theorem *3.3.2*

The basis functions $s_0^{sym}, \ldots, s_4^{sym}, s_{01}^{sym}, \ldots, s_{34}^{sym}$ *satisfy the following conditions:*

1. $s_i^{sym}(A_j) = \delta_{ij}$, $i,j \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$.
2. Each basis function is biquadratic on the base of K (contained in the plane $z = 0$).
3. Each basis function is quadratic on all triangular faces of K.
4. Each basis function is continuous on the interelement boundaries of K (contained in the planes $x = y$ and $x = -y$).
5. The sum of the basis functions is unity (i.e., $\sum s_i^{sym}(x,y,z) \equiv 1$, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$).
6. The basis functions s_i^{sym}, $i \in \{0,1,2,3,4,01,12,23,03,02,04,14,24,34\}$ vanish on all faces not containing node i.

Proof :

The proof is an immediate consequence of the mirror image mappings and Lemma 3.3.2.

4. Software Implementation and Numerical Experiments

Numerical experiments were carried out to test the accuracy of all the pyramidal elements presented in the previous section. In order to conduct the tests, software was developed to discretize the domain with a pyramidal mesh, calculate the system stiffness matrix and load vector, and solve the resulting linear system of equations. Below, we describe the procedure of the experiments and the development of the software by breaking the process down into a natural subdivision of tasks. We note that, in general, all software for the finite element method would follow a similar process and can be broken in to similar subprocesses.

4.1 FEM Software Development

Mesh Construction

The first step is to discretize the domain Ω. In our case, since we want to measure the accuracy of the elements, we desire a simple domain so we can compare the approximate solution u_h to a known solution u. Thus, we choose as our domain $\Omega = (0,1) \times (0,1) \times (0,1)$, and in all cases the experiment conducted was to solve Poisson's equation with Dirichlet boundary conditions:

$$\begin{aligned} -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) &= f(x,y,z) \qquad (x,y,z) \in \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega \end{aligned} \tag{4.1.1}$$

where $\Omega = (0,1) \times (0,1) \times (0,1)$. The true solution chosen for the experiments was:

$$u = \sin(\pi x)\sin(2\pi y)\sin(3\pi z).$$

In order to discretize Ω by a three dimensional pyramidal mesh we first construct a mesh consisting of cubes, with an equal number of cubes along each axis. The number of cubes along an axis, N, is an input parameter so that we can control grid "fineness" or "coarseness" of the mesh. Each of the N^3 cubes is then divided into six equal sized pyramidal elements (Figure 4.1).

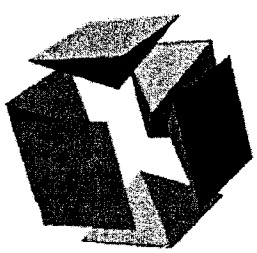


Figure 4.1.
Cube decomposed into six pyramids.

Thus, we have a system consisting of $6N^3$ pyramidal elements. The number of nodes in the system is then determined depending on the type of pyramidal element being used. During the mesh construction we also calculate and store the global coordinate locations of all the nodes. Note that although each element has 5, 13, or 14 nodes depending on the element type, many of the nodes are common to multiple elements. In order to reduce the storage space needed for the mesh, we only store a node and its coordinates once. With each element we only store references to the nodes that make up its construction. To accomplish this we use two arrays, the first one, *NodeInfo*, contains the nodal x,y,z coordinates in three dimensional space, and the second, *ElemInfo*, an array of elements, contains appropriate references into the array of nodes. Before we can do this however, we need to determine a convenient enumeration of elements and nodes for the overall mesh, as well as a local enumeration of the nodes for an element. Figure 4.2 shows an example of the global node numbering used for the linear 5-node pyramidal elements and a mesh parameter of $N = 2$. (Note that pyramid boundary lines have been removed in order to present a clearer view of the nodes.)

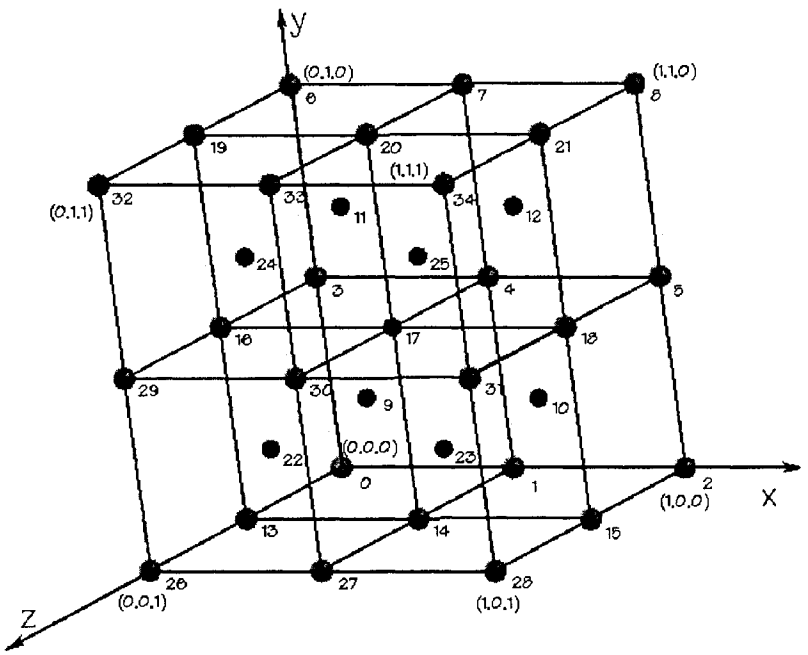


Figure 4.2.
Global node numbering for linear case with $N = 2$.

The associated nodal array structure for this case is then:

NodeInfo =

Node:	0	1	2	...	34
Coordinates:	(0,0,0)	(0.5,0,0)	(1,0,0)	...	(1,1,1)

We adopt a similar numbering scheme for the elements; however we treat the six elements of a cube as a group, counting them together before enumerating the next group. For example, for the mesh in Figure 4.2 we have a total of eight cubes and 48 pyramids, with the pyramids

numbered 0-47. The first cube is the cube with corners $(0,0,0)$ and $(0.5,0.5,0.5)$ and it consists of pyramids $0-5$, the second cube has corners at $(0.5,0,0)$ and $(1,0.5,0.5)$ and consists of pyramids $6-11$, the 3rd cube has corners at $(0.5,0,0)$ and $(0.5,1,0.5)$ and consists of pyramids $12-17$,... the last cube has corners at $(0.5,0.5,0.5)$, $(1,1,1)$ and consists of pyramids $42-47$. Within each cube the numbering sequence of the pyramids is as follows: the first pyramid is the pyramid with its base on the base of the cube (the $y = 0$ plane in Figure 4.2), the second is the pyramid with its base on the right face of the cube (i.e., a 90^o counter-clockwise rotation from the first element), the third is the pyramid with its base on the top face of the cube (i.e., a 90^o counter-clockwise rotation from the previous element), the fourth has its base on the left face of the cube (i.e., another 90^o counter-clockwise rotation from the previous element), the fifth is the pyramid with its base on the front face of the cube, and the final sixth pyramid is the one having as its base the back face of the cube. Lastly, the node numbering sequence local to an element will follow that given in the previous section's discussion. Thus, the element array structure associated with the mesh in Figure 4.2 is:

$ElemInfo =$

Element:	0	1	2	3	4	5	...	47
Node 0:	13	14	17	16	16	0		17
Node 1:	14	17	16	13	17	1		18
Node 2:	1	4	3	0	14	4		21
Node 3:	0	1	4	3	13	3		20
Node 4:	9	9	9	9	9	9	...	25

Clearly, we can now use the entries in the ElemInfo array as references into the NodeInfo array to obtain the global coordinates of the nodes for a given element. In other words, we can find the coordinates of the j-th node for a given element i using the relation:

$$NodeInfo\big[ElemInfo[j,i]\big].$$

During mesh construction we can also store other relevant information associated with individual nodes or elements. For instance, we can indicate whether or not a node is a boundary node with a simple modification to the *NodeInfo* array as follows:

$NodeInfo =$

Node:	0	1	...	9	...	34
Coordinates:	$(0,0,0)$	$(0.5,0,0)$		$(0.25,0.25,0.25)$		$(1,1,1)$
Node Type:	Boundary	Boundary	...	Internal	...	Boundary

Note that, in our case, since the boundary conditions are the same on all boundaries of the domain, we need only to mark a node as being on the boundary or not being on the boundary; we do not need to store information indicating to which boundary the node belongs. Thus, we would actually require only one additional bit of storage per node to store this information. Although this information could always be calculated based on the node's coordinates, using a small amount of additional storage such as this will save on tedious and unnecessary computations later. We do point out, however, that the storage space required for even simple structures such as these can become prohibitive for even moderate values of N. As stated earlier, since we are working in three dimensional space, the number of pyramidal elements in the mesh is $6N^3$. The number of nodes in the mesh also grows at a $O(N^3)$ rate. For example, in the linear case, with five nodes per element, we have a total of $(N+1)^3 + N^3$ nodes, since

we only store a node once. Thus, a mesh with a seemingly small N value such as $N = 32$ will yield a mesh with almost 200,000 elements, and over 68,000 nodes!

Generation of Stiffness Matrix and Load Vector

Once the domain has been discretized and the mesh has been constructed, the system stiffness matrix and system load vector can be generated. In order to reduce the space required to store these entities we start by removing boundary nodes from consideration for the time being. Since their values are known we do not need to include them in the system to be solved; their values can simply be placed back into the solution vector at a later time. We also note that, in order to save storage space, the stiffness matrix is stored in a sparse format. Let us first examine the stiffness matrix generation process. To generate the stiffness matrix, A, each element in the system is checked in turn, and pairs of nodes that are part of that element are compared. If both nodes are internal to the system (where boundary conditions do not apply) the value of the integral over the element is approximated, and this value is added to the stiffness matrix entry associated with the two nodes of that element. Using the linear case as an example, this procedure can be summarized as follows:

Let a_{ij}^K represent the system matrix entry associated with the i-th and j-th nodes of element K, and let N be the mesh parameter described above.

Then we check the pairs of nodes associated with each element in the system by looping over all the elements:

```
for K = 1 to 6N^3 (number of elements in system)
  for i = 1 to 5 (number of nodes per element)
    for j = 1 to 5 (number of nodes per element)
      if both node i and node j of element K are not
                boundary nodes then
```

$$a_{ij}^K = a_{ij}^K + \int_K \left(\frac{\partial \psi_i}{\partial x} \cdot \frac{\partial \psi_j}{\partial x} + \frac{\partial \psi_i}{\partial y} \cdot \frac{\partial \psi_j}{\partial y} + \frac{\partial \psi_i}{\partial z} \cdot \frac{\partial \psi_j}{\partial z} \right) dxdydz \qquad (4.1.2)$$

```
      end-if
    end-for
  end-for
end-for
```

where ψ_m, $m = 1,\ldots,5$ are the basis functions of the pyramidal element, and numerical cubature formulae are employed in order to evaluate the integral over the element. We will discuss how the numerical integration is implemented in a moment, but first we examine how the load vector is generated.

The procedure for calculating the entries of the load vector involves looping over all the elements in the system and checking the nodes of each element. If the node is a boundary node then we ignore it for the time being; if, however, the node is an internal node then we calculate the value of the integral over the element and add it to the associated entry in the load vector. Continuing with the linear case as an example, this procedure can be summarized as follows:

Let b_i^K be the load vector entry associated with the i-th node of element K, and again let N

be the mesh parameter. Then we have

```
for K = 1 to 6N^3 (number of elements in system)
  for i = 1 to 5 (number of nodes per element)
    if node i of element K is not a boundary node then
      b_i^K = b_i^K + ∫_K f ψ_i dxdydz                (4.1.3)
    end-if
  end-for
end-for
```

where ψ_m, $m = 1,\ldots,5$ are the basis functions of the pyramidal element and f is the function from (4.1.1).

In order to integrate over the pyramidal elements, each pyramidal element is first divided into two or four tetrahedra depending on whether we are applying the extra symmetries to the basis functions for the element (Section 3.1-3.4). Once the pyramid is divided into tetrahedra we use Gaussian cubature formulae for tetrahedra [Keast, 1986], [Maeztu, Maza, 1995], [Cools, and Rabinowitz, 1993], [Cools, 1999], [Stroud, 1971] to approximate the values of the integrals. The values of the integrals for each tetrahedron are then summed to determine the integral over the composite pyramidal element. However, since the basis functions are given in terms of only a single reference element, and since these functions are defined in terms of local coordinates, we must carry out a transformation before approximating the integrals.

Coordinate Transformations

Since the points of integration as well as the basis functions themselves are defined in terms of coordinates local to the element, we can define a method of mapping between the global nodal coordinates and the local element coordinates. Let (ξ,η,ζ) represent a point in the local element coordinates. In particular, we will see that these points are associated with the integration. We then let (x,y,z) represent a point in the global system. Fortunately the basis functions themselves provide us with a convenient method of establishing a coordinate transformation. For each element we have:

$$\begin{aligned} x &= \psi_1 x_1 + \psi_2 x_2 + \ldots + \psi_m x_m = \boldsymbol{\psi}^T \mathbf{x} \\ y &= \psi_1 y_1 + \psi_2 y_2 + \ldots + \psi_m y_m = \boldsymbol{\psi}^T \mathbf{y} \\ z &= \psi_1 z_1 + \psi_2 z_2 + \ldots + \psi_m z_m = \boldsymbol{\psi}^T \mathbf{z} \end{aligned} \qquad (4.1.4)$$

where $\boldsymbol{\psi} = \left[\psi_1\ \psi_2 \ldots \psi_m\right]^T$, $i = 1,2,\ldots,m$ are the element basis functions given in terms of the local coordinates (ξ,η,ζ), and $\mathbf{x} = \left[x_1\, x_2 \ldots x_m\right]^T$, $\mathbf{y} = \left[y_1\, y_2 \ldots y_m\right]^T$, $\mathbf{z} = \left[z_1\, z_2 \ldots z_m\right]^T$ are the global coordinates of the nodes for the element. In the integration of (4.1.2) we are concerned with partial derivatives, so we consider the following relations:

$$\begin{bmatrix} \frac{\partial \psi_i}{\partial \xi} \\ \frac{\partial \psi_i}{\partial \eta} \\ \frac{\partial \psi_i}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial \psi_i}{\partial x} \\ \frac{\partial \psi_i}{\partial y} \\ \frac{\partial \psi_i}{\partial z} \end{bmatrix} = J \begin{bmatrix} \frac{\partial \psi_i}{\partial x} \\ \frac{\partial \psi_i}{\partial y} \\ \frac{\partial \psi_i}{\partial z} \end{bmatrix} \qquad (4.1.5)$$

In this equation, the left-hand side can be evaluated as the functions ψ_i are specified in local coordinates. The matrix J is the Jacobian matrix. Thus, in order to find the global derivatives we invert J to get

$$\begin{bmatrix} \frac{\partial \psi_i}{\partial x} \\ \frac{\partial \psi_i}{\partial y} \\ \frac{\partial \psi_i}{\partial z} \end{bmatrix} = J^{-1} \begin{bmatrix} \frac{\partial \psi_i}{\partial \xi} \\ \frac{\partial \psi_i}{\partial \eta} \\ \frac{\partial \psi_i}{\partial \zeta} \end{bmatrix}$$

In terms of the shape functions defining the coordinate transformation ψ we have

$$J = \begin{bmatrix} \sum \frac{\partial \psi_i}{\partial \xi} x_i & \sum \frac{\partial \psi_i}{\partial \xi} y_i & \sum \frac{\partial \psi_i}{\partial \xi} z_i \\ \sum \frac{\partial \psi_i}{\partial \eta} x_i & \sum \frac{\partial \psi_i}{\partial \eta} y_i & \sum \frac{\partial \psi_i}{\partial \eta} z_i \\ \sum \frac{\partial \psi_i}{\partial \zeta} x_i & \sum \frac{\partial \psi_i}{\partial \zeta} y_i & \sum \frac{\partial \psi_i}{\partial \zeta} z_i \end{bmatrix} \qquad (4.1.6)$$

$$= \begin{bmatrix} \frac{\partial \psi_1}{\partial \xi} & \frac{\partial \psi_2}{\partial \xi} & \cdots & \frac{\partial \psi_m}{\partial \xi} \\ \frac{\partial \psi_1}{\partial \eta} & \frac{\partial \psi_2}{\partial \eta} & \cdots & \frac{\partial \psi_m}{\partial \eta} \\ \frac{\partial \psi_1}{\partial \zeta} & \frac{\partial \psi_2}{\partial \zeta} & \cdots & \frac{\partial \psi_m}{\partial \zeta} \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_m & y_m & z_m \end{bmatrix}$$

To transform the variables and the region with respect to which the integration is made, a standard process will be used which involves the determinant of J. Thus, for instance a volume element becomes

$$dxdydz = \det J \, d\xi d\eta d\zeta.$$

This type of transformation is valid irrespective of the number of coordinates used. For a more detailed examination of coordinate transformations and the use of reference elements in the finite element method please refer to [Zienkiewicz, 1989, pp.181-191], or [Kardestuncer, 1987, pp.1.139-1.152].

Numerical Integration

Gaussian cubature formulae for tetrahedra are used to evaluate the above integrals. An in-depth discussion of numerical integration methods is beyond the scope of this work. For more information please refer to texts on numerical integration [Stroud, 1971], [Stroud, 1974], the finite element method [Zienkiewicz, 1989], [Kardestuncer, 1987], or general texts on numerical methods or scientific computing [Rao, 2002], [Heath, 2002]. The general idea behind Gaussian quadrature is to approximate the integration of a function over a domain by summation. At each step of the summation the function is evaluated at given points, the Gaussian points, and the results are weighted by given values, the Gaussian weights. In other words,

$$\int f(x,y,z)dxdydz \approx \sum_{i=1}^{Q} w_i f(\xi,\eta,\zeta)_i,$$

where,

w_i is the weight to be used for point i,

$(\xi,\eta,\zeta)_i$ is the point given by the quadrature formula being used,

Q is the number of points to be used in the given quadrature formula.

Based on this and the above discussion on coordinate transformations, we may now proceed and evaluate the integrals (4.1.2) and (4.1.3).

For stiffness matrix integrals we proceed as follows. Firstly, we take partial derivatives of the basis functions for the particular element in question. For example, in the 5-node pyramid case, before applying the symmetries (3.1.4) and using as our variables (ξ,η,ζ), we have:

$$\frac{\partial \psi_0}{\partial \xi} = \frac{\partial p_0(\xi,\eta,\zeta)}{\partial \xi} = \begin{cases} -\frac{1}{4}(\zeta-\eta+1) & \text{for } x > y \\ \frac{1}{4}(\eta+\zeta-1) & \text{for } x \le y \end{cases},$$

$$\frac{\partial \psi_0}{\partial \eta} = \frac{\partial p_0(\xi,\eta,\zeta)}{\partial \eta} = \begin{cases} \frac{1}{4}(\xi+\zeta-1) & \text{for } x > y \\ -\frac{1}{4}(\zeta-\xi+1) & \text{for } x \le y \end{cases},$$

$$\frac{\partial \psi_0}{\partial \zeta} = \frac{\partial p_0(\xi,\eta,\zeta)}{\partial \zeta} = \begin{cases} -\frac{1}{4}(\xi-\eta+2\zeta) & \text{for } x > y \\ -\frac{1}{4}(\eta-\xi+2\zeta) & \text{for } x \le y \end{cases},$$

$$\frac{\partial \psi_1}{\partial \xi} = \frac{\partial p_1(\xi,\eta,\zeta)}{\partial \xi} = \begin{cases} \frac{1}{4}(\zeta-\eta+1) & \text{for } x > y \\ -\frac{1}{4}(\eta+\zeta-1) & \text{for } x \le y \end{cases},$$

$$\vdots \qquad\qquad \vdots$$

$$\frac{\partial \psi_4}{\partial \zeta} = \frac{\partial p_1(\xi,\eta,\zeta)}{\partial \zeta} = 1.$$

For each quadrature point $(\xi,\eta,\zeta)_i$ we evaluate the partial derivatives to obtain the matrix

$$\begin{bmatrix} \frac{\partial \psi_0}{\partial \xi}(\xi,\eta,\zeta)_i & \frac{\partial \psi_1}{\partial \xi}(\xi,\eta,\zeta)_i & \cdots & \frac{\partial \psi_4}{\partial \xi}(\xi,\eta,\zeta)_i \\ \frac{\partial \psi_0}{\partial \eta}(\xi,\eta,\zeta)_i & \frac{\partial \psi_1}{\partial \eta}(\xi,\eta,\zeta)_i & \cdots & \frac{\partial \psi_4}{\partial \eta}(\xi,\eta,\zeta)_i \\ \frac{\partial \psi_0}{\partial \zeta}(\xi,\eta,\zeta)_i & \frac{\partial \psi_1}{\partial \zeta}(\xi,\eta,\zeta)_i & \cdots & \frac{\partial \psi_4}{\partial \zeta}(\xi,\eta,\zeta)_i \end{bmatrix}. \qquad (4.1.7)$$

We then compute the Jacobian matrix using (4.1.6). Finally, we extract the i-th and j-th columns of (4.1.7) and multiply them by the inverse of the Jacobian in order to obtain the derivatives in (4.1.2). In other words we calculate the integral of our pyramidal element as follows:

$$\int_K \left(\frac{\partial \psi_i}{\partial x} \cdot \frac{\partial \psi_j}{\partial x} + \frac{\partial \psi_i}{\partial y} \cdot \frac{\partial \psi_j}{\partial y} + \frac{\partial \psi_i}{\partial z} \cdot \frac{\partial \psi_j}{\partial z} \right) dxdydz, \approx$$

$$\sum_{t=1}^{T} \sum_{q=1}^{Q} w_q \left(\frac{\partial \psi_i}{\partial x}(\alpha_q) \cdot \frac{\partial \psi_j}{\partial x}(\alpha_q) + \frac{\partial \psi_i}{\partial y}(\alpha_q) \cdot \frac{\partial \psi_j}{\partial y}(\alpha_q) + \frac{\partial \psi_i}{\partial z}(\alpha_q) \cdot \frac{\partial \psi_j}{\partial z}(\alpha_q) \right) \det J,$$

where,

T is the number of tetrahedra that make up our pyramidal element (two before applying symmetries, four after),

Q is the number of cubature points for the given tetrahedra cubature formula,

w_q is the weight to be used for cubature point q,

$\alpha_q = (\xi,\eta,\zeta)_q$ is the q-th cubature point for the given tetrahedra cubature formula,

$\frac{\partial \psi_i}{\partial x}, \frac{\partial \psi_i}{\partial y}, \frac{\partial \psi_i}{\partial z}, etc.$, are found by extracting the i-th and j-th columns of (4.1.7) and multiplying them by the inverse of the Jacobian, i.e.,

$$\begin{bmatrix} \frac{\partial \psi_i}{\partial x} \\ \frac{\partial \psi_i}{\partial y} \\ \frac{\partial \psi_i}{\partial z} \end{bmatrix} = J^{-1} \begin{bmatrix} \frac{\partial \psi_i}{\partial \xi}(\xi,\eta,\zeta)_q \\ \frac{\partial \psi_i}{\partial \eta}(\xi,\eta,\zeta)_q \\ \frac{\partial \psi_i}{\partial \zeta}(\xi,\eta,\zeta)_q \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial \psi_j}{\partial x} \\ \frac{\partial \psi_j}{\partial y} \\ \frac{\partial \psi_j}{\partial z} \end{bmatrix} = J^{-1} \begin{bmatrix} \frac{\partial \psi_j}{\partial \xi}(\xi,\eta,\zeta)_q \\ \frac{\partial \psi_j}{\partial \eta}(\xi,\eta,\zeta)_q \\ \frac{\partial \psi_j}{\partial \zeta}(\xi,\eta,\zeta)_q \end{bmatrix}.$$

Similarly, for the load vector entries (4.2.3) we have

$$\int_K f\psi_i \, dxdydz \approx$$

$$\sum_{t=1}^{T}\sum_{q=1}^{Q} w_q\left(\psi_i(\xi,\eta,\zeta)_q f(x,y,z)\right)$$

where,

T is the number of tetrahedra that make up our pyramidal element (two before applying symmetries, four after),

Q is the number of cubature points for the given tetrahedra cubature formula,

w_q is the weight to be used for cubature point q,

$(\xi,\eta,\zeta)_q$ is the q-th cubature point for the given tetrahedra cubature formula,

(x,y,z) are determined by the relation (4.1.4).

Solution of System

Once the system stiffness matrix and load vector have been generated, we may proceed to solving the linear system of equations. Three different solvers were implemented in the software used for the experiments. Firstly, Gaussian elimination was chosen, since it generates the most accurate results and is generally the most reliable solution method. Secondly, Gauss-Seidel was implemented in order to increase efficiency. Lastly, to further reduce computational costs for larger test cases, the conjugate gradient method with preconditioning was employed to solve the system. In this case, incomplete Cholesky factorization was used as the preconditioner.

4.2 Computational Results

Three different sets of experiments were performed, corresponding to the three basic types of pyramidal elements presented in section 3. For each type of element, experiments were performed to check the accuracy of the elements by varying the mesh parameter N, and by testing cases before and after the mirror image mappings were made to gain more symmetry.

In the five node element case, experiments were conducted for values of N ranging from 4 to 128. Table 4.1 summarizes the results for the basis functions both before and after the mirror image mappings were made. For each value of N, the table lists values for (i) $h = 1/N$; (ii) the L_2-norms of $u - \hat{u}_h$ and $u - u_h$, where u is the true solution, $\hat{u}_h$ is the finite element solution corresponding to Wieners' transformed basis functions (3.1.2) and u_h is the finite element solution corresponding to the new symmetric basis functions (3.1.5). From Table 4.1, we see that the symmetrized elements yield a better result than the non-symmetrized elements, even though the number of degrees of freedom is the same. We also note that in both cases the element achieves the optimal approximation order of $O(h^2)$.

$h = 1/N$	$\|u - \hat{u}_h\|_0$	$\frac{\|u-\hat{u}_{2h}\|_0}{\|u-\hat{u}_h\|_0}$	$\|u - u_h\|_0$	$\frac{\|u-u_{2h}\|_0}{\|u-u_h\|_0}$
1/4	6.10e-2	None	5.83e-2	None
1/8	1.62e-2	3.775	1.51e-2	3.871
1/16	4.08e-3	3.966	3.78e-3	3.987
1/32	1.02e-3	3.992	9.45e-4	3.997
1/64	2.55e-4	3.998	2.36e-4	3.999
1/128	6.39e-5	3.999	5.91e-5	4.000

Table 4.1. Test results for 5-node pyramidal element before and after symmetries are applied.

Similar experiments were conducted for the thirteen and fourteen node elements presented in sections 3.2 and 3.3. The results for the 13-node pyramidal element are summarized in Table 4.2. In this case we see a substantial improvement over the non-symmetric basis functions. The discretization error is almost 3.5 times better on the finest mesh where $h = 1/64$. We note that the rate of convergence for the non-symmetric basis functions is still of order $O(h^2)$, but for the new symmetric basis functions the rate of convergence is in the order higher than $O(h^2)$, though it is less than the order of $O(h^3)$ that one might expect.

$h = 1/N$	$\|u - \hat{u}_h\|_0$	$\frac{\|u-\hat{u}_{2h}\|_0}{\|u-\hat{u}_h\|_0}$	$\|u - u_h\|_0$	$\frac{\|u-u_{2h}\|_0}{\|u-u_h\|_0}$
1/4	9.80e-3	None	8.68e-3	None
1/8	1.49e-3	6.571	1.08e-3	8.074
1/16	2.93e-4	5.093	1.48e-4	7.251
1/32	6.72e-5	4.356	2.37e-5	6.249
1/64	1.64e-5	4.097	4.73e-6	5.016

Table 4.2. Test results for 13-node pyramidal element, before and after symmetries are applied.

Tests on the 14-node pyramidal element were conducted using the two sets of basis functions presented in Section 3.3 (Case I and Case II). The results of these experiments are summarized in Tables 4.3 and 4.4. We note that the Case II basis functions give an improvement over the Case I functions, with a discretization error that is over twice as good on the finest mesh with $h = 1/64$. Since the basis functions (3.3.1) and (3.3.4) for the 14-node element were carefully developed to yield highly accurate results, the application of the extra symmetries results in a less dramatic improvement than was found for the previous element cases.

$h = 1/N$	$\|u - \hat{u}_h\|_0$	$\frac{\|u-\hat{u}_{2h}\|_0}{\|u-\hat{u}_h\|_0}$	$\|u - u_h\|_0$	$\frac{\|u-u_{2h}\|_0}{\|u-u_h\|_0}$
1/4	3.42e-3	None	3.07e-3	None
1/8	8.51e-4	4.019	7.62e-04	4.029
1/16	2.13e-4	3.995	1.92e-4	3.969
1/32	5.72e-5	3.724	5.02e-5	3.825
1/64	1.37e-5	4.175	1.20e-5	4.183

Table 4.3. Test results for 14-node pyramidal element with Case I basis functions, before and after symmetries are applied.

$h = 1/N$	$\|u - \hat{u}_h\|_0$	$\frac{\|u-\hat{u}_{2h}\|_0}{\|u-\hat{u}_h\|_0}$	$\|u - u_h\|_0$	$\frac{\|u-u_{2h}\|_0}{\|u-u_h\|_0}$
1/4	1.61e-3	None	1.58e-3	None
1/8	3.77e-4	4.278	3.69e-4	4.273
1/16	9.23e-5	4.086	9.04e-5	4.084
1/32	2.30e-5	4.013	2.25e-5	4.018
1/64	5.73e-6	4.014	5.61e-6	4.011

Table 4.4. Test results for 14-node pyramidal element with Case II basis functions, before and after symmetries are applied.

From Tables 4.3 and 4.4, we see that the rate of convergence for the symmetric and non-symmetric basis functions for both Case I and Case II is of order $O(h^2)$, again less than the order $O(h^3)$ that one might expect. Since the application of the extra symmetries requires a non-trivial amount of additional computation, effectively doubling the number of basis function evaluations needed, we must also consider the efficiency of the method used. From the test results we see that adding symmetries to the Case II basis for the 14-node pyramidal element gives only a slight improvement in accuracy. Thus, we can conclude that for the 14-node element, best overall performance is achieved by the Case II basis without the extra symmetries applied.

5. Conclusion

The finite element method provides us with a very powerful tool for approximating highly complex systems in science and engineering. Developed initially as an engineering tool for structural analysis, the method has since found widespread acceptance in many areas of applied science. The method consists of discretizing a continuous problem over a grid, or mesh, composed of simple geometric shapes, or "elements". The most commonly used elements are simple intervals in the one dimensional case, triangles or quadrilaterals for two dimensional space, and tetrahedra or hexahedra in the case of three dimensions.

When we consider the three dimensional case, we note that both hexahedral and tetrahedral elements have a number of advantages and disadvantages, particularly when it comes to adaptive mesh generation. For instance, repeated anisotropic subdivision of tetrahedral elements can cause serious loss of mesh quality leading to inaccurate solutions. Although hexahedra can be subdivided anisotropically without loss of quality, hexahedral adaptation schemes tend to generate "hanging" vertices when a hexahedron cannot be split into smaller hexahedra without continuously propagating the mesh refinement into regions where it is not desired. One way to overcome these deficiencies is to construct a mesh which consists of both hexahedral and tetrahedral elements. A mesh constructed in this manner would use hexahedra to fill in geometrically simple regions of the domain where no sharp corners or curves exist, while tetrahedra would be used to fill in the remaining, more geometrically complex, regions where hexahedra refinement is less suitable. Unfortunately, when using both tetrahedra and hexahedra in the same mesh, problems arise at the areas where the two types of elements must be joined together since the element types do not conform. Thus, another type of element is needed to properly connect hexahedral and tetrahedral portions of a mixed finite element mesh. A pyramidal element is an ideal element type for making these types of connections between hexahedra and tetrahedra.

In this work, we have examined new pyramidal mortar elements which are highly symmetric in nature. This high degree of symmetry reduces the artificial anisotropy present in previous pyramidal elements and leads to a pyramidal element which produces superior accuracy. We first studied a five-node pyramidal element suitable for connections between four node linear tetrahedra and eight node bilinear hexahedra. Then we examined a thirteen node element most suited for joining ten node quadratic tetrahedral elements to twenty node hexahedra. Next, we considered the case of a new fourteen node pyramidal element that is ideal for interfacing ten node tetrahedral elements and twenty-seven node hexahedral elements. In all cases the basis functions for these elements and the process of their construction was examined in detail and proof of their theoretical correctness was also given. Finally, computer software was developed and used to conduct numerical experiments which illustrated the improvements these elements offer. As a result of these experiments we can conclude that the new symmetric basis functions always yield a better discretization error than their less symmetric counterparts.

Areas For Further Research

The problem of finding a pyramidal element which is proven to achieve the optimal rate of convergence for the quadratic case, a rate of $O(h^3)$, is an open problem. The development of pyramidal elements with piecewise polynomial basis functions which are cubic, or higher in degree, is another area of possible future study. Additional testing of these elements as interface elements in mixed hexahedral and tetrahedral meshes, under more applied circumstances, could also be conducted in order to compare them further with other interfacing methods.

Bibliography

Aftosmis, M., Gaitonde, D., and Travares, T. S., "On the Accuracy, Stability, and Monotonicity of Various Reconstruction Algorithms for Unstructured Meshes", AIAA-94-0415, In *Proc. of 32nd AIAA Aerospace Sciences Meeting and Exhibit,* 1994.

Benzley, Steven E., Perry, E., Merkley, K., Clark, B., and Sjaardama, G, "A Comparison of All Hexagonal and All Tetrahedral Finite Element Meshes for Elastic and Elasto-plastic Analysis", In *Proc. 4th International Meshing Roundtable*, Albuquerque, New Mexico, pp. 179-192, 1995.

Biswas, Rupak, and Strawn, Roger C., "Mesh Quality Control for Multiply-refined Tetrahedral Grids", *Applied Numerical Mathematics*, Vol. 20, No. 4, pp. 337-348, 1996.

Biswas, Rupak, and Strawn, Roger C., "Tetrahedral and Hexahedral Mesh Adaptation for CFD Problems", *Applied Numerical Mathematics*, Vol. 26, No. 1-2, pp. 135-151, 1998.

Bramble, James H., Zlamal, Milos, "Triangular Elements in the Finite Element Method", *Mathematics of Computation*, Vol. 24, No. 112, pp. 809-820, 1970.

Bretl, John L., "Connecting Solid Finite Element Models That Have Dissimilar Meshes on the Mating Surface", In *Proc. MSC/NASTRAN User Conference*, 1984.

Clough, R. W., "The Finite Element Method in Plane Stress Analysis", *In Proc. 2nd ASCE Conference on Electronic Computation*, Pittsburg, Pennsylvania, Sept. 1960.

Clough, R. W., "The Finite Element Method After Twenty-five Years: A Personal View", Computers and Structures, Vol. 12, No. 4, pp. 361-370, 1980.

Cook, Robert D., Malkus, David S., and Plesha, Michael E. *Concepts and Applications of Finite Element Analysis.* New York: John Wiley & Sons, 1989.

Cools, Ronald, and Rabinowitz, Philip, "Monomial Cubature rules since "Stroud": A Compilation", *Journal of Computational and Applied Mathematics*, Vol. 48, pp. 309-326, 1993.

Cools, Ronald, "Monomial Cubature rules since "Stroud": A Compilation - part 2", *Journal of Computational and Applied Mathematics*, Vol. 112, pp. 21-27, 1999.

Douglas, C. C., "Multigrid methods in science and engineering", *IEEE Computational Science and Engineering*, Vol. 3, No.4, pp. 55-68, 1997.

Duff, Iain S., Grimes, Roger G., and Lewis, G., "Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)", *Tech. Rep. RAL 92-086*, Chilton, Oxon, England, 1992.

Duff, Ians S. "A Review of Frontal Methods for Solving Linear Systems", *Computer Physics Communications*, Vol. 97, No. 1-2, pp. 45-52, 1996.

George, Alan, "Nested Dissection of a Regular Finite Element Mesh", *SIAM Journal of Numerical Analysis*, Vol. 10, No. 2, pp. 345-363, 1973.

Gradinaru, V., and Hiptmair, R., "Whitney Elements on Pyramids", *Electronic Transactions on Numerical Analysis*, Vol. 8, pp. 154-168, 1999.

Heath, M. T. *Scientific Computing: An Introductory Survey*, 2nd ed. Boston: McGraw-Hill, 2002

Henrici, P., "Fast Fourier Methods in Computational Complex Analysis", *SIAM Review*, Vol. 21, pp. 418-527, 1979.

Hlaváček, I., and Křížek, M., "On Exact Results in the Finite Element Method", *Appl. Math.*, 46, pp. 467-478, 2001.

Irons, B.M., "A Frontal Solution Program for Finite Element Analysis", *International Journal for Numerical Methods in Engineering*, Vol. 2 , pp. 5-32, 1970.

Johnson, Claes. *Numerical solution of partial differential equations by the finite element method.* Cambridge: Cambridge University Press, 1987.

Kardestuncer, H, ed. *Finite Element Handbook.* New York: McGraw-Hill, 1987.

Keast, Patrick, "Moderate Degree Tetrahedral Quadrature Formulas", *Computer Methods in Applied Mechanics and Engineering*, Vol. 55, pp. 339-348, 1986.

Křížek, M, Liu, L, and Neittaanmäki, P., "On Harmonic and Biharmonic Finite Elements", *GAKUTO Internat. Ser. Math. Sci. Appl.*, Gakkotosho, Tokyo, 15, pp. 146-154, 2001.

Křížek, M, and Neittaanmäki, P. *Finite Element Approximation of Variational Problems and Applications.* Pitman Monographs and Surveys in Pure and Applied Mathematics Vol. 50, Longman Scientific & Technical, Harlow, 1990.

Kwon, Y. W. and Bang, H. *The Finite Element Method using MATLAB.* New York: CRC Press, 1997.

Liu, Liping, Davies, Kevin, Yuan, Kewei and Křížek, Michal, "On Symmetric Pyramidal Finite Elements", *Dynamics of Continuous Discrete and Impulsive Systems, Series B: Applications and Algorithms*, Vol. 11, No 1-2, pp. 213-227, 2004.

Maeztu, José, and Maza, Eduardo, "An Invariant Quadrature rule of Degree 11 for the Tetrahedron", *Comptes rendus de l'Académie des sciences, Série I: Mathématique*, Vol. 321, No. 9, pp. 1263-1267, 1995.

McCormick, Stephen F. *Multilevel Adaptive Methods for Partial Differential Equations.* Philadelphia: Society for Industrial and Applied Mathematics (SIAM), 1989.

Owen, Steven J. and Saigal, Sunil, "Formation of Pyramidal Elements for Hexahedra to Tetrahedra Transitions", *Computer Methods in Applied Mechanics and Engineering*, Vol. 190, pp. 4504-4518, 2001.

Rao, S. S. *Applied Numerical Methods for Engineers and Scientists.* Upper Saddle River, NJ: Prentice-Hall, 2002.

Stewart. David E. and Leyk, Zbigniew, "Meschach Library", version 1.2b, April 1994. Available at the Netlib Repository, <http://www.netlib.no/netlib/c/meschach/readme>

Stroud, A.H.. *Approximate Calculation of Multiple Integrals.* Englewood Cliffs, NJ: Prentice-Hall, 1971.

Stroud, A. H. *Numerical quadrature and solution of ordinary differential equations : a textbook for a beginning course in numerical analysis.* New York : Springer, 1974.

Swarztrauber, Paul N., "Fast Poisson Solvers", In *Studies in Numerical Analysis*, Golub, G. H. ed., Vol. 24, pp. 319-370, Washington, DC: Math. Assoc. America, 1984.

Turner, M. J., Clough, H., Martin, H. C., and Topp, J. L. "Stiffness and Deflection Analysis of Complex Structures", *Journal of the Aeronautical Sciences*, Vol. 23, No. 9, pp. 805-823, 1956.

Wendroff, Burton. *Theoretical Numerical Analysis*. New York: Academic Press, 1966.

Wieners, C., "Conforming Discretizations on Tetrahedrons, Pyramids, Prisms and Hexahedrons", Univ. Stuttgart, Bericht, 97/15, 1-9, 1997.

Zienkiewicz, O. C. *The Finite Element Method*, 3rd ed. New York: McGraw-Hill, 1989.

Zgainski, F.X., Coulomb, J.L., Maréchal, Y., Claeyssen, F., and Brunotte, X., "A New Family of Finite Elements : The Pyramidal Elements", *IEEE Transactions on Magnetics*, Vol.32, No.3, pp. 1393-1396, 1996.

Appendix I:
Basis Functions for Common Reference Elements

For convenience, we briefly summarize some of the more common reference elements and their basis function. For a more comprehensive listing please refer to [Kardestuncer, 1987, p. 2.121-2.131].

A1.1. Two-dimensional reference elements

3-node Linear Triangle

Let K be a triangle in the x,y plane with vertices $a_1 = (0,0)$, $a_2 = (1,0)$, $a_3 = (0,1)$. Then the basis functions $\varphi \in P_K = P_1$ are:

$$\varphi_1 = 1 - x - y$$
$$\varphi_2 = x$$
$$\varphi_3 = y.$$

6-node Quadratic triangle

Let K be a triangle in the x,y plane with vertices $a_1 = (0,0)$, $a_2 = (1,0)$, $a_3 = (0,1)$, and edge midpoints at $a_{12} = (\frac{1}{2},0)$, $a_{13} = (0,\frac{1}{2})$, $a_{23} = (\frac{1}{2},\frac{1}{2})$. Then the basis functions $\varphi \in P_K = P_2$ are:

$$\varphi_1 = (1 - x - y)(1 - 2x - 2y)$$
$$\varphi_2 = x(2x - 1)$$
$$\varphi_3 = y(2y - 1)$$
$$\varphi_{12} = 4x(1 - x - y)$$
$$\varphi_{13} = 4y(1 - x - y)$$
$$\varphi_{23} = 4xy.$$

4-node Bilinear Quadrilateral

Let K be a square in the x,y plane with vertices $a_1 = (-1,-1)$, $a_2 = (1,-1)$, $a_3 = (1,1)$, $a_4 = (-1,1)$. Then the basis functions $\varphi \in Q_K = Q_1$ are:

$$\varphi_1 = \tfrac{1}{4}(1 - x)(1 - y)$$
$$\varphi_2 = \tfrac{1}{4}(x + 1)(1 - y)$$
$$\varphi_3 = \tfrac{1}{4}(x + 1)(y + 1)$$

$$\varphi_4 = \tfrac{1}{4}(1-x)(y+1).$$

8-node Biquadratic Quadrilateral

Let K be a square in the x,y plane with vertices $a_1 = (-1,-1)$, $a_2 = (1,-1)$, $a_3 = (1,1)$, $a_4 = (-1,1)$, edge midpoints $a_5 = (0,-1)$, $a_6 = (1,0)$, $a_7 = (0,1)$, $a_8 = (-1,0)$. Then the basis functions $\varphi \in Q_K = Q_2$ are:

$$\varphi_1 = -\tfrac{1}{4}(1-x)(1-y)(x+y+1)$$
$$\varphi_2 = \tfrac{1}{4}(x+1)(1-y)(x-y-1)$$
$$\varphi_3 = \tfrac{1}{4}(x+1)(y+1)(x+y-1)$$
$$\varphi_4 = -\tfrac{1}{4}(1-x)(y+1)(x-y+1)$$
$$\varphi_5 = \tfrac{1}{2}(1-x)(x+1)(1-y)$$
$$\varphi_6 = \tfrac{1}{2}(x+1)(1-y)(y+1)$$
$$\varphi_7 = \tfrac{1}{2}(x+1)(1-x)(y+1)$$
$$\varphi_8 = \tfrac{1}{2}(1-x)(y+1)(1-y).$$

9-node Biquadratic Quadrilateral

Let K be a square in the x,y plane with vertices $a_1 = (-1,-1)$, $a_2 = (1,-1)$, $a_3 = (1,1)$, $a_4 = (-1,1)$, edge midpoints $a_5 = (0,-1)$, $a_6 = (1,0)$, $a_7 = (0,1)$, $a_8 = (-1,0)$, and a central node $a_9 = (0,0)$. Then the basis functions $\varphi \in Q_K = Q_2$ are:

$$\varphi_1 = \tfrac{1}{4}(1-x)(1-y)xy$$
$$\varphi_2 = -\tfrac{1}{4}(x+1)(1-y)xy$$
$$\varphi_3 = \tfrac{1}{4}(x+1)(y+1)xy$$
$$\varphi_4 = -\tfrac{1}{4}(1-x)(y+1)xy$$
$$\varphi_5 = -\tfrac{1}{2}(1-x)(1-y)(x+1)y$$
$$\varphi_6 = \tfrac{1}{2}(x+1)(1-y)(y+1)x$$
$$\varphi_7 = \tfrac{1}{2}(x+1)(y+1)(1-x)y$$
$$\varphi_8 = -\tfrac{1}{2}(1-x)(y+1)(1-y)x$$
$$\varphi_9 = (x+1)(1-x)(y+1)(1-y).$$

A1.2. Three-dimensional elements

4-node Linear Tetrahedron

Let K be a tetrahedron in x,y,z with vertices $a_1 = (0,0,0)$, $a_2 = (1,0,0)$, $a_3 = (0,1,0)$, $a_4 = (0,0,1)$. Then the basis functions $\varphi \in P_K = P_1$ are:

$$\varphi_1 = 1-x-y-z$$

$$\varphi_2 = x$$
$$\varphi_3 = y$$
$$\varphi_4 = z.$$

10-node Quadratic Tetrahedron

Let K be a tetrahedron in x,y,z with vertices $a_1 = (0,0,0)$, $a_2 = (1,0,0)$, $a_3 = (0,1,0)$, $a_4 = (0,0,1)$, and edge midpoint nodes $a_{12} = (\frac{1}{2},0,0)$, $a_{13} = (0,\frac{1}{2},0)$, $a_{14} = (0,0,\frac{1}{2})$, $a_{23} = (\frac{1}{2},\frac{1}{2},0)$, $a_{24} = (\frac{1}{2},0,\frac{1}{2})$, $a_{34} = (0,\frac{1}{2},\frac{1}{2})$. Then the basis functions $\varphi \in P_K = P_2$ are:

$$\varphi_1 = (1-x-y-z)(1-2x-2y-2z)$$
$$\varphi_2 = x(2x-1)$$
$$\varphi_3 = y(2y-1)$$
$$\varphi_4 = z(2z-1)$$
$$\varphi_{12} = 4x(1-x-y-z)$$
$$\varphi_{13} = 4y(1-x-y-z)$$
$$\varphi_{14} = 4z(1-x-y-z)$$
$$\varphi_{23} = 4xy$$
$$\varphi_{24} = 4xz$$
$$\varphi_{34} = 4yz.$$

8-node Trilinear Hexahedron

Let K be a hexahedron in x,y,z with vertices $a_1 = (-1,-1,-1)$, $a_2 = (1,-1,-1)$, $a_3 = (1,1,-1)$, $a_4 = (-1,1,-1)$, $a_5 = (-1,-1,1)$, $a_6 = (1,-1,1)$, $a_7 = (1,1,1)$, $a_8 = (-1,1,1)$. Then the basis functions $\varphi \in Q_K = Q_1$ are:

$$\varphi_1 = \tfrac{1}{8}(1-x)(1-y)(1-z)$$
$$\varphi_2 = \tfrac{1}{8}(x+1)(1-y)(1-z)$$
$$\varphi_3 = \tfrac{1}{8}(x+1)(y+1)(1-z)$$
$$\varphi_4 = \tfrac{1}{8}(1-x)(y+1)(1-z)$$
$$\varphi_5 = \tfrac{1}{8}(1-x)(1-y)(z+1)$$
$$\varphi_6 = \tfrac{1}{8}(x+1)(1-y)(z+1)$$
$$\varphi_7 = \tfrac{1}{8}(x+1)(y+1)(z+1)$$
$$\varphi_8 = \tfrac{1}{8}(1-x)(y+1)(z+1).$$

27-node Triquadratic Hexahedron

Let K be a hexahedron in x,y,z with corner vertices $a_1 = (-1,-1,-1)$, $a_3 = (1,-1,-1)$,

$a_5 = (1,1,-1)$, $a_7 = (-1,1,-1)$, $a_{19} = (-1,-1,1)$, $a_{21} = (1,-1,1)$, $a_{23} = (1,1,1)$, $a_{25} = (-1,1,1)$. Then the basis functions for Q_K in Q_2 are:

$$\varphi_1 = -\tfrac{1}{8}(1-x)(1-y)(1-z)xyz$$
$$\varphi_2 = \tfrac{1}{4}(1-x)(1+x)(1-y)(1-z)yz$$
$$\varphi_3 = \tfrac{1}{8}(1+x)(1-y)(1-z)xyz$$
$$\varphi_4 = -\tfrac{1}{4}(1+x)(1-y)(1+y)(1-z)xz$$
$$\varphi_5 = -\tfrac{1}{8}(1+x)(1+y)(1-z)xyz$$
$$\varphi_6 = -\tfrac{1}{4}(1+x)(1-x)(1+y)(1-z)yz$$
$$\varphi_7 = \tfrac{1}{8}(1-x)(1+y)(1-z)xyz$$
$$\varphi_8 = \tfrac{1}{4}(1-x)(1+y)(1-y)(1-z)xz$$
$$\varphi_9 = -\tfrac{1}{2}(1+x)(1-x)(1+y)(1-y)(1-z)z$$
$$\varphi_{10} = \tfrac{1}{4}(1-x)(1-y)(1-z)(1+z)xy$$
$$\varphi_{11} = -\tfrac{1}{2}(1-x)(1+x)(1-y)(1-z)(1+z)y$$
$$\varphi_{12} = -\tfrac{1}{4}(1+x)(1-y)(1-z)(1+z)xy$$
$$\varphi_{13} = \tfrac{1}{2}(1+x)(1-y)(1+y)(1-z)(1+z)x$$
$$\varphi_{14} = \tfrac{1}{4}(1+x)(1+y)(1-z)(1+z)xy$$
$$\varphi_{15} = \tfrac{1}{2}(1+x)(1-x)(1+y)(1-z)(1+z)y$$
$$\varphi_{16} = -\tfrac{1}{4}(1-x)(1+y)(1-z)(1+z)xy$$
$$\varphi_{17} = -\tfrac{1}{2}(1-x)(1+y)(1-y)(1-z)(1+z)x$$
$$\varphi_{18} = (1+x)(1-x)(1+y)(1-y)(1-z)(1+z)$$
$$\varphi_{19} = \tfrac{1}{8}(1-x)(1-y)(1+z)xyz$$
$$\varphi_{20} = -\tfrac{1}{4}(1-x)(1+x)(1-y)(1+z)yz$$
$$\varphi_{21} = -\tfrac{1}{8}(1+x)(1-y)(1+z)xyz$$
$$\varphi_{22} = \tfrac{1}{4}(1+x)(1-y)(1+y)(1+z)xz$$
$$\varphi_{23} = \tfrac{1}{8}(1+x)(1+y)(1+z)xyz$$
$$\varphi_{24} = \tfrac{1}{4}(1+x)(1-x)(1+y)(1+z)yz$$
$$\varphi_{25} = -\tfrac{1}{8}(1-x)(1+y)(1+z)xyz$$
$$\varphi_{26} = -\tfrac{1}{4}(1-x)(1+y)(1-y)(1+z)xz$$
$$\varphi_{27} = \tfrac{1}{2}(1+x)(1-x)(1+y)(1-y)(1+z)z.$$

Appendix II:
Program Source Code Listings

The following C program is used to test the properties of the different pyramidal elements presented in section three. Note that the Meschach code library [Stewart, Leyk, 1994] is required in order to compile this program.

```
/*********************************************************************
* Programmer: Kevin Davies
* File: pyrprog.c
* Version: 4.0
* Date last modified: Dec 12, 2004
* Description: This is the top-level program for testing pyramidal
*   elements.  When run, the user is prompted to select element type
*   (5-node linear, 13-node or 14-node quadratic), symmetry settings
*   true solution, solver method, and to enter the mesh control
*   paremeter N.
*   The user can also select whether to run the profram as a
*   background process or interactively.  Some options for output
*   are also available.
*
*   Note: This Program uses routines for sparse matrices from
*   the Meschach code Library by David E. Steward & Zbigniew Leyk

*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// following includes needed to spawn new processes
#include <sys/types.h>
#include <unistd.h>

// Meschach file includes:
#include "../meschach/matrix.h"
#include "../meschach/matrix2.h"
#include "../meschach/sparse.h"
#include "../meschach/sparse2.h"
#include "../meschach/iter.h"

/************ Constants ***********/
// Boundry condition for all boundry nodes
#define BOUNDRY 0
// value assigned to all node with unknowns
#define INTERNAL -200
// output file containing true & approx. solution and error vectors
#define PYR5 1
#define PYR13 2
#define PYR14_CASE_I 3
#define PYR14_CASE_II 4
```

```
#define SYMMETRIC 1
#define NONSYMMETRIC 2
#define GAUSS_ELIM 1
#define GAUSS_SEIDEL 2
#define CG 3
// Absolute Size limit
#define MAXSIZE 128
#define CUBESIZE 1.0
#define SOLFILE "solution.txt"
#define MATFILE "matrix.m"
#define TEMPFILE "matrix.temp"
// Log file containinig final errors and solution times
#define LOGFILE "result-ctri-dec04-18.txt"

// Type definition for program timing structure
typedef struct   {
   clock_t  begin_clock,
            save_clock;
   time_t   begin_time,
            save_time;
} time_keeper;

/************************  Glabal variables  *******************************
const int quadDeg4pts = 11;  // number of points for degree 4 quadrature
const int quadDeg7pts = 31;  // number of points for degree 7 quadrature
int **elements;              // array for elements
double **nodeCors;           // array for node coordinates
int **nodeTypes;             // array for node types and boundry condition
int noElemts, noCubes, noNodes, unknowns = 0;
int background;              // true if spawning background process
int symBasis;    // true if applying symmetries to basis functions,
                    // false otherwise
int pyrType;     // type of pyramid to be used for calculation
int nodesPerElm;   // number of nodes per element
double pi;
time_keeper tk;  // timing variable

// constants for booleans
const int trueVal = 1;
const int falseVal = 0;

// include files containing basis functions and derivatives
#include "pyr5basis.c"
#include "pyr13basis.c"
#include "pyr14basis.c"
// include file contaiing true solutions
#include "solution.c"
// include file for 11 point, degree 4 gaussian cubeature
#include "cubeature4.c"
// include file for 24 point, degree 6 gaussian cubeature
#include "cubeature6.c"
// include file for 31 point, degree 7 gaussian cubeature
#include "cubeature7.c"
// include file for 45 point, degree 8 gaussian cubeature
#include "cubeature8.c"
// include file for 87 point, degree 11 gaussian cubeature
#include "cubeature11.c"

// include files for constructing mesh and linear system
```

```
#include "mesh_gen.c"
#include "entry_intgrls.c"
// include file containing linear system solvers
#include "solvers.c"

/***********  Function Prototypes  ***********/
void init(int size);
void matrixGen(SPMAT *A, VEC *rhsv);
double det3x3(MAT *A);
void v_abs(VEC *v);
double rhs_entry(int i, MAT *X, double detj);
double elm_entry(int i, int j, MAT *X, double detj);
double solution(double x, double y, double z);
double l2IntegNorm(VEC *uh);
double elemnInteg(int elem, MAT *X, VEC *uh, double detj);
void start_time(void);
void stop_time(double *user_t, double *real_t);

int main(void)   {

   int size = 2;   // (default) size of mesh (number of small cubes along
                                                one side)
   int selection;  // variable to hold user selections
   SPMAT *A, *LLT;   // A is system matrix, LLT used in solving system
   VEC *f, *U, *u_approx, *error, *uh;   // f: RHS vector, U: true sol'n vector
                                         // u_approx: approx sol'n vector,
                                         // error: error vector
   int save;        // true/false save matrix & solution to files
   int solverType;  // Type of solver to be used
   int cId; // child ID for forking backgroung process
   int num_steps, i, j;
   double maxError, rowsum, L2norm; // variables for error
   double user_time, real_time;    // variables used for timing
   int c;
   char saveResponce, junk;   // variabls to hold user selections
   FILE *solOfp, *matOfp, *tempFile, *errorOfp; // output files

   // temp variables
   int rowCount = 0, rowNonzeros = 0, totNonzeros = 0;

/******************  Get User Input  ****************************/
   printf("\t    Pyramidal Element - Test Program\n");
   printf("\t------ Please select Element Type ------\n");
   printf("\t 5-Node Pyramid Element                (1)\n");
   printf("\t13-Node Pyramid Element                (2)\n");
   printf("\t14-Node Pyramid Element                (3)\n");
   printf("\tExit program                           (4)\n\t-> ");
   scanf("%d", &selection);
   if((selection <1) || (selection > 3))  {
      printf("Exiting program...\n");
      exit(1);
   }
   else
      if(selection == 1)   {
         pyrType = PYR5;
      }
      else
         if(selection == 2)   {
            pyrType = PYR13;
```

```
      }
      else
         if(selection == 3)   {
            printf("\t          PYR-14 - test program\n");
            printf("\t---------Please make a selection--------\n");
            printf("\tCase I Basis functions                (1)\n");
            printf("\tCase II Basis functions               (2)\n\t-> ");
            scanf("%d", &selection);
            if(selection == 1)
               pyrType = PYR14_CASE_I;
            else
               if(selection == 2)
                  pyrType = PYR14_CASE_II;
               else  {
                  printf("Invalid selection, Exiting program...\n");
                  exit(1);
               }
         }
if(pyrType == PYR5)
   printf("\t          PYR-5 - test program\n");
else
   if(pyrType == PYR13)
      printf("\t          PYR-13 - test program\n");
   else
      if(pyrType == PYR14_CASE_I)  {
         printf("\t          PYR-14 - test program\n");
         printf("\t         Case I basis functions\n");
      }
      else  {
         printf("\t          PYR-14 - test program\n");
         printf("\t         Case II basis functions\n");
      }
printf("\t---------Please make a selection--------\n");
printf("\tApply standard basis functions       (1)\n");
printf("\tApply extra symmetries               (2)\n");
printf("\tExit program                         (3)\n\t-> ");
scanf("%d", &selection);
if(selection == 1)   {
   symBasis = falseVal;
   printf("\tApplying standard basis functions\n");
}
else   {
   if(selection == 2)   {
      symBasis = trueVal;
      printf("\tApplying extra symmetries\n");
   }
   else   {
      printf("Exiting program...\n");
      exit(1);
   }
}
printf("\t----  Please select Solver Method  -----\n");
printf("\tGaussian Elimination                 (1)\n");
printf("\tGauss-Seidel                         (2)\n");
printf("\tPreconditioned Conjugate Gradient    (3)\n\t-> ");
scanf("%d", &selection);
if(selection == 1)
   solverType = GAUSS_ELIM;
else
```

```
      if(selection == 2)
         solverType = GAUSS_SEIDEL;
      else
         solverType = CG;  //use CG by default

   printf("\tPlease enter size of mesh (N-value)\n\t-> ");
   scanf("%d", &size);
   if((size > MAXSIZE) || (size < 2))   {
      printf("Cannot support mesh with N-value of: %d. Program halted\n",
                  size);
      exit(1);
   }
   printf("\tSave matrix and solution to files? (y/n)\n\t-> ");
   scanf("%c%c", &junk, &saveResponce);
   if((saveResponce == 'y') || (saveResponce == 'Y'))
      save = 1;
   else
      save = 0;
   printf("\tSpawn background process? (y/n)\n\t-> ");
   scanf("%c%c", &junk, &saveResponce);
   if((saveResponce == 'y') || (saveResponce == 'Y'))
      background = 1;
   else
      background = 0;

   if(background)   {
      printf("Spawning child process for calculations\n");
      printf("Results will be writen to: %s\n", LOGFILE);
      if ((cId = fork()) != 0)   {
         printf("child pid: %d\n", cId);
         return 0;
      }
   }

/******************  Generate System  ***************************/
   init(size);
   makeMesh(size);  // #of unknowns not known until this done

   // Allocate matrix and vectors
   A = sp_get(unknowns, unknowns, 27);
   f = v_get(unknowns);
   U = v_get(unknowns);
   error = v_get(unknowns);
   // printf("generating system matrix and RHS...\n");

   matrixGen(A, f);
   if(save)   {
      tempFile = fopen(TEMPFILE, "w");
      if(!tempFile)
         printf("Unable to save to matrix file\n");
      else  {
         printf("Saving matrix to file %s\n", MATFILE);
         for(i = 0; i< unknowns; i++)  {
            for(j = 0; j< unknowns; j++)  {
               if(sp_get_val(A, i, j) != 0.0)   {
                  fprintf(tempFile, "%d,%d,%.10f\n",i,j,sp_get_val(A, i, j));
                  rowCount++;
               }
            }
```

```
            if(rowCount > rowNonzeros)
               rowNonzeros = rowCount;
            totNonzeros += rowCount;
            rowCount = 0;
         }
         printf("the most non-zero entries in a row is: %d\n", rowNonzeros);
         printf("Total non-zero entries is: %d\n", totNonzeros);
         fclose(tempFile);
         matOfp = fopen(MATFILE, "w");
         tempFile = fopen(TEMPFILE, "r");
         if(!tempFile || !matOfp)
            printf("Unable to save to matrix file\n");
         else   {
            fprintf(matOfp, "%d,%d,%d\n", unknowns, unknowns, totNonzeros);
            c = getc(tempFile);
            while(c != EOF)   {
               putc(c, matOfp);
               c = getc(tempFile);
            }
            fclose(matOfp);
            fclose(tempFile);
         }
         remove(TEMPFILE);
      }
   }
/************************* Solve System ******************************/

   u_approx = v_get(A->m);

// Start timer
   start_time();
   if(solverType == GAUSS_ELIM)  {
      naiveGauss(A, f, u_approx);
   }
   else
      if(solverType == GAUSS_SEIDEL)   {
         for(i = 0; i < A->m; i++)
            u_approx->ve[i] = 0.0;  // Use zero vector as initial guess
         GaussSeidel(A, f, u_approx, 100, 0.000001, 1.0);
      }
      else
         if(solverType == CG) {
            LLT = sp_copy(A);
            spICHfactor(LLT);
            iter_spcg(A, LLT, f ,1e-7, u_approx, 1000, &num_steps);
         }
// stop timer
   stop_time(&user_time, &real_time);

// *** Done solving ***

   for(i = 0; i < noNodes; i++)  {
     if (nodeTypes[i][1] < 0)
        U->ve[nodeTypes[i][0]-1] = solution(nodeCors[i][0],
                                            nodeCors[i][1], nodeCors[i][2]);
   }

/************  Calculate error  *************************/
   v_sub(U, u_approx, error);
```

```
   maxError = v_norm_inf(error);

   if(!save)   {
      V_FREE(U);
      V_FREE(error);
   }
   uh = v_get(noNodes);
   for(i = 0; i < noNodes; i++)  {
      if (nodeTypes[i][1] == BOUNDRY)   {
         uh->ve[i] = 0.0;
      }
      else  {
         uh->ve[i] = u_approx->ve[nodeTypes[i][0]-1];
      }
   }
   L2norm = l2IntegNorm(uh);

/********************  Write to output files ************************/

   if(!background)   {
      printf("Maximum error for size %d mesh: %.10f\n", size, maxError);
      printf("L2 norm of error for size %d mesh: %.10f\n", size, L2norm);
      printf("User time to solve system: %.10f sec.\n", user_time);
      printf("Real time to solve system: %.10f sec.\n", real_time);
   }
   errorOfp = fopen(LOGFILE, "a");
   if(errorOfp)   {
      switch(pyrType)   {
      case PYR5:
         fprintf(errorOfp, "Results for 5-node Pyramidal Element:\n");
         break;
      case PYR13:
         fprintf(errorOfp, "Results for 13-node Pyramidal Element:\n");
         break;
      case PYR14_CASE_I:
         fprintf(errorOfp, "%s\n",
               "Results for 14-node Pyramidal Element, and Case I Basis:");
         break;
      case PYR14_CASE_II:
         fprintf(errorOfp, "%s\n",
               "Results for 14-node Pyramidal Element, and Case II Basis:");
      }
      if(symBasis)
         fprintf(errorOfp, "%s\n",
               "Applying extra symmetries to basis functions results in:");
      else
         fprintf(errorOfp,"Applying standard basis functions results in:\n");
      fprintf(errorOfp, "Maximum error for size %d mesh: %.10f\n",
                        size, maxError);
      fprintf(errorOfp, "L2 norm of error for size %d mesh: %.10f\n",
                        size, L2norm);
      switch(solverType)   {
      case GAUSS_ELIM:
         fprintf(errorOfp, "Solution time using Gaussian Elimination:\n");
         break;
      case GAUSS_SEIDEL:
         fprintf(errorOfp, "Solution time using Gauss-Seidel:\n");
         break;
      case CG:
```

```
            fprintf(errorOfp, "%s\n",
                "Solution time using Conjugate Gradient with preconditioning:");
        }
        fprintf(errorOfp,"User time to solve system: %.10f sec.\n",user_time);
        fprintf(errorOfp,"Real time to solve system: %.10f sec.\n",real_time);
        fclose(errorOfp);
    }
    if(save)   {
        solOfp = fopen(SOLFILE, "w");
        if(!solOfp)
            printf("Unable to write to output file\n");
        else  {
            fprintf(solOfp, "Node #    True Sol'n  Approx. Sol'n   Error\n");
            for(i = 0; i < noNodes; i++)  {
                if (nodeTypes[i][1] == BOUNDRY)   {
                    fprintf(solOfp, "%d  %f  %f  %f\n", i, 0.0, 0.0, 0.0);
                }
                else  {
                    fprintf(solOfp, "%d  %f  %f  %f\n",i,U->ve[nodeTypes[i][0]-1],
                        u_approx->ve[nodeTypes[i][0]-1],
                        error->ve[nodeTypes[i][0]-1]);
                }
            }
        }
        fclose(solOfp);
    }
    return 0;
} //end main

/*****************************************************************************
*
*  Initialization function to allocate memory and set global variables
*
*****************************************************************************/
void init(int size)  {

    int i;
    noElemts = 6.0 * pow(size, 3);
    noCubes = pow(size, 3);
    if(pyrType == PYR5)  {
        noNodes = pow(size+1, 3) + pow(size, 3);
        nodesPerElm = 5;
    }
    else
        if(pyrType == PYR13) {
            noNodes = 13.0*pow(size, 3) + 9.0*pow(size, 2) + 6.0*size + 1;
            nodesPerElm = 13;
        }
        else  {     // Assume 14-node
            noNodes = 16.0*pow(size, 3) + 12.0*pow(size, 2) + 6.0*size + 1;
            nodesPerElm = 14;
        }

    pi = 4.0 * atan(1.0);

    elements = malloc(sizeof(int*) * noElemts);   // one row per element
    if (elements == NULL)   {
        printf("Memory allocation error -- program halted\n");
        exit(1);
```

```
    }
    for(i = 0; i < noElemts; i++)   {
       elements[i] = malloc(sizeof(int) * nodesPerElm);
       if (elements[i] == NULL)   {
          printf("Memory allocation error -- program halted\n");
          exit(1);
       }
    }
    nodeCors = malloc(sizeof(double*) * noNodes);   // one row per node
    if (nodeCors == NULL)   {
       printf("Memory allocation error -- program halted\n");
       exit(1);
    }
    for(i = 0; i < noNodes; i++)   {
       nodeCors[i] = malloc(sizeof(double) * 3); //three corrdinates per node
       if (nodeCors[i] == NULL)   {
          printf("Memory allocation error -- program halted\n");
          exit(1);
       }
    }
    nodeTypes = malloc(sizeof(int*) * noNodes);   // one row per node
    if (nodeTypes == NULL)   {
       printf("Memory allocation error -- program halted\n");
       exit(1);
    }
    for(i = 0; i < noNodes; i++)   {
       nodeTypes[i] = malloc(sizeof(int) * 2);   // two entries per node
       if (nodeTypes[i] == NULL)   {
          printf("Memory allocation error -- program halted\n");
          exit(1);
       }
    }
}

// Generate system matrix and RHS vector
void matrixGen(SPMAT *A, VEC *rhsv)   {

   int firstTime = TRUE, i, j, k, l, m, Ai, Aj, ri;
   double temp1, temp2, detj;

   MAT *X, *Jac; // X: matrix containing x,y,z coordinates,
                 // Jac: Jacobian matrix

   X = m_get(nodesPerElm,3);
   Jac = m_get(3,3);

   for(k = 0; k < noElemts; k++) {

      if(!background)   {  // show progress
         if((k % (noElemts/10)) == 0)
            printf("building element entry %d of %d total entries\n",
                        k, noElemts);
      }
      for(i = 0; i < nodesPerElm; i++)  {
         if (nodeTypes[elements[k][i]][1] < 0)  {
            for(l = 0; l < nodesPerElm; l++)   {
               for(m = 0; m < 3; m++)
                  X->me[l][m] = nodeCors[elements[k][l]][m];
            }
```

```
            if (firstTime == TRUE)  {
               jacob(X, Jac);
               detj = det3x3(Jac);
               firstTime = FALSE;
            }
            temp1 = rhs_entry(i, X, detj);
            ri = nodeTypes[elements[k][i]][0]-1;
            rhsv->ve[ri] = rhsv->ve[ri] + temp1;
            for(j = 0; j < nodesPerElm; j++)  {
               if (nodeTypes[elements[k][j]][1] < 0)  {
                  temp2 = elm_entry(i, j, X, detj);
                  Ai = nodeTypes[elements[k][i]][0]-1;
                  Aj = nodeTypes[elements[k][j]][0]-1;
                  temp2 = temp2 + sp_get_val(A, Ai, Aj);
                  sp_set_val(A, Ai, Aj, temp2);
               }
            }
         }
      }
   }
   M_FREE(X);
   M_FREE(Jac);

}

// Function to calculate deteminant of 3x3 matrix
double det3x3(MAT *A)   {
   double det;

   det=(A->me[0][0]*A->me[1][1]*A->me[2][2]-
        A->me[0][0]*A->me[1][2]*A->me[2][1])-
       (A->me[0][1]*A->me[1][0]*A->me[2][2]-
        A->me[0][1]*A->me[1][2]*A->me[2][0])+
       (A->me[0][2]*A->me[1][0]*A->me[2][1]-
        A->me[0][2]*A->me[1][1]*A->me[2][0]);
   return det;
}

// Function to make vector absolute value
void v_abs(VEC *v)   {
   int i;

   for(i = 0; i < v->dim; i++)   {
      if(v->ve[i] < 0.0)
         v->ve[i] = -1.0 * v->ve[i] ;
   }
}

/*****************************************************************************
* Author: Kevin Davies
* Description: function for finding a single RHS load vector entry.
*  inputs:
*     X - a matrix of global coordinates, s.t.
*         the first column consists of the x coordinate values, the second
*         column consists of y values, and the third consists of z values.
*     i - element node index
*     detj - determinant of Jacobian matrix
*  outputs:
*     returns value to be added to RHS vector entry
```

```
*****************************************************************************/
double rhs_entry(int i, MAT *X, double detj) {
   int j;
   double vol = 1.0/3.0, ui = 0.0;

   if(symBasis)   {
      for(j = 1; j <=4; j++)  {
         ui = ui + rhs_tet(X, i, j);
      }
   }
   else   {
      vol = 2.0/3.0;
      for(j = 1; j <=2; j++)  {
         ui = ui + rhs_tet(X, i, j);
      }
   }
   ui = (ui * detj * vol);
   return ui;
}

/*****************************************************************************
% Function: elm_entry
% Description: function to find value of i,j-th entry of the element
%    matrix.
% inputs:
%     X - a matrix of global coordinates, s.t.
%         the first column consists of the x coordinate values, the second
%         column consists of y values, and the third consists of z values.
%     i,j - the i,j index values indicating the current entry in the
%         element matrix.
*****************************************************************************/

double elm_entry(int i, int j, MAT *X, double detj)  {
   double aij = 0.0, vol = 1.0/3.0;
   int k;
   if(symBasis)   {
      for(k = 1; k <= 4; k++) {
         aij = aij + tet(X, i, j, k);
      }
   }
   else   {
      vol = 2.0/3.0;
      for(k = 1; k <= 2; k++) {
         aij = aij + tet(X, i, j, k);
      }
   }
   aij = (aij * detj * vol);
   return aij;
}

/*****************************************************************
*
*  Funtion for calculating L_2 Norm of error
*     (i.e., ||u-u_h||_{L_2} )
*
*****************************************************************/

double l2IntegNorm(VEC *uh)   {
   double l2norm = 0.0, elemSum = 0.0;
```

```
    int firstTime = TRUE, i, j, k, l, m;
    double detj;

    MAT *X, *Jac; // X: matrix containing x,y,z coordinates,
                  // Jac: Jacobian matrix
    X = m_get(nodesPerElm,3);
    Jac = m_get(3,3);
    for(k = 0; k < noElemts; k++) {
       for(l = 0; l < nodesPerElm; l++)    {
          for(m = 0; m < 3; m++)
             X->me[l][m] = nodeCors[elements[k][l]][m];
       }
       if (firstTime == TRUE)  {
          jacob(X, Jac);
          detj = det3x3(Jac);
          firstTime = FALSE;
       }
       elemSum = elemSum + elemnInteg(k, X, uh, detj);
    }
    M_FREE(X);
    M_FREE(Jac);
    l2norm = sqrt(elemSum);
    return l2norm;
}

/*********************************************************************
*
*  Support function for calculating error norm, find integral
*     over element.
*
*********************************************************************/
double elemnInteg(int elem, MAT *X, VEC *uh, double detj)    {
    int tet, noTets;
    double tetInteg = 0.0, vol;
    MAT *N, *A, *K;
    VEC *Q;
    int k, m, i, quadPoints;
    double temp = 0.0, temp2, wi;
    N = m_get(1,nodesPerElm);
    A = m_get(3,4);
    K = m_get(1,3);
    Q = v_get(3);
    switch(pyrType)    {
       case PYR5:
          quadPoints = quadDeg4pts;
          break;
       case PYR13:
          quadPoints = quadDeg7pts;
          break;
       case PYR14_CASE_I:
          quadPoints = quadDeg7pts;
          break;
       case PYR14_CASE_II:
          quadPoints = quadDeg7pts;
    }
    if(symBasis)    {
       vol = 1.0/3.0;
       noTets = 4;
    }
```

```
else   {
   vol = 2.0/3.0;
   noTets = 2;
}
for(tet = 1; tet <= noTets; tet++)    {
   getTetPoints(A, tet);
   for(k = 0; k < quadPoints; k++)  {
      switch(pyrType)   {
         case PYR5:
            cubature4(A, k+1, Q, &wi);
            break;
         case PYR13:
            cubature7(A, k+1, Q, &wi);
            break;
         case PYR14_CASE_I:
            cubature7(A, k+1, Q, &wi);
            break;
         case PYR14_CASE_II:
            cubature7(A, k+1, Q, &wi);
      }
      for(m = 1; m <= nodesPerElm; m++)  {
         if(symBasis)   {
            switch(pyrType)   {
            case PYR5:
               N->me[0][m-1] = pyr5symBasis(m-1, Q->ve[0],
                                            Q->ve[1], Q->ve[2]);
               break;
            case PYR13:
               N->me[0][m-1] = pyr13symBasis(m-1, Q->ve[0],
                                             Q->ve[1], Q->ve[2]);
               break;
            case PYR14_CASE_I:
               N->me[0][m-1] = pyr14C1symBasis(m-1, Q->ve[0],
                                               Q->ve[1], Q->ve[2]);
               break;
            case PYR14_CASE_II:
               N->me[0][m-1] = pyr14C2symBasis(m-1, Q->ve[0],
                                               Q->ve[1], Q->ve[2]);
            }
         }
         else  {
            switch(pyrType)   {
            case PYR5:
               N->me[0][m-1] = pyr5basis(m-1, Q->ve[0],
                                         Q->ve[1], Q->ve[2]);
               break;
            case PYR13:
               N->me[0][m-1] = pyr13basis(m-1, Q->ve[0],
                                          Q->ve[1], Q->ve[2]);
               break;
            case PYR14_CASE_I:
               N->me[0][m-1] = pyr14C1basis(m-1, Q->ve[0],
                                            Q->ve[1], Q->ve[2]);
               break;
            case PYR14_CASE_II:
               N->me[0][m-1] = pyr14C2basis(m-1, Q->ve[0],
                                            Q->ve[1], Q->ve[2]);
            }
         }
```

```
            }
            m_mlt(N, X, K);
            temp = 0.0;
            for(i = 0; i < nodesPerElm; i++)    {
               if(symBasis)    {
                  switch(pyrType)    {
                  case PYR5:
                     temp = temp + uh->ve[elements[elem][i]] *
                              pyr5symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                     break;
                  case PYR13:
                     temp = temp + uh->ve[elements[elem][i]] *
                              pyr13symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                     break;
                  case PYR14_CASE_I:
                     temp = temp + uh->ve[elements[elem][i]] *
                              pyr14C1symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                     break;
                  case PYR14_CASE_II:
                     temp = temp + uh->ve[elements[elem][i]] *
                              pyr14C2symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                  }

               }
               else  {
                  switch(pyrType)    {
                  case PYR5:
                     temp = temp + uh->ve[elements[elem][i]] *
                               pyr5basis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                     break;
                  case PYR13:
                     temp = temp + uh->ve[elements[elem][i]] *
                               pyr13basis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                     break;
                  case PYR14_CASE_I:
                     temp = temp + uh->ve[elements[elem][i]] *
                               pyr14C1basis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                     break;
                  case PYR14_CASE_II:
                     temp = temp + uh->ve[elements[elem][i]] *
                               pyr14C2basis(i, Q->ve[0], Q->ve[1], Q->ve[2]);
                  }

               }
            }
            temp2 = solution(K->me[0][0], K->me[0][1], K->me[0][2]) - temp;
            tetInteg = tetInteg + wi * pow(temp2, 2);
         }
      }
      M_FREE(N);
      M_FREE(A);
      M_FREE(K);
      V_FREE(Q);
      return (tetInteg * vol * detj);
}

/*****************************************************************
*
*  Timing functions to start/stop test clock
```

```
*
**********************************************************************/
void start_time(void)    {
   tk.begin_clock = tk.save_clock = clock();
   tk.begin_time = tk.save_time = time(NULL);
}

void stop_time(double *user_t, double *real_t)    {
   tk.save_clock = clock();
   tk.save_time = time(NULL);
   *user_t = (tk.save_clock - tk.begin_clock) / (double) CLOCKS_PER_SEC;
   *real_t = difftime(tk.save_time, tk.begin_time);
}
```

```
/*****************************************************************************
* Author: Kevin Davies
* File mesh_gen.c
* Version: 4.0
* Date last modified: Dec 15, 2004
* Description: functions for forming correspondence between element
*     nodes numbers and global node numbering, as well as calculating
*     global nodal coordinate values.
*****************************************************************************/

/*****************************************************************************
*                                5-Node Pyramid Case                         *
*****************************************************************************/

void pyr5mesh(int size) {
   int i,j,k, index, size_sq, size_p1_sq;
   int l,m;  // variables for testing

   size_sq = pow(size, 2);
   size_p1_sq = pow(size+1, 2);
   for(i = 0; i < size; i++)   {   // each layer of cubes
      for(j = 0; j < size; j++)  {   // each row of cubes
         for(k = 0; k < size; k++)   {   // each column of cubes
               // 1st (bottom) element of the cube
            index = (i*size*size+j*size+k)*6;
            elements[index][0] = (size_p1_sq + size_sq) *
                                 (i+1) + (size+1) * j + k;
            elements[index][1] = (size_p1_sq + size_sq) *
                                 (i+1) + (size+1) * j + k+ 1;
            elements[index][2] = (size_p1_sq + size_sq) * i +
                                 (size+1) * j + 1 + k;
            elements[index][3] = (size_p1_sq + size_sq) *
                                 i + (size+1) * j + k;
            elements[index][4] = size_p1_sq * (i+1) + size_sq
                                 * i + (size) * j + k;
               // 2nd (right-side) element of the cube
            elements[index+1][0] = (size_p1_sq + size_sq) *
                                    (i+1) + (size+1) * j + k+ 1;
            elements[index+1][1] = (size_p1_sq + size_sq) * (i+1)
                                    + (size+1) * j + k+ 1 + size+1;
            elements[index+1][2] = (size_p1_sq + size_sq) * i+(size+1)*
                                    j + 1 + k+ size+1;
            elements[index+1][3] = (size_p1_sq + size_sq) * i+(size+1)*
                                    j + 1+ k;
            elements[index+1][4] = size_p1_sq * (i+1)+size_sq * i+(size)*
                                    j + k;
               // 3rd (top) element of the cube
            elements[index+2][0] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
                                    j + k+ 1 + size+1;
            elements[index+2][1] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
                                    j + k+ size+1;
            elements[index+2][2] = (size_p1_sq+size_sq) * i+(size+1)*
                                    j + k+ size+1;
            elements[index+2][3] = (size_p1_sq+size_sq) * i+(size+1)*
                                    j + k+ 1 + size+1;
            elements[index+2][4] = size_p1_sq * (i+1)+size_sq * i+(size)*
                                    j + k;
               // 4th (left-side) element of the cube
            elements[index+3][0] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
```

```
                                    j + k+ size+1;
            elements[index+3][1] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
                                    j+ k;
            elements[index+3][2] = (size_p1_sq+size_sq) * i+(size+1)*
                                    j+ k;
            elements[index+3][3] = (size_p1_sq+size_sq) * i+(size+1)*
                                    j + k+ size+1;
            elements[index+3][4] = size_p1_sq * (i+1)+size_sq * i+(size)*
                                    j + k;
               // 5th (front) element of the cube
            elements[index+4][0] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
                                    j + k+ size+1;
            elements[index+4][1] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
                                    j + k+ 1 + size+1;
            elements[index+4][2] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
                                    j + k+ 1;
            elements[index+4][3] = (size_p1_sq+size_sq) * (i+1)+(size+1)*
                                    j+ k;
            elements[index+4][4] = size_p1_sq * (i+1)+size_sq * i+(size)*
                                    j + k;
               // 6th (back) element of the cube
            elements[index+5][0] = (size_p1_sq + size_sq) * i+(size+1)*
                                    j+ k;
            elements[index+5][1] = (size_p1_sq + size_sq) * i+(size+1)*
                                    j + k+ 1;
            elements[index+5][2] = (size_p1_sq + size_sq) * i+(size+1)*
                                    j + k+ 1 + size+1;
            elements[index+5][3] = (size_p1_sq + size_sq) * i+(size+1)*
                                    j + k+ size+1;
            elements[index+5][4] = size_p1_sq * (i+1)+size_sq * i+(size)*
                                    j + k;
         }
      }
   }
}

void pyr5mapping(int size)  {
   int nodeIndex, i, j, k;

   // calculate x,y,z coordinates for corner nodes of small cubes
   for(i = 0; i < size+1; i++)   {   // each layer of cubes
      for(j = 0; j < size+1; j++)  {   // each row of cubes
         for(k = 0; k < size+1; k++)   {   // each column of cubes
            nodeIndex = (int)(pow(size+1, 2) + pow(size, 2))*i+(size+1)*j+k;
            nodeCors[nodeIndex][0] = (double) k*CUBESIZE/size;
            nodeCors[nodeIndex][1] = (double) j*CUBESIZE/size;
            nodeCors[nodeIndex][2] = (double) i*CUBESIZE/size;
         }
      }
   }

   // calculate x,y,z coordinates for centre nodes of cubes
   for(i = 0; i < size; i++)   {   // each layer of cubes
      for(j = 0; j < size; j++)  {   // each row of cubes
         for(k = 0; k < size; k++)   {   // each column of cubes
           nodeIndex=(int)pow(size+1,2)*(i+1)+(int)pow(size,2)*i+(size)*j+k;
           nodeCors[nodeIndex][0]=(double)k*CUBESIZE/size+CUBESIZE/(size*2);
           nodeCors[nodeIndex][1]=(double)j*CUBESIZE/size+CUBESIZE/(size*2);
           nodeCors[nodeIndex][2]=(double)i*CUBESIZE/size+CUBESIZE/(size*2);
```

```
            }
        }
    }
    for(i = 0;i < noNodes;i++) {
        // determine surface nodes for assigning boundry conditions
        if ((nodeCors[i][0] == 0.0) || (nodeCors[i][0] == 1.0) ||
              (nodeCors[i][1] == 0.0) || (nodeCors[i][1] == 1.0) ||
              (nodeCors[i][2] == 0.0) || (nodeCors[i][2] == 1.0))   {
            nodeTypes[i][0] = 0;
            nodeTypes[i][1] = BOUNDRY;
        }
        else  {
            nodeTypes[i][0] = ++unknowns;
            nodeTypes[i][1] = INTERNAL;
        }
    }
}

/*************************************************************************
*                              13-Node Pyramid Case                      *
*************************************************************************/

void pyr13mesh(int size) {
    int i,j,k, index, size_sq, layerOffset;
    int l,m;  // variables for testing

    size_sq = (int)pow(size, 2);
    layerOffset = 13.0*size_sq + 6.0*size + 2;
    for(i = 0; i < size; i++)   {   // each layer of cubes
        for(j = 0; j < size; j++)  {   // each row of cubes
            for(k = 0; k < size; k++)   {   // each column of cubes
                  // 1st (bottom) element of the cube
                index = (i*size*size+j*size+k)*6;
                elements[index][0] = (i+1)*layerOffset +j*(3*size+2) +2*k;
                elements[index][1] = (i+1)*layerOffset +j*(3*size+2) +2*k +1;
                elements[index][2] = (i+1)*layerOffset +j*(3*size+2) +2*k +2;
                elements[index][3] = i*layerOffset +j*(2*size+1) +k +7*size_sq
                                     + 4*size +2;
                elements[index][4] = i*layerOffset +j*(3*size+2) +2*k +2;
                elements[index][5] = i*layerOffset +j*(3*size+2) +2*k +1;
                elements[index][6] = i*layerOffset +j*(3*size+2) +2*k;
                elements[index][7] = i*layerOffset +j*(2*size+1) +k +7*size_sq
                                     + 4*size + 1;
                elements[index][8] = i*layerOffset +j*(4*size) +2*k +9*size_sq
                                     + 6*size + 2;
                elements[index][9] = i*layerOffset +j*(4*size) +2*k +9*size_sq
                                     + 6*size +3;
                elements[index][10] = i*layerOffset +j*(4*size)+2*k +3*size_sq
                                     + 4*size +2;
                elements[index][11] = i*layerOffset +j*(4*size) +2*k +3*size_sq
                                     + 4*size +1;
                elements[index][12] = i*layerOffset +j*(2*size+1) +k +7*size_sq
                                     + 5*size +2;
                   // 2nd (right-side) element of the cube
                index++;
                elements[index][0] = (i+1)*layerOffset +j*(3*size+2) +2*k+2;
                elements[index][1] = (i+1)*layerOffset +j*(3*size+2)+k+2*size+2;
                elements[index][2] = (i+1)*layerOffset +(j+1)*(3*size+2)+2*k +2;
                elements[index][3] = i*layerOffset +(j+1)*(2*size+1)+k+7*size_sq
```

```
                        + 4*size +2;
elements[index][4] = i*layerOffset +(j+1)*(3*size+2) +2*k +2;
elements[index][5] = i*layerOffset +j*(3*size+2) +k +2*size +2;
elements[index][6] = i*layerOffset +j*(3*size+2) +2*k +2;
elements[index][7] = i*layerOffset +j*(2*size+1) +k +7*size_sq
                        + 4*size +2;
elements[index][8] = i*layerOffset +j*(4*size) +2*k +9*size_sq
                        + 6*size +3;
elements[index][9] = i*layerOffset +j*(4*size) +2*k +9*size_sq
                        + 8*size +3;
elements[index][10] = i*layerOffset +j*(4*size) +2*k +3*size_sq
                        + 6*size +2;
elements[index][11] = i*layerOffset +j*(4*size) +2*k +3*size_sq
                        + 4*size +2;
elements[index][12] = i*layerOffset +j*(2*size+1) +k +7*size_sq
                        + 5*size +2;
   // 3rd (top) element of the cube
index++;
elements[index][0] = (i+1)*layerOffset +(j+1)*(3*size+2)+2*k+2;
elements[index][1] = (i+1)*layerOffset +(j+1)*(3*size+2)+2*k +1;
elements[index][2] = (i+1)*layerOffset +(j+1)*(3*size+2)+2*k;
elements[index][3] = i*layerOffset +(j+1)*(2*size+1)+k+7*size_sq
                        + 4*size + 1;
elements[index][4] = i*layerOffset+(j+1)*(3*size+2)+2*k;
elements[index][5] = i*layerOffset+(j+1)*(3*size+2)+2*k +1;
elements[index][6] = i*layerOffset+(j+1)*(3*size+2)+2*k +2;
elements[index][7] = i*layerOffset+(j+1)*(2*size+1)+k+7*size_sq
                        + 4*size +2;
elements[index][8] = i*layerOffset +j*(4*size)+2*k+9*size_sq
                        + 8*size + 3;
elements[index][9] = i*layerOffset +j*(4*size)+2*k+9*size_sq
                        + 8*size + 2;
elements[index][10] = i*layerOffset +j*(4*size)+2*k+3*size_sq
                        + 6*size + 1;
elements[index][11] = i*layerOffset +j*(4*size)+2*k+3*size_sq
                        + 6*size + 2;
elements[index][12] = i*layerOffset +j*(2*size+1)+k+7*size_sq
                        + 5*size + 2;
   // 4th (left-side) element of the cube
index++;
elements[index][0] = (i+1)*layerOffset+(j+1)*(3*size+2) +2*k;
elements[index][1] = (i+1)*layerOffset+j*(3*size+2)+k+2*size+1;
elements[index][2] = (i+1)*layerOffset+j*(3*size+2)+2*k;
elements[index][3] = i*layerOffset + j*(2*size+1)+k+7*size_sq
                        + 4*size + 1;
elements[index][4] = i*layerOffset+j*(3*size+2)+2*k;
elements[index][5] = i*layerOffset+j*(3*size+2)+k+2*size+1;
elements[index][6] = i*layerOffset+(j+1)*(3*size+2)+2*k;
elements[index][7] = i*layerOffset+(j+1)*(2*size+1)+k+7*size_sq
                        + 4*size + 1;
elements[index][8] = i*layerOffset +j*(4*size) +2*k +9*size_sq
                        + 8*size + 2;
elements[index][9] = i*layerOffset +j*(4*size) +2*k +9*size_sq
                        + 6*size + 2;
elements[index][10] = i*layerOffset +j*(4*size)+2*k+3*size_sq
                        + 4*size + 1;
elements[index][11] = i*layerOffset +j*(4*size) +2*k +3*size_sq
                        + 6*size + 1;
elements[index][12] = i*layerOffset +j*(2*size+1) +k +7*size_sq
```

```
                                + 5*size + 2;
            // 5th (frort) element of the cube
         index++;
         elements[index][0] = (i+1)*layerOffset+(j+1)*(3*size+2)+2*k;
         elements[index][1] = (i+1)*layerOffset+(j+1)*(3*size+2)+2*k +1;
         elements[index][2] = (i+1)*layerOffset+(j+1)*(3*size+2)+2*k +2;
         elements[index][3] = (i+1)*layerOffset+ j*(3*size+2)+k+2*size+2;
         elements[index][4] = (i+1)*layerOffset+ j*(3*size+2)+2*k+2;
         elements[index][5] = (i+1)*layerOffset+ j*(3*size+2)+2*k+1;
         elements[index][6] = (i+1)*layerOffset+ j*(3*size+2)+2*k;
         elements[index][7] = (i+1)*layerOffset+ j*(3*size+2)+k+2*size+1;
         elements[index][8] = i*layerOffset +j*(4*size)+2*k+9*size_sq
                                + 8*size + 2;
         elements[index][9] = i*layerOffset +j*(4*size) +2*k +9*size_sq
                                + 8*size + 3;
         elements[index][10] = i*layerOffset +j*(4*size)+2*k+9*size_sq
                                + 6*size + 3;
         elements[index][11] = i*layerOffset +j*(4*size)+2*k +9*size_sq
                                + 6*size + 2;
         elements[index][12] = i*layerOffset +j*(2*size+1) +k +7*size_sq
                                + 5*size + 2;
            // 6th (back) element of the cube
         index++;
         elements[index][0] = i*layerOffset +j*(3*size+2) +2*k;
         elements[index][1] = i*layerOffset +j*(3*size+2) +2*k +1;
         elements[index][2] = i*layerOffset +j*(3*size+2) +2*k +2;
         elements[index][3] = i*layerOffset +j*(3*size+2) +k+ 2*size +2;
         elements[index][4] = i*layerOffset +(j+1)*(3*size+2)+2*k +2;
         elements[index][5] = i*layerOffset +(j+1)*(3*size+2)+2*k +1;
         elements[index][6] = i*layerOffset +(j+1)*(3*size+2)+2*k;
         elements[index][7] = i*layerOffset +j*(3*size+2) + k+2*size+1;
         elements[index][8] = i*layerOffset +j*(4*size) +2*k+3*size_sq
                                + 4*size + 1;
         elements[index][9] = i*layerOffset+j*(4*size)+2*k+3*size_sq
                                + 4*size + 2;
         elements[index][10] = i*layerOffset+j*(4*size)+2*k+3*size_sq
                                + 6*size + 2;
         elements[index][11] = i*layerOffset+j*(4*size)+2*k+3*size_sq
                                + 6*size + 1;
         elements[index][12] = i*layerOffset+j*(2*size+1)+k+7*size_sq
                                + 5*size + 2;
      }
   }
}
}

void pyr13mapping(int size)  {
   int nodeIndex, offset, size_sq, i, j, k;

   // calculate x,y,z coordinates for corner nodes of small cubes
   size_sq = (int)pow(size, 2);
   offset = 13 * size_sq + 6 * size + 2;
   for(i = 0; i < size+1; i++)   {   // each layer of cubes
      for(j = 0; j < size+1; j++)  {   // each row of cubes
         for(k = 0; k < size+1; k++)   {   // each column of cubes
         // node 3
            nodeIndex = i*offset + j*(3*size+2) + 2*k;
            nodeCors[nodeIndex][0] = (double) k/size;
            nodeCors[nodeIndex][1] = (double) j/size;
```

```
nodeCors[nodeIndex][2] = (double) i/size;
if(i < size)   {
// node 03
   nodeIndex = i*offset + 7*size_sq+4*size+1+j*(2*size+1)+k;
   nodeCors[nodeIndex][0] = (double) k/size;
   nodeCors[nodeIndex][1] = (double) j/size;
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(2.0*size);
}
if(k < size)   {
// node 23
   nodeIndex = i*offset + j*(3*size+2) + 2*k + 1;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][1] = (double) j/size;
   nodeCors[nodeIndex][2] = (double) i/size;
}
if(j < size)   {
// node 23 in 90 degree _clockwise_ rotation
   nodeIndex = i*offset + j*(3*size+2) + k + 2*size+1;
   nodeCors[nodeIndex][0] = (double) k/size;
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][2] = (double) i/size;
}
if ((i < size) && (j < size) && (k < size))   {
// node 34
   nodeIndex = i*offset + 3*size_sq + 4*size+1+j*(4*size)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);
// node 14
   nodeIndex = i*offset +3*size_sq +4*size+1+j*(4*size)+2*k+1;
   nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);
// node 02 in 180 degree flip
   nodeIndex = i*offset +3*size_sq +6*size+1 +j*(4*size)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);
// node 03 in 180 degree flip
   nodeIndex = i*offset +3*size_sq +6*size+1 +j*(4*size)+2*k+1;
   nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);
// node 4 (centre)
   nodeIndex = i*offset + 7*size_sq + 5*size+2 + j*(2*size+1)+k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(2.0*size);
// node 04
   nodeIndex = i*offset + 9*size_sq + 6*size+2 + j*(4*size)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);
// node 14
   nodeIndex = i*offset +9*size_sq +6*size+2 +j*(4*size)+2*k+1;
   nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);
// node 14 in 180 degree flip
```

```
                nodeIndex = i*offset + 9*size_sq + 8*size+2 + j*(4*size)+2*k;
                nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
                nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
                nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);
             // node 04 in 180 degree flip
                nodeIndex = i*offset + 9*size_sq + 8*size+2+j*(4*size)+2*k+1;
                nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
                nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
                nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);
             }
          }
       }
    }
    for(i = 0;i < noNodes;i++) {
       // determine surface nodes for assigning boundry conditions
       if ((nodeCors[i][0] == 0.0) || (nodeCors[i][0] == 1.0) ||
             (nodeCors[i][1] == 0.0) || (nodeCors[i][1] == 1.0) ||
             (nodeCors[i][2] == 0.0) || (nodeCors[i][2] == 1.0))   {
          nodeTypes[i][0] = 0;
          nodeTypes[i][1] = BOUNDRY;
       }
       else  {
          nodeTypes[i][0] = ++unknowns;
          nodeTypes[i][1] = INTERNAL;
       }
    }
}
/*****************************************************************************
*                              14-Node Pyramid Case                          *
*****************************************************************************/
void pyr14mesh(int size) {
   int i,j,k, index, size_sq, layerOffset;
   int l,m;  // variables for testing

   size_sq = (int)pow(size, 2);
   layerOffset = 16.0*size_sq + 8.0*size + 2;
   for(i = 0; i < size; i++)   {   // each layer of cubes
      for(j = 0; j < size; j++)  {   // each row of cubes
         for(k = 0; k < size; k++)   {   // each column of cubes
               // 1st (bottom) element of the cube
            index = (i*size*size+j*size+k)*6;
            elements[index][0] = (i+1)*layerOffset + j*(4*size+2) +2*k;
            elements[index][1] = (i+1)*layerOffset + j*(4*size+2) +2*k +1;
            elements[index][2] = (i+1)*layerOffset + j*(4*size+2) +2*k +2;
            elements[index][3] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                                    + 4*size + 3;
            elements[index][4] = i*layerOffset+j*(4*size+2)+2*k +2;
            elements[index][5] = i*layerOffset+j*(4*size+2)+2*k +1;
            elements[index][6] = i*layerOffset+j*(4*size+2)+2*k;
            elements[index][7] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                                    + 4*size + 1;
            elements[index][8] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                                    + 4*size + 2;
            elements[index][9] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                                    + 8*size + 2;
            elements[index][10] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                                    + 8*size + 3;
            elements[index][11] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                                    + 4*size + 2;
```

```
elements[index][12] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                      + 4*size + 1;
elements[index][13] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                      + 6*size + 3;
   // 2nd (right-side) element of the cube
index++;
elements[index][0] = (i+1)*layerOffset+j*(4*size+2)+2*k+2;
elements[index][1] = (i+1)*layerOffset+j*(4*size+2)+2*k+2*size+3;
elements[index][2] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k+2;
elements[index][3] = i*layerOffset + (j+1)*(4*size+2) + 2*k
                      + 8*size_sq + 4*size + 3;
elements[index][4] = i*layerOffset+(j+1)*(4*size+2)+2*k+2;
elements[index][5] = i*layerOffset+j*(4*size+2)+2*k+2*size+3;
elements[index][6] = i*layerOffset+j*(4*size+2)+2*k+2;
elements[index][7] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                      + 4*size + 3;
elements[index][8] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                      + 6*size + 4;
elements[index][9] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                      + 8*size + 3;
elements[index][10] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                      + 10*size + 3;
elements[index][11] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                      + 6*size + 2;
elements[index][12] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                      + 4*size + 2;
elements[index][13] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                      + 6*size + 3;
   // 3rd (top) element of the cube
index++;
elements[index][0] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k+2;
elements[index][1] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k+1;
elements[index][2] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k;
elements[index][3] = i*layerOffset+(j+1)*(4*size+2) +2*k
                     + 8*size_sq +4*size +1;
elements[index][4] = i*layerOffset+(j+1)*(4*size+2)+2*k;
elements[index][5] = i*layerOffset+(j+1)*(4*size+2)+2*k+1;
elements[index][6] = i*layerOffset+(j+1)*(4*size+2)+2*k+2;
elements[index][7] = i*layerOffset+(j+1)*(4*size+2)+2*k
                     + 8*size_sq +4*size + 3;
elements[index][8] = i*layerOffset+(j+1)*(4*size+2)+2*k
                     + 8*size_sq +4*size +2;
elements[index][9] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 10*size + 3;
elements[index][10] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 10*size + 2;
elements[index][11] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                     + 6*size + 1;
elements[index][12] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                     + 6*size + 2;
elements[index][13] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                     + 6*size + 3;
   // 4th (left-side) element of the cube
index++;
elements[index][0] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k;
elements[index][1] =(i+1)*layerOffset+j*(4*size+2)+2*k+2*size+1;
elements[index][2] = (i+1)*layerOffset+j*(4*size+2)+2*k;
elements[index][3] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                     + 4*size + 1;
```

```
elements[index][4] = i*layerOffset+j*(4*size+2)+2*k;
elements[index][5] = i*layerOffset+j*(4*size+2)+2*k+2*size+1;
elements[index][6] = i*layerOffset+(j+1)*(4*size+2)+2*k;
elements[index][7] = i*layerOffset+(j+1)*(4*size+2)+2*k
                     + 8*size_sq + 4*size + 1;
elements[index][8] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                     + 6*size + 2;
elements[index][9] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 10*size + 2;
elements[index][10] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 8*size + 2;
elements[index][11] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                     + 4*size + 1;
elements[index][12] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                     + 6*size + 1;
elements[index][13] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                     + 6*size + 3;
   // 5th (front) element of the cube
index++;
elements[index][0] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k;
elements[index][1] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k+1;
elements[index][2] = (i+1)*layerOffset+(j+1)*(4*size+2)+2*k+2;
elements[index][3] = (i+1)*layerOffset+j*(4*size+2) +2*k
                     + 2*size + 3;
elements[index][4] = (i+1)*layerOffset+j*(4*size+2)+2*k+2;
elements[index][5] = (i+1)*layerOffset+j*(4*size+2)+2*k+1;
elements[index][6] = (i+1)*layerOffset+j*(4*size+2)+2*k;
elements[index][7] = (i+1)*layerOffset+j*(4*size+2)+2*k
                     + 2*size+1;
elements[index][8] = (i+1)*layerOffset+j*(4*size+2)+2*k
                     + 2*size + 2;
elements[index][9] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 10*size + 2;
elements[index][10] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 10*size + 3;
elements[index][11] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 8*size + 3;
elements[index][12] = i*layerOffset+j*(4*size)+2*k+12*size_sq
                     + 8*size + 2;
elements[index][13] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                     + 6*size + 3;
   // 6th (back) element of the cube
index++;
elements[index][0] = i*layerOffset+j*(4*size+2)+2*k;
elements[index][1] = i*layerOffset+j*(4*size+2)+2*k+1;
elements[index][2] = i*layerOffset+j*(4*size+2)+2*k+2;
elements[index][3] = i*layerOffset+j*(4*size+2)+2*k+2*size+3;
elements[index][4] = i*layerOffset+(j+1)*(4*size+2)+2*k+2;
elements[index][5] = i*layerOffset+(j+1)*(4*size+2)+2*k+1;
elements[index][6] = i*layerOffset+(j+1)*(4*size+2)+2*k;
elements[index][7] = i*layerOffset+j*(4*size+2)+2*k+2*size+1;
elements[index][8] = i*layerOffset+j*(4*size+2)+2*k+2*size+2;
elements[index][9] = i*layerOffset+j*(4*size)+ 2*k+4*size_sq
                     + 4*size + 1;
elements[index][10] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                     + 4*size + 2;
elements[index][11] = i*layerOffset+j*(4*size)+2*k+4*size_sq
                     + 6*size + 2;
elements[index][12] = i*layerOffset+j*(4*size)+2*k+4*size_sq
```

```
                                    + 6*size + 1;
            elements[index][13] = i*layerOffset+j*(4*size+2)+2*k+8*size_sq
                                    + 6*size + 3;
         }
      }
   }
}

void pyr14mapping(int size)  {
   int nodeIndex, offset, size_sq, i, j, k;

   // calculate x,y,z coordinates for corner nodes of small cubes
   size_sq = (int)pow(size, 2);
   offset = 16 * size_sq + 8 * size + 2;
   for(i = 0; i < size+1; i++)   {   // each layer of cubes
      for(j = 0; j < size+1; j++)  {   // each row of cubes
         for(k = 0; k < size+1; k++)   {   // each column of cubes
         // node 3 (node 6)
            nodeIndex = i*offset + j*(4*size+2) + 2*k;
            nodeCors[nodeIndex][0] = (double) k/size;
            nodeCors[nodeIndex][1] = (double) j/size;
            nodeCors[nodeIndex][2] = (double) i/size;
            if(i < size)   {
            // node 03 (node 7)
               nodeIndex = i*offset+8*size_sq+4*size+1+j*(4*size+2)+2*k;
               nodeCors[nodeIndex][0] = (double) k/size;
               nodeCors[nodeIndex][1] = (double) j/size;
               nodeCors[nodeIndex][2] = (double) i/size +1.0/(2.0*size);
            }
            if(k < size)   {
            // node 23 (node 5)
               nodeIndex = i*offset + j*(4*size+2) + 2*k + 1;
               nodeCors[nodeIndex][0] = (double) k/size +1.0/(2.0*size);
               nodeCors[nodeIndex][1] = (double) j/size;
               nodeCors[nodeIndex][2] = (double) i/size;
            }
            if(j < size)   {
            // node 23 in 90 degree _clockwise_ rotation (node 5, elm #3)
               nodeIndex = i*offset + j*(4*size+2) + 2*k + 2*size+1;
               nodeCors[nodeIndex][0] = (double) k/size;
               nodeCors[nodeIndex][1] = (double) j/size + 1.0/(2.0*size);
               nodeCors[nodeIndex][2] = (double) i/size;
            }
            if ((i < size) && (k < size))   {
            // node 02 (node 8)
               nodeIndex = i*offset +8*size_sq+4*size+2+j*(4*size+2)+2*k;
               nodeCors[nodeIndex][0] = (double) k/size + 1.0/(2.0*size);
               nodeCors[nodeIndex][1] = (double) j/size;
               nodeCors[nodeIndex][2] = (double) i/size + 1.0/(2.0*size);
            }
            if ((i < size) && (j < size))   {
            // node 02 in 90 degree clockwise rotation about z(node 8,elm #3)
               nodeIndex = i*offset + 8*size_sq +6*size+2+j*(4*size+2)+2*k;
               nodeCors[nodeIndex][0] = (double) k/size;
               nodeCors[nodeIndex][1] = (double) j/size + 1.0/(2.0*size);
               nodeCors[nodeIndex][2] = (double) i/size + 1.0/(2.0*size);
            }
            if ((k < size) && (j < size))   {
            // node 02 in 90 degree clockwise rotation about x
```

```
// (node 8,elm #5 (back))
   nodeIndex = i*offset + 2*size+2 + j*(4*size+2)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][2] = (double) i/size;
}
if ((i < size) && (j < size) && (k < size))   {
// node 34 (node 12)
   nodeIndex = i*offset +4*size_sq +4*size+1+j*(4*size)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);

// node 24 (node 11)
   nodeIndex = i*offset +4*size_sq +4*size+1+j*(4*size)+2*k+1;
   nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);

// node 04 in 180 degree flip (node 11, ele 2)
   nodeIndex = i*offset +4*size_sq +6*size+1 +j*(4*size)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);

 // node 03 in 180 degree flip (node 12, elm 2)
   nodeIndex = i*offset+4*size_sq +6*size+1 +j*(4*size)+2*k+1;
   nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(4.0*size);

// node 4 (centre, node 13)
   nodeIndex = i*offset +8*size_sq+6*size+3+j*(4*size+2)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(2.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 1.0/(2.0*size);

// node 04 (node 9)
   nodeIndex = i*offset +12*size_sq +8*size+2 +j*(4*size)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);

 // node 14 (node 10)
   nodeIndex = i*offset +12*size_sq+8*size+2+j*(4*size)+2*k+1;
   nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);

 // node 14 in 180 degree flip (node 10, elm 2)
   nodeIndex = i*offset +12*size_sq+10*size+2+j*(4*size)+2*k;
   nodeCors[nodeIndex][0] = (double) k/size + 1.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);

 // node 04 in 180 degree flip (node 9, elm 2)
   nodeIndex = i*offset+12*size_sq+10*size+2+j*(4*size)+2*k+1;
   nodeCors[nodeIndex][0] = (double) k/size + 3.0/(4.0*size);
   nodeCors[nodeIndex][1] = (double) j/size + 3.0/(4.0*size);
```

```
                nodeCors[nodeIndex][2] = (double) i/size + 3.0/(4.0*size);
            }
         }
      }
   }
   for(i = 0;i < noNodes;i++) {
      // determine surface nodes for assigning boundry conditions
      if ((nodeCors[i][0] == 0.0) || (nodeCors[i][0] == 1.0) ||
            (nodeCors[i][1] == 0.0) || (nodeCors[i][1] == 1.0) ||
            (nodeCors[i][2] == 0.0) || (nodeCors[i][2] == 1.0))   {
         nodeTypes[i][0] = 0;
         nodeTypes[i][1] = BOUNDRY;
      }
      else  {
         nodeTypes[i][0] = ++unknowns;
         nodeTypes[i][1] = INTERNAL;
      }
   }
}

/*************************************************************************
*
*  Control function, calls appropriate mesh generation functions
*
*************************************************************************/
void makeMesh(int size) {

   if(pyrType == PYR5) {
      pyr5mesh(size);
      pyr5mapping(size);
   }
   else  {
      if(pyrType == PYR13)   {
         pyr13mesh(size);
         pyr13mapping(size);
      }
      else  {
         if((pyrType == PYR14_CASE_I) ||(pyrType == PYR14_CASE_II))  {
            pyr14mesh(size);
            pyr14mapping(size);
         }
         else  {
            printf("%s\n",
            "Error in: makeMesh(), Invalid mesh type for mesh generation.");
            printf("Program Halted...\n");
            exit(1);
         }
      }
   }
}
```

```
/*************************************************************************
* Author: Kevin Davies
* File entry_intgrls.c
* Version: 4.0
* Date last modified: Dec 15, 2004
* Description: functions for calculating integral values for system
*     matrix and load vector entries.
*************************************************************************/

/*************************************************************************
*  Function to get vertex points of tetrahedra
*************************************************************************/
void getTetPoints(MAT *A, int i)   {

   switch(i)   {
      case 1:
         // set vertex coordinates for tetrahedron #1
         if(symBasis)   {
            A->me[0][0] = -1.0;
            A->me[1][0] = -1.0;
            A->me[2][0] =  0.0;

            A->me[0][1] =  1.0;
            A->me[1][1] = -1.0;
            A->me[2][1] =  0.0;

            A->me[0][2] =  0.0;
            A->me[1][2] =  0.0;
            A->me[2][2] =  1.0;

            A->me[0][3] =  0.0;
            A->me[1][3] =  0.0;
            A->me[2][3] =  0.0;
         }
         else   {
            A->me[0][0] =  1.0;
            A->me[1][0] = -1.0;
            A->me[2][0] =  0.0;

            A->me[0][1] =  1.0;
            A->me[1][1] =  1.0;
            A->me[2][1] =  0.0;

            A->me[0][2] =  0.0;
            A->me[1][2] =  0.0;
            A->me[2][2] =  1.0;

            A->me[0][3] = -1.0;
            A->me[1][3] = -1.0;
            A->me[2][3] =  0.0;
         }
         break;
      case 2:
         // set vertex coordinates for tetrahedron #2
         if(symBasis)   {
            A->me[0][0] =  1.0;
            A->me[1][0] = -1.0;
            A->me[2][0] =  0.0;
```

```
            A->me[0][1] =  1.0;
            A->me[1][1] =  1.0;
            A->me[2][1] =  0.0;

            A->me[0][2] =  0.0;
            A->me[1][2] =  0.0;
            A->me[2][2] =  1.0;

            A->me[0][3] =  0.0;
            A->me[1][3] =  0.0;
            A->me[2][3] =  0.0;
        }
        else   {
            A->me[0][0] =  1.0;
            A->me[1][0] =  1.0;
            A->me[2][0] =  0.0;

            A->me[0][1] = -1.0;
            A->me[1][1] =  1.0;
            A->me[2][1] =  0.0;

            A->me[0][2] =  0.0;
            A->me[1][2] =  0.0;
            A->me[2][2] =  1.0;

            A->me[0][3] = -1.0;
            A->me[1][3] = -1.0;
            A->me[2][3] =  0.0;
        }
        break;
    case 3:
        // set vertex coordinates for tetrahedron #3
        A->me[0][0] =  1.0;
        A->me[1][0] =  1.0;
        A->me[2][0] =  0.0;

        A->me[0][1] = -1.0;
        A->me[1][1] =  1.0;
        A->me[2][1] =  0.0;

        A->me[0][2] =  0.0;
        A->me[1][2] =  0.0;
        A->me[2][2] =  1.0;

        A->me[0][3] =  0.0;
        A->me[1][3] =  0.0;
        A->me[2][3] =  0.0;
        break;
    case 4:
        // set vertex coordinates for tetrahedron #4
        A->me[0][0] = -1.0;
        A->me[1][0] =  1.0;
        A->me[2][0] =  0.0;

        A->me[0][1] = -1.0;
        A->me[1][1] = -1.0;
        A->me[2][1] =  0.0;

        A->me[0][2] =  0.0;
```

```
            A->me[1][2] =  0.0;
            A->me[2][2] =  1.0;

            A->me[0][3] =  0.0;
            A->me[1][3] =  0.0;
            A->me[2][3] =  0.0;
            break;
         default:
            printf("Invalid tet #\n");
      }
   }

/*****************************************************************************
% Author: Kevin Davies
% Description: function to find jacobian matrix, ie. N*X
% inputs:
%   X - a matrix of global coordinates, s.t.
%       the first column consists of the x coordinate values, the second
%       column consists of y values, and the third consists of z values.
% outputs:
%   J - Jacobian matrix
%
*****************************************************************************/
void jacob(MAT *X, MAT *J)   {
   MAT *N, *A;
   VEC *Q;
   int m;
   double x,y,z, wi, h = 0.000001;
   N = m_get(3,nodesPerElm);
   A = m_get(3,4);
   Q = v_get(3);

   getTetPoints(A, 1);
   switch(pyrType)   {
   case PYR5:
      cubature4(A, 1, Q, &wi);
      break;
   case PYR13:
      cubature4(A, 1, Q, &wi);
      break;
   case PYR14_CASE_I:
      cubature4(A, 1, Q, &wi);
      break;
   case PYR14_CASE_II:
      cubature4(A, 1, Q, &wi);
   }
   x = Q->ve[0];
   y = Q->ve[1];
   z = Q->ve[2];
   if(!symBasis)   {
      switch(pyrType)   {
      case PYR5:
         for( m = 0; m < nodesPerElm; m++)   {
            N->me[0][m] = pyr5basis_dx(m,x,y,z);
            N->me[1][m] = pyr5basis_dy(m,x,y,z);
            N->me[2][m] = pyr5basis_dz(m,x,y,z);
         }
         break;
      case PYR13:
```

```
            for( m = 0; m < nodesPerElm; m++)    {
               N->me[0][m] = pyr13basis_dx(m,x,y,z);
               N->me[1][m] = pyr13basis_dy(m,x,y,z);
               N->me[2][m] = pyr13basis_dz(m,x,y,z);
            }
            break;
         case PYR14_CASE_I:  // Approximations to Partial dirivatives
            for( m = 0; m < nodesPerElm; m++)    {
               N->me[0][m] = (pyr14C1basis(m,x+h,y,z)-pyr14C1basis(m,x,y,z))/h;
               N->me[1][m] = (pyr14C1basis(m,x,y+h,z)-pyr14C1basis(m,x,y,z))/h;
               N->me[2][m] = (pyr14C1basis(m,x,y,z+h)-pyr14C1basis(m,x,y,z))/h;
            }
            break;
         case PYR14_CASE_II:
            for( m = 0; m < nodesPerElm; m++)    {
               N->me[0][m] = pyr14C2basis_dx(m,x,y,z);
               N->me[1][m] = pyr14C2basis_dy(m,x,y,z);
               N->me[2][m] = pyr14C2basis_dz(m,x,y,z);
            }
         }
      }
      else   {
         switch(pyrType)    {
         case PYR5:
            for( m = 0; m < nodesPerElm; m++)    {
               N->me[0][m] = pyr5symBasis_dx(m,x,y,z);
               N->me[1][m] = pyr5symBasis_dy(m,x,y,z);
               N->me[2][m] = pyr5symBasis_dz(m,x,y,z);
            }
            break;
         case PYR13:
            for( m = 0; m < nodesPerElm; m++)    {
               N->me[0][m] = pyr13symBasis_dx(m,x,y,z);
               N->me[1][m] = pyr13symBasis_dy(m,x,y,z);
               N->me[2][m] = pyr13symBasis_dz(m,x,y,z);
            }
            break;
         case PYR14_CASE_I:  // Approximations to Partial dirivatives
          for( m = 0; m < nodesPerElm; m++)    {
            N->me[0][m]=(pyr14C1symBasis(m,x+h,y,z)-pyr14C1symBasis(m,x,y,z))/h;
            N->me[1][m]=(pyr14C1symBasis(m,x,y+h,z)-pyr14C1symBasis(m,x,y,z))/h;
            N->me[2][m]=(pyr14C1symBasis(m,x,y,z+h)-pyr14C1symBasis(m,x,y,z))/h;
          }
          break;
         case PYR14_CASE_II:
            for( m = 0; m < nodesPerElm; m++)    {
               N->me[0][m] = pyr14C2symBasis_dx(m,x,y,z);
               N->me[1][m] = pyr14C2symBasis_dy(m,x,y,z);
               N->me[2][m] = pyr14C2symBasis_dz(m,x,y,z);
            }
         }
      }
      m_mlt(N, X, J);
      M_FREE(N);
      M_FREE(A);
      V_FREE(Q);
}
```

```
/*************************************************************************
* Author: Kevin Davies
* Description: function for integrating over tetrahedra for a system
*     matrix entry.
* inputs:
*   X - a matrix of global coordinates, s.t.
*       the first column consists of the x coordinate values, the second
*       column consists of y values, and the third consists of z values.
*   i - index of first node in element
*   j - index of second node in element
*   tetNo - tetrahedra index
* outputs:
*   returns quadrature over tetrhedral portion of element
*
*************************************************************************/

double tet(MAT *X, int i, int j, int tetNo)   {
   MAT *N, *A, *K, *J, *Jinv;
   VEC *Ni, *Nj, *v, *Q;
   int k,m;
   double x,y,z, t = 0.0, wi, quadPoints, h = 0.000001;
   N = m_get(3,nodesPerElm);
   A = m_get(3,4);
   K = m_get(1,3);
   J = m_get(3,3);
   Jinv = m_get(3,3);
   Ni = v_get(3);
   Nj = v_get(3);
   v = v_get(3);
   Q = v_get(3);

   switch(pyrType)   {
      case PYR5:
         quadPoints = quadDeg4pts;
         break;
      case PYR13:
         quadPoints = quadDeg4pts;
         break;
      case PYR14_CASE_I:
         quadPoints = quadDeg4pts;
         break;
      case PYR14_CASE_II:
         quadPoints = quadDeg4pts;
   }
   getTetPoints(A, tetNo);
   for(k = 0; k < quadPoints; k++)  {
      switch(pyrType)   {
      case PYR5:
         cubature4(A, k+1, Q, &wi);
         break;
      case PYR13:
         cubature4(A, k+1, Q, &wi);
         break;
      case PYR14_CASE_I:
         cubature4(A, k+1, Q, &wi);
         break;
      case PYR14_CASE_II:
         cubature4(A, k+1, Q, &wi);
```

```
}
x = Q->ve[0];
y = Q->ve[1];
z = Q->ve[2];
if(!symBasis)   {
   switch(pyrType)   {
   case PYR5:
      for( m = 0; m < nodesPerElm; m++)   {
         N->me[0][m] = pyr5basis_dx(m,x,y,z);
         N->me[1][m] = pyr5basis_dy(m,x,y,z);
         N->me[2][m] = pyr5basis_dz(m,x,y,z);
      }
      break;
   case PYR13:
      for( m = 0; m < nodesPerElm; m++)   {
         N->me[0][m] = pyr13basis_dx(m,x,y,z);
         N->me[1][m] = pyr13basis_dy(m,x,y,z);
         N->me[2][m] = pyr13basis_dz(m,x,y,z);
      }
      break;
   case PYR14_CASE_I:  // Approximations to Partial dirivatives
     for( m = 0; m < nodesPerElm; m++)   {
       N->me[0][m]=(pyr14C1basis(m,x+h,y,z)-pyr14C1basis(m,x,y,z))/h;
       N->me[1][m]=(pyr14C1basis(m,x,y+h,z)-pyr14C1basis(m,x,y,z))/h;
       N->me[2][m]=(pyr14C1basis(m,x,y,z+h)-pyr14C1basis(m,x,y,z))/h;
     }
     break;
   case PYR14_CASE_II:
      for( m = 0; m < nodesPerElm; m++)   {
         N->me[0][m] = pyr14C2basis_dx(m,x,y,z);
         N->me[1][m] = pyr14C2basis_dy(m,x,y,z);
         N->me[2][m] = pyr14C2basis_dz(m,x,y,z);
      }
   }
}
else   {
   switch(pyrType)   {
   case PYR5:
      for( m = 0; m < nodesPerElm; m++)   {
         N->me[0][m] = pyr5symBasis_dx(m,x,y,z);
         N->me[1][m] = pyr5symBasis_dy(m,x,y,z);
         N->me[2][m] = pyr5symBasis_dz(m,x,y,z);
      }
      break;
   case PYR13:
      for( m = 0; m < nodesPerElm; m++)   {
         N->me[0][m] = pyr13symBasis_dx(m,x,y,z);
         N->me[1][m] = pyr13symBasis_dy(m,x,y,z);
         N->me[2][m] = pyr13symBasis_dz(m,x,y,z);
      }
      break;
   case PYR14_CASE_I:  // Approximations to Partial dirivatives
      for( m = 0; m < nodesPerElm; m++)   {
         N->me[0][m] = (pyr14C1symBasis(m,x+h,y,z) -
                        pyr14C1symBasis(m,x,y,z)) / h;
         N->me[1][m] = (pyr14C1symBasis(m,x,y+h,z) -
                        pyr14C1symBasis(m,x,y,z)) / h;
         N->me[2][m] = (pyr14C1symBasis(m,x,y,z+h) -
                        pyr14C1symBasis(m,x,y,z)) / h;
```

```
            }
            break;
         case PYR14_CASE_II:
            for( m = 0; m < nodesPerElm; m++)   {
               N->me[0][m] = pyr14C2symBasis_dx(m,x,y,z);
               N->me[1][m] = pyr14C2symBasis_dy(m,x,y,z);
               N->me[2][m] = pyr14C2symBasis_dz(m,x,y,z);
            }
         }
      }
      m_mlt(N, X, J);
      m_inverse(J, Jinv);
      get_col(N, i, v);
      mv_mlt(Jinv, v, Ni);
      get_col(N, j, v);
      mv_mlt(Jinv, v, Nj);
      t =t+wi*(Ni->ve[0]*Nj->ve[0]+Ni->ve[1]*Nj->ve[1]+Ni->ve[2]*Nj->ve[2]);
   }
   M_FREE(N);
   M_FREE(A);
   M_FREE(K);
   M_FREE(J);
   M_FREE(Jinv);
   V_FREE(Ni);
   V_FREE(Nj);
   V_FREE(Q);
   V_FREE(v);
   return t;
}

/*****************************************************************************
% Description: function for forming rhs vector from tetrahedra
% inputs:
%   X - a matrix of global coordinates, s.t.
%       the first column consists of the x coordinate values, the second
%       column consists of y values, and the third consists of z values.
%   i - element node index(0-4)
%   j - tetrahedra index(1-4)
% outputs:
%   returns quadrature over j-th tetrhedral portion of element
%
*****************************************************************************/
double rhs_tet(MAT *X, int i, int j)   {
   MAT *N, *A, *K;
   VEC *Q;
   int k, m;
   double t = 0.0, wi, quadPoints;
   N = m_get(1,nodesPerElm);
   A = m_get(3,4);
   K = m_get(1,3);
   Q = v_get(3);

   switch(pyrType)   {
      case PYR5:
         quadPoints = quadDeg4pts;
         break;
      case PYR13:
         quadPoints = quadDeg7pts;
         break;
```

```
   case PYR14_CASE_I:
      quadPoints = quadDeg7pts;
      break;
   case PYR14_CASE_II:
      quadPoints = quadDeg7pts;
}
getTetPoints(A, j);
for(k = 0; k < quadPoints ; k++)  {
   switch(pyrType)   {
   case PYR5:
      cubature4(A, k+1, Q, &wi);
      break;
   case PYR13:
      cubature7(A, k+1, Q, &wi);
      break;
   case PYR14_CASE_I:
      cubature7(A, k+1, Q, &wi);
      break;
   case PYR14_CASE_II:
      cubature7(A, k+1, Q, &wi);
   }
   if(symBasis)   {
      switch(pyrType)   {
      case PYR5:
         for(m = 1; m <= nodesPerElm; m++)
            N->me[0][m-1] = pyr5symBasis(m-1, Q->ve[0],
                                         Q->ve[1], Q->ve[2]);
         break;
      case PYR13:
         for(m = 1; m <= nodesPerElm; m++)
            N->me[0][m-1] = pyr13symBasis(m-1, Q->ve[0],
                                          Q->ve[1], Q->ve[2]);
         break;
      case PYR14_CASE_I:
         for(m = 1; m <= nodesPerElm; m++)
            N->me[0][m-1] = pyr14C1symBasis(m-1, Q->ve[0],
                                            Q->ve[1], Q->ve[2]);
         break;
      case PYR14_CASE_II:
         for(m = 1; m <= nodesPerElm; m++)
            N->me[0][m-1] = pyr14C2symBasis(m-1, Q->ve[0],
                                            Q->ve[1], Q->ve[2]);
      }
   }
   else  {
      switch(pyrType)   {
      case PYR5:
         for(m = 1; m <= nodesPerElm; m++)
            N->me[0][m-1] = pyr5basis(m-1, Q->ve[0],
                                      Q->ve[1], Q->ve[2]);
         break;
      case PYR13:
         for(m = 1; m <= nodesPerElm; m++)
            N->me[0][m-1] = pyr13basis(m-1, Q->ve[0],
                                       Q->ve[1], Q->ve[2]);
         break;
      case PYR14_CASE_I:
         for(m = 1; m <= nodesPerElm; m++)
            N->me[0][m-1] = pyr14C1basis(m-1, Q->ve[0],
```

```
                                                Q->ve[1], Q->ve[2]);
            break;
         case PYR14_CASE_II:
            for(m = 1; m <= nodesPerElm; m++)
               N->me[0][m-1] = pyr14C2basis(m-1, Q->ve[0],
                                                Q->ve[1], Q->ve[2]);
         }
      }
      m_mlt(N, X, K);
      if(symBasis)   {
         switch(pyrType)   {
         case PYR5:
            t = t + wi* ( pyr5symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2]) *
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
            break;
         case PYR13:
            t = t + wi* ( pyr13symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2])*
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
            break;
         case PYR14_CASE_I:
            t = t + wi* ( pyr14C1symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2])*
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
            break;
         case PYR14_CASE_II:
            t = t + wi* ( pyr14C2symBasis(i, Q->ve[0], Q->ve[1], Q->ve[2])*
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
         }
      }
      else  {
         switch(pyrType)   {
         case PYR5:
            t = t + wi* ( pyr5basis(i, Q->ve[0], Q->ve[1], Q->ve[2]) *
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
            break;
         case PYR13:
            t = t + wi* ( pyr13basis(i, Q->ve[0], Q->ve[1], Q->ve[2]) *
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
            break;
         case PYR14_CASE_I:
            t = t + wi* ( pyr14C1basis(i, Q->ve[0], Q->ve[1], Q->ve[2]) *
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
            break;
         case PYR14_CASE_II:
            t = t + wi* ( pyr14C2basis(i, Q->ve[0], Q->ve[1], Q->ve[2]) *
                        func(K->me[0][0], K->me[0][1], K->me[0][2]) );
         }
      }
   }
   M_FREE(N);
   M_FREE(A);
   M_FREE(K);
   V_FREE(Q);
   return t;
}
```

```
/*************************************************************************
* Programmer: Kevin Davies
* File: pyr5basis.c
* Version: 2.3
* Date last modified: Aug 25, 2004
* Description: Linear basis functions and their partial derivatives.
*
*************************************************************************/

/*************************************************************************
* Basis functions based on transformation of Wieners' basis functions.
*  (Before symmetries)
*************************************************************************/
double pyr5basis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      if(x > y)
         f = 0.25*(x-z-1.0)*(y-z-1.0) + 0.5*z*(y-z-1.0);
      else
         f = 0.25*(x-z-1.0)*(y-z-1.0) + 0.5*z*(x-z-1.0);
      break;
   case 1:
      if(x > y)
         f = 0.25*(x-z+1.0)*(-y+z+1.0) - 0.5*z*(y-z+1.0);
      else
         f = 0.25*(x-z+1.0)*(-y+z+1.0) - 0.5*z*(x-z+1.0);
      break;
   case 2:
      if(x > y)
         f = 0.25*(x-z+1.0)*(y-z+1.0) + 0.5*z*(y-z+1.0);
      else
         f = 0.25*(x-z+1.0)*(y-z+1.0) + 0.5*z*(x-z+1.0);
      break;
   case 3:
      if(x > y)
         f = 0.25*(-x+z+1.0)*(y-z+1.0) - 0.5*z*(y-z+1.0);
      else
         f = 0.25*(-x+z+1.0)*(y-z+1.0) - 0.5*z*(x-z+1.0);
      break;
   case 4:
      f = z;
      break;
   default:
      printf("invalid case detected\n");
   }
   return f;
}

double pyr5basis_dx(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      if(x > y)
         f = 0.25*(y-z-1.0);
      else
         f = 0.25*(y+z-1.0);
```

```
      break;
   case 1:
      if(x > y)
         f = 0.25*(-y+z+1.0);
      else
         f = 0.25*(-y-z+1.0);
      break;
   case 2:
      if(x > y)
         f = 0.25*(y-z+1.0);
      else
         f = 0.25*(y+z+1.0);
      break;
   case 3:
      if(x > y)
         f = 0.25*(-y+z-1.0);
      else
         f = 0.25*(-y-z-1.0);
      break;
   case 4:
      f = 0.0;
      break;
   default:
      printf("invalid case detected\n");
   }
   return f;
}

double pyr5basis_dy(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      if(x > y)
         f = 0.25*(x+z-1.0);
      else
         f = 0.25*(x-z-1.0);
      break;
   case 1:
      if(x > y)
         f = 0.25*(-x-z-1.0);
      else
         f = 0.25*(-x+z-1.0);
      break;
   case 2:
      if(x > y)
         f = 0.25*(x+z+1.0);
      else
         f = 0.25*(x-z+1.0);
      break;
   case 3:
      if(x > y)
         f = 0.25*(-x-z+1.0);
      else
         f = 0.25*(-x+z+1.0);
      break;
   case 4:
      f = 0.0;
      break;
```

```
   default:
      printf("invalid case detected\n");
   }
   return f;
}

double pyr5basis_dz(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      if(x > y)
         f = 0.25*(y-x) - 0.5*z;
      else
         f = 0.25*(x-y) - 0.5*z;
      break;
   case 1:
      if(x > y)
         f = 0.25*(-y+2.0*z+x-2.0);
      else
         f = 0.25*(y+2.0*z-x-2.0);
      break;
   case 2:
      if(x > y)
         f = 0.25*(y-2.0*z-x);
      else
         f = 0.25*(-y-2.0*z+x);
      break;
   case 3:
      if(x > y)
         f = 0.25*(-y+2.0*z+x-2.0);
      else
         f = 0.25*(y+2.0*z-x-2.0);
      break;
   case 4:
      f = 1.0;
      break;
   default:
      printf("invalid case detected\n");
   }
   return f;
}

/*************************************************************************
* Basis functions and partial derivatives
* with symmetries about x and y axis
*************************************************************************/

/*************************************************************************
* Basis functions and partial derivatives
* with symmetries about x axis
*************************************************************************/

double pyr5symBasis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      f = pyr5basis(0,x,y,z) + pyr5basis(1,-x,y,z);
```

```
      f = f/2.0;
      break;
   case 1:
      f = pyr5basis(1,x,y,z) + pyr5basis(0,-x,y,z);
      f = f/2.0;
      break;
   case 2:
      f = pyr5basis(2,x,y,z) + pyr5basis(3,-x,y,z);
      f = f/2.0;
      break;
   case 3:
      f = pyr5basis(3,x,y,z) + pyr5basis(2,-x,y,z);
      f = f/2.0;
      break;
   case 4:
      f = pyr5basis(4,x,y,z);
      break;
   default:
      printf("invalid case detected\n");
   }
   return f;
}

double pyr5symBasis_dx(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      f = pyr5basis_dx(0,x,y,z) - pyr5basis_dx(1,-x,y,z);
      f = f/2.0;
      break;
   case 1:
      f = pyr5basis_dx(1,x,y,z) - pyr5basis_dx(0,-x,y,z);
      f = f/2.0;
      break;
   case 2:
      f = pyr5basis_dx(2,x,y,z) - pyr5basis_dx(3,-x,y,z);
      f = f/2.0;
      break;
   case 3:
      f = pyr5basis_dx(3,x,y,z) - pyr5basis_dx(2,-x,y,z);
      f = f/2.0;
      break;
   case 4:
      f = pyr5basis_dx(4,x,y,z);
      break;
   default:
      printf("invalid case detected\n");
   }
   return f;
}
double pyr5symBasis_dy(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      f = pyr5basis_dy(0,x,y,z) + pyr5basis_dy(1,-x,y,z);
      f = f/2.0;
      break;
```

```
   case 1:
      f = pyr5basis_dy(1,x,y,z) + pyr5basis_dy(0,-x,y,z);
      f = f/2.0;
      break;
   case 2:
      f = pyr5basis_dy(2,x,y,z) + pyr5basis_dy(3,-x,y,z);
      f = f/2.0;
      break;
   case 3:
      f = pyr5basis_dy(3,x,y,z) + pyr5basis_dy(2,-x,y,z);
      f = f/2.0;
      break;
   case 4:
      f = pyr5basis_dy(4,x,y,z);
      break;
   default:
      printf("invalid case detected\n");
   }
   return f;
}

double pyr5symBasis_dz(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:
      f = pyr5basis_dz(0,x,y,z) + pyr5basis_dz(1,-x,y,z);
      f = f/2.0;
      break;
   case 1:
      f = pyr5basis_dz(1,x,y,z) + pyr5basis_dz(0,-x,y,z);
      f = f/2.0;
      break;
   case 2:
      f = pyr5basis_dz(2,x,y,z) + pyr5basis_dz(3,-x,y,z);
      f = f/2.0;
      break;
   case 3:
      f = pyr5basis_dz(3,x,y,z) + pyr5basis_dz(2,-x,y,z);
      f = f/2.0;
      break;
   case 4:
      f = pyr5basis_dz(4,x,y,z);
      break;
   default:
      printf("invalid case detected\n");
   }
   return f;
}
```

```
/*********************************************************************
* Programmer: Kevin Davies
* File: pyr13basis.c
* Version: 3.3
* Date last modified: Aug 25, 2004
* Description: Pyr-13 Quadratic basis functions and their partial
*      derivatives.
*
*********************************************************************/

/*********************************************************************
* Basis functions based on transformation of Wieners' basis functions.
*  (Before symmetries)
*********************************************************************/
double pyr13basis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      if(x > y)
         f = 0.25*(x+y+1.0)*(x+z-1.0)*(-y+z+1.0);
      else
         f = 0.25*(x+y+1.0)*(-x+z+1.0)*(y+z-1.0);
      break;
   case 2:   // node 1
      if(x > y)
         f = 0.25*(x-y-1.0)*((x+z+1.0)*(-y+z+1.0) - 4.0*z);
      else
         f = 0.25*(x-y-1.0)*(x-z+1.0)*(-y-z+1.0);
      break;
   case 4:   // node 2
      if(x > y)
         f = 0.25*(x+y-1.0)*(x+z+1.0)*(y-z+1.0);
      else
         f = 0.25*(x+y-1.0)*(x-z+1.0)*(y+z+1.0);
      break;
   case 6:   // node 3
      if(x > y)
         f = 0.25*(x-y+1.0)*(x+z-1.0)*(y-z+1.0);
      else
         f = 0.25*(x-y+1.0)*((x-z-1.0)*(y+z+1.0) + 4.0*z);
      break;
   case 12:   // node 4
         f = z*(2.0*z-1.0);
      break;
   case 1:   // midpoint node, between corner nodes 1 and 0
      if(x > y)
         f = 0.5*(x+z-1.0)*((y-z-1.0)*(x+1.0) + 2.0*z);
      else
         f = 0.5*(x-z+1.0)*(y+z-1.0)*(x-1.0);
      break;
   case 3:   // midpoint, between corner nodes 1 and 2
      if(x > y)
         f = -0.5*(y-z+1.0)*((x+z+1.0)*(y-1.0) + 2.0*z);
      else
         f = -0.5*(x-z+1.0)*(y+z-1.0)*(y+1.0);
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      if(x > y)
```

```
            f = -0.5*(y-z+1.0)*(x+z-1.0)*(x+1.0);
         else
            f = -0.5*(x-z+1.0)*((y+z+1.0)*(x-1.0) +2.0*z);
         break;
      case 7:   // midpoint, between corner nodes 3 and 0
         if(x > y)
            f = 0.5*(y-z+1.0)*(x+z-1.0)*(y-1.0);
         else
            f = 0.5*(y+z-1.0)*((x-z-1.0)*(y+1.0) +2.0*z);
         break;
      case 8:   // midpoint, between corner nodes 4 and 0
         if(x > y)
            f = z*(y-z-1.0)*(x+z-1.0);
         else
            f = z*(x-z-1.0)*(y+z-1.0);
         break;
      case 9:   // midpoint, between corner nodes 4 and 1
         if(x > y)
            f = -z*((x+z+1.0)*(y-z-1.0) + 4.0*z);
         else
            f = -z*(x-z+1.0)*(y+z-1.0);
         break;
      case 10:   // midpoint, between corner nodes 4 and 2
         if(x > y)
            f = z*(y-z+1.0)*(x+z+1.0);
         else
            f = z*(x-z+1.0)*(y+z+1.0);
         break;
      case 11:   // midpoint, between corner nodes 4 and 3
         if(x > y)
            f = -z*(y-z+1.0)*(x+z-1.0);
         else
            f = -z*((y+z+1.0)*(x-z-1.0) +4.0*z);
         break;
      default:
         printf("invalid case detected for basis functions\n");
      }
      return f;
}

double pyr13basis_dx(int i, double x, double y, double z) {
      double f;

      switch(i)   {
      case 0:   // node 0
         if(x > y)
            f = 0.25*(x+z-1.0)*(-y+z+1.0) + 0.25*(x+y+1.0)*(-y+z+1.0);
         else
            f = 0.25*(-x+z+1.0)*(y+z-1.0) - 0.25*(x+y+1.0)*(y+z-1.0);
         break;
      case 1:   // midpoint node, between corner nodes 1 and 0
         if(x > y)
            f = 0.5*((y-z-1.0)*(x+1.0) + 2.0*z) + 0.5*(x+z-1.0)*(y-z-1.0);
         else
            f = 0.5*(y+z-1.0)*(x-1.0) + 0.5*(x-z+1.0)*(y+z-1.0);
         break;
      case 2:   // node 1
         if(x > y)
            f = 0.25*((x+z+1.0)*(-y+z+1.0) -4.0*z) +0.25*(x-y-1.0)*(-y+z+1.0);
```

```
   else
      f = 0.25*(x-z+1.0)*(-y-z+1.0) + 0.25*(x-y-1.0)*(-y-z+1.0);
   break;
case 3:   // midpoint node, between corner nodes 1 and 2
   if(x > y)
      f = -0.5*(y-z+1.0)*(y-1.0);
   else
      f = -0.5*(y+z-1.0)*(y+1.0);
   break;
case 4:   // node 2
   if(x > y)
      f = 0.25*(x+z+1.0)*(y-z+1.0) + 0.25*(x+y-1.0)*(y-z+1.0);
   else
      f = 0.25*(x-z+1.0)*(y+z+1.0) + 0.25*(x+y-1.0)*(y+z+1.0);
   break;
case 5:   // midpoint node, between corner nodes 2 and 3
   if(x > y)
      f = -0.5*(y-z+1.0)*(x+1.0) - 0.5*(y-z+1.0)*(x+z-1.0);
   else
      f = -0.5*((y+z+1.0)*(x-1.0) +2.0*z) - 0.5*(x-z+1.0)*(y+z+1.0);
   break;
case 6:   // node 3
   if(x > y)
      f = 0.25*(x+z-1.0)*(y-z+1.0) + 0.25*(x-y+1.0)*(y-z+1.0);
   else
      f = -0.25*((-x+z+1.0)*(y+z+1.0)-4.0*z)-0.25*(-x+y-1.0)*(y+z+1.0);
   break;
case 7:   // midpoint node, between corner nodes 0 and 3
   if(x > y)
      f = 0.5*(y-z+1.0)*(y-1.0);
   else
      f = 0.5*(y+z-1.0)*(y+1.0);
   break;
case 8:   // midpoint node, between corner nodes 0 and 4
   if(x > y)
      f = z*(y-z-1.0);
   else
      f = z*(y+z-1.0);
   break;
case 9:   // midpoint node, between corner nodes 1 and 4
   if(x > y)
      f = -z*(y-z-1.0);
   else
      f = -z*(y+z-1.0);
   break;
case 10:   // midpoint node, between corner nodes 2 and 4
   if(x > y)
      f = z*(y-z+1.0);
   else
      f = z*(y+z+1.0);
   break;
case 11:   // midpoint node, between corner nodes 3 and 4
   if(x > y)
      f = -z*(y-z+1.0);
   else
      f = -z*(y+z+1.0);
   break;
case 12:   // node 4
   f = 0.0;
```

```
      break;
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}

double pyr13basis_dy(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      if(x > y)
         f = 0.25*(x+z-1.0)*(-y+z+1.0) - 0.25*(x+y+1.0)*(x+z-1.0);
      else
         f = 0.25*(-x+z+1.0)*(y+z-1.0) + 0.25*(x+y+1.0)*(-x+z+1.0);
      break;
   case 1:   // midpoint node, between corner nodes 0 and 1
      if(x > y)
         f = 0.5*(x+z-1.0)*(x+1.0);
      else
         f = 0.5*(x-z+1.0)*(x-1.0);
      break;
   case 2:   // node 1
      if(x > y)
         f = -0.25*((x+z+1.0)*(-y+z+1.0)-4.0*z) -0.25*(x-y-1.0)*(x+z+1.0);
      else
         f = -0.25*(x-z+1.0)*(-y-z+1.0)-0.25*(x-y-1.0)*(x-z+1.0);
      break;
   case 3:   // midpoint node, between corner nodes 1 and 2
      if(x > y)
         f = -0.5*((x+z+1.0)*(y-1.0) +2.0*z) -0.5*(y-z+1.0)*(x+z+1.0);
      else
         f = -0.5*(x-z+1.0)*(y+1.0) - 0.5*(x-z+1.0)*(y+z-1.0);
      break;
   case 4:   // node 2
      if(x > y)
         f = 0.25*(x+z+1.0)*(y-z+1.0) +0.25*(x+y-1.0)*(x+z+1.0);
      else
         f = 0.25*(x-z+1.0)*(y+z+1.0) +0.25*(x+y-1.0)*(x-z+1.0);
      break;
   case 5:   // midpoint node, between corner nodes 2 and 3
      if(x > y)
         f = -0.5*(x+z-1.0)*(x+1.0);
      else
         f = -0.5*(x-z+1.0)*(x-1.0);
      break;
   case 6:   // node 3
      if(x > y)
         f = -0.25*(x+z-1.0)*(y-z+1.0) + 0.25*(x-y+1.0)*(x+z-1.0);
      else
         f = 0.25*((-x+z+1.0)*(y+z+1.0)-4.0*z)+0.25*(-x+y-1.0)*(-x+z+1.0);
      break;
   case 7:   // midpoint node, between corner nodes 0 and 3
      if(x > y)
         f = 0.5*(x+z-1.0)*(y-1.0) + 0.5*(y-z+1.0)*(x+z-1.0);
      else
         f = 0.5*((x-z-1.0)*(y+1.0) +2.0*z) + 0.5*(y+z-1.0)*(x-z-1.0);
      break;
```

```
   case 8:   // midpoint node, between corner nodes 0 and 4
      if(x > y)
         f = z*(x+z-1.0);
      else
         f = z*(x-z-1.0);
      break;
   case 9:   // midpoint node, between corner nodes 1 and 4
      if(x > y)
         f = -z*(x+z+1.0);
      else
         f = -z*(x-z+1.0);
      break;
   case 10:   // midpoint node, between corner nodes 2 and 4
      if(x > y)
         f = z*(x+z+1.0);
      else
         f = z*(x-z+1.0);
      break;
   case 11:   // midpoint node, between corner nodes 3 and 4
      if(x > y)
         f = -z*(x+z-1.0);
      else
         f = -z*(x-z-1.0);
      break;
   case 12:   // node 4
      f = 0.0;
      break;
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}

double pyr13basis_dz(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      if(x > y)
         f = 0.25*(x+y+1.0)*(-y+z+1.0) + 0.25*(x+y+1.0)*(x+z-1.0);
      else
         f = 0.25*(x+y+1.0)*(y+z-1.0) + 0.25*(x+y+1.0)*(-x+z+1.0);
      break;
   case 1:   // midpoint node, between corner nodes 0 and 1
      if(x > y)
         f = 0.5*((y-z-1.0)*(x+1.0)+2.0*z) + 0.5*(x+z-1.0)*(-x+1.0);
      else
         f = -0.5*(y+z-1.0)*(x-1.0) + 0.5*(x-z+1.0)*(x-1.0);
      break;
   case 2:   // node 1
      if(x > y)
         f = 0.25*(x-y-1.0)*(2.0*z -2.0 + x - y);
      else
         f = -0.25*(x-y-1.0)*(-y-z+1.0) - 0.25*(x-y-1.0)*(x-z+1.0);
      break;
   case 3:   // midpoint node, between corner nodes 1 and 2
      if(x > y)
         f = 0.5*((x+z+1.0)*(y-1.0) +2.0*z) -0.5*(y-z+1.0)*(y+1.0);
      else
```

```
            f = 0.5*(y+z-1.0)*(y+1.0) - 0.5*(x-z+1.0)*(y+1.0);
         break;
      case 4:   // node 2
         if(x > y)
            f = 0.25*(x+y-1.0)*(y-z+1.0) - 0.25*(x+y-1.0)*(x+z+1.0);
         else
            f = -0.25*(x+y-1.0)*(y+z+1.0) + 0.25*(x+y-1.0)*(x-z+1.0);
         break;
         case 5:   // midpoint node, between corner nodes 2 and 3
            if(x > y)
            f = 0.5*(x+z-1.0)*(x+1.0) - 0.5*(y-z+1.0)*(x+1.0);
         else
            f = 0.5*((y+z+1.0)*(x-1.0) +2.0*z) - 0.5*(x-z+1.0)*(x+1.0);
         break;
      case 6:   // node 3
         if(x > y)
            f = 0.25*(x-y+1.0)*(y-z+1.0) - 0.25*(x-y+1.0)*(x+z-1.0);
         else
            f = 0.25*(-x+y-1.0)*(y +2.0*z -x -2.0);
         break;
      case 7:   // midpoint node, between corner nodes 0 and 3
         if(x > y)
            f = -0.5*(x+z-1.0)*(y-1.0) + 0.5*(y-z+1.0)*(y-1.0);
         else
            f = 0.5*((x-z-1.0)*(y+1.0) +2.0*z) + 0.5*(y+z-1.0)*(-y+1.0);
         break;
      case 8:   // midpoint node, between corner nodes 0 and 4
         if(x > y)
            f = (y-z-1.0)*(x+z-1.0) - z*(x+z-1.0) + z*(y-z-1.0);
         else
            f = (x-z-1.0)*(y+z-1.0) - z*(y+z-1.0) + z*(x-z-1.0);
         break;
      case 9:   // midpoint node, between corner nodes 1 and 4
         if(x > y)
            f = (-x-z-1.0)*(y-z-1.0) -4.0*z +z*(x-y-2.0+2.0*z);
         else
            f = (-x+z-1.0)*(y+z-1.0) +z*(y+z-1.0) -z*(x-z+1.0);
         break;
      case 10:   // midpoint node, between corner nodes 2 and 4
         if(x > y)
            f = (y-z+1.0)*(x+z+1.0) -z*(x+z+1.0) +z*(y-z+1.0);
         else
            f = (x-z+1.0)*(y+z+1.0) -z*(y+z+1.0) +z*(x-z+1.0);
         break;
      case 11:   // midpoint node, between corner nodes 3 and 4
         if(x > y)
            f = (-y+z-1.0)*(x+z-1.0) +z*(x+z-1.0) -z*(y-z+1.0);
         else
            f = (-y-z-1.0)*(x-z-1.0) -4.0*z -z*(x-y -2.0*z + 2.0);
         break;
      case 12:   // node 4
         f = 4.0*z - 1.0;
         break;
      default:
         printf("invalid case detected for basis functions\n");
      }
      return f;
}
```

```
/*************************************************************************
* Basis functions and partial derivatives
* with symmetries about x and y axis
*************************************************************************/

/*************************************************************************
* apply symmetries by using transformed basis functions
*************************************************************************/

double pyr13symBasis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      f = pyr13basis(0,x,y,z) + pyr13basis(2,-x,y,z);
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr13basis(2,x,y,z) + pyr13basis(0,-x,y,z);
      f = f/2.0;
      break;
   case 4:   // node 2
      f = pyr13basis(4,x,y,z) + pyr13basis(6,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr13basis(6,x,y,z) + pyr13basis(4,-x,y,z);
      f = f/2.0;
      break;
   case 12:   // node 4
      f = pyr13basis(12,x,y,z);
      break;
   case 1:   // midpoint, between corner nodes 1 and 0
      f = pyr13basis(1,x,y,z) + pyr13basis(1,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 2 and 1
      f = pyr13basis(3,x,y,z) + pyr13basis(7,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      f = pyr13basis(5,x,y,z) + pyr13basis(5,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      f = pyr13basis(7,x,y,z) + pyr13basis(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint, between corner nodes 4 and 0
      f = pyr13basis(8,x,y,z) + pyr13basis(9,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 4 and 1
      f = pyr13basis(9,x,y,z) + pyr13basis(8,-x,y,z);
      f = f/2.0;
      break;
   case 10:   // midpoint, between corner nodes 4 and 2
      f = pyr13basis(10,x,y,z) + pyr13basis(11,-x,y,z);
      f = f/2.0;
```

```
      break;
   case 11:   // midpoint, between corner nodes 4 and 3
      f = pyr13basis(11,x,y,z) + pyr13basis(10,-x,y,z);
      f = f/2.0;
      break;
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}

double pyr13symBasis_dx(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      f = pyr13basis_dx(0,x,y,z) - pyr13basis_dx(2,-x,y,z);
      f = f/2.0;
      break;
   case 1:   // midpoint, between corner nodes 0 and 1
      f = pyr13basis_dx(1,x,y,z) - pyr13basis_dx(1,-x,y,z);
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr13basis_dx(2,x,y,z) - pyr13basis_dx(0,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 1 and 2
      f = pyr13basis_dx(3,x,y,z) - pyr13basis_dx(7,-x,y,z);
      f = f/2.0;
      break;
   case 4:   // node 2
      f = pyr13basis_dx(4,x,y,z) - pyr13basis_dx(6,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 2 and 3
      f = pyr13basis_dx(5,x,y,z) - pyr13basis_dx(5,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr13basis_dx(6,x,y,z) - pyr13basis_dx(4,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 0 and 3
      f = pyr13basis_dx(7,x,y,z) - pyr13basis_dx(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint, between corner nodes 0 and 4
      f = pyr13basis_dx(8,x,y,z) - pyr13basis_dx(9,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 1 and 4
      f = pyr13basis_dx(9,x,y,z) - pyr13basis_dx(8,-x,y,z);
      f = f/2.0;
      break;
   case 10:   // midpoint, between corner nodes 2 and 4
      f = pyr13basis_dx(10,x,y,z) - pyr13basis_dx(11,-x,y,z);
      f = f/2.0;
      break;
```

```
    case 11:   // midpoint, between corner nodes 3 and 4
       f = pyr13basis_dx(11,x,y,z) - pyr13basis_dx(10,-x,y,z);
       f = f/2.0;
       break;
    case 12:   // node 4
       f = 0.0;
       break;
    default:
       printf("invalid case detected for basis functions\n");
    }
    return f;
}

double pyr13symBasis_dy(int i, double x, double y, double z) {
    double f;

    switch(i)   {
    case 0:   // node 0
       f = pyr13basis_dy(0,x,y,z) + pyr13basis_dy(2,-x,y,z);
       f = f/2.0;
       break;
    case 1:   // midpoint, between corner nodes 0 and 1
       f = pyr13basis_dy(1,x,y,z) + pyr13basis_dy(1,-x,y,z);
       f = f/2.0;
       break;
    case 2:   // node 1
       f = pyr13basis_dy(2,x,y,z) + pyr13basis_dy(0,-x,y,z);
       f = f/2.0;
       break;
    case 3:   // midpoint, between corner nodes 1 and 2
       f = pyr13basis_dy(3,x,y,z) + pyr13basis_dy(7,-x,y,z);
       f = f/2.0;
       break;
    case 4:   // node 2
       f = pyr13basis_dy(4,x,y,z) + pyr13basis_dy(6,-x,y,z);
       f = f/2.0;
       break;
    case 5:   // midpoint, between corner nodes 2 and 3
       f = pyr13basis_dy(5,x,y,z) + pyr13basis_dy(5,-x,y,z);
       f = f/2.0;
       break;
    case 6:   // node 3
       f = pyr13basis_dy(6,x,y,z) + pyr13basis_dy(4,-x,y,z);
       f = f/2.0;
       break;
    case 7:   // midpoint, between corner nodes 0 and 3
       f = pyr13basis_dy(7,x,y,z) + pyr13basis_dy(3,-x,y,z);
       f = f/2.0;
       break;
    case 8:   // midpoint, between corner nodes 0 and 4
       f = pyr13basis_dy(8,x,y,z) + pyr13basis_dy(9,-x,y,z);
       f = f/2.0;
       break;
    case 9:   // midpoint, between corner nodes 1 and 4
       f = pyr13basis_dy(9,x,y,z) + pyr13basis_dy(8,-x,y,z);
       f = f/2.0;
       break;
    case 10:   // midpoint, between corner nodes 2 and 4
       f = pyr13basis_dy(10,x,y,z) + pyr13basis_dy(11,-x,y,z);
```

```
      f = f/2.0;
      break;
   case 11:   // midpoint, between corner nodes 3 and 4
      f = pyr13basis_dy(11,x,y,z) + pyr13basis_dy(10,-x,y,z);
      f = f/2.0;
      break;
   case 12:   // node 4
      f = 0.0;
      break;
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}

double pyr13symBasis_dz(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      f = pyr13basis_dz(0,x,y,z) + pyr13basis_dz(2,-x,y,z);
      f = f/2.0;
      break;
   case 1:   // midpoint, between corner nodes 0 and 1
      f = pyr13basis_dz(1,x,y,z) + pyr13basis_dz(1,-x,y,z);
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr13basis_dz(2,x,y,z) + pyr13basis_dz(0,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 1 and 2
      f = pyr13basis_dz(3,x,y,z) + pyr13basis_dz(7,-x,y,z);
      f = f/2.0;
      break;
   case 4:   // node 2
      f = pyr13basis_dz(4,x,y,z) + pyr13basis_dz(6,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 2 and 3
      f = pyr13basis_dz(5,x,y,z) + pyr13basis_dz(5,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr13basis_dz(6,x,y,z) + pyr13basis_dz(4,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 0 and 3
      f = pyr13basis_dz(7,x,y,z) + pyr13basis_dz(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint, between corner nodes 0 and 4
      f = pyr13basis_dz(8,x,y,z) + pyr13basis_dz(9,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 1 and 4
      f = pyr13basis_dz(9,x,y,z) + pyr13basis_dz(8,-x,y,z);
      f = f/2.0;
      break;
```

```
    case 10:   // midpoint, between corner nodes 2 and 4
       f = pyr13basis_dz(10,x,y,z) + pyr13basis_dz(11,-x,y,z);
       f = f/2.0;
       break;
    case 11:   // midpoint, between corner nodes 3 and 4
       f = pyr13basis_dz(11,x,y,z) + pyr13basis_dz(10,-x,y,z);
       f = f/2.0;
       break;
    case 12:   // node 4
       f = pyr13basis_dz(12,x,y,z);
       break;
    default:
       printf("invalid case detected for basis functions\n");
    }
    return f;
}
```

```
/*************************************************************************
* Programmer: Kevin Davies
* File: pyr14basis.c
* Version: 3.0
* Date last modified: Dec 12, 2004
* Description: Quadratic basis functions and their partial derivatives
*        for 14 node pyramidal element.
*
*************************************************************************/

/*************************************************************************
*
*                Case I for 14-Node Pyramidal Element
*
*************************************************************************/

double pyr14C1basis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      if(x > y)
         f = 0.25*(x+z)*(y-z)*(x+z-1.0)*(y-z-1.0);
      else
         f = 0.25*(x-z)*(y+z)*(x-z-1.0)*(y+z-1.0);
      break;
   case 1:   // midpoint node, between corner nodes 1 and 0
      if(x > y)
         f = -0.5*(x+z-1.0)*(y-z-1.0)*(x-z+1.0)*(y-z);
      else
         f = -0.5*(x-z+1.0)*(y+z-1.0)*(x-z-1.0)*(y-z);
      break;
   case 2:   // node 1
      if(x > y)
         f = 0.25*(x+z)*(z-y)*(x*(1.0-y) +z*(z+x-y-2.0) -y +1.0) -z*(x-y);
      else
         f = 0.25*(x-z)*(y+z)*(y+z-1.0)*(x-z+1.0);
      break;
   case 3:   // midpoint, between corner nodes 1 and 2
      if(x > y)
         f = -0.5*(-y+z-1.0)*(x+z)*((z+x-y-2.0)*z +x*(1.0-y) -y +1.0);
      else
         f = 0.5*(x-z+1.0)*(y+z-1.0)*(-y+z-1.0)*(x+z);
      break;
   case 4:   // node 2
      if(x > y)
         f = 0.25*(x+z)*(y-z)*(y-z+1.0)*(x+z+1.0);
      else
         f = 0.25*(x-z)*(y+z)*(x-z+1.0)*(y+z+1.0);
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      if(x > y)
         f = 0.5*(x-z+1.0)*(-y+z-1.0)*(x+z-1.0)*(y+z);
      else
         f = -0.5*(x-z+1.0)*(y+z)*((-z+x-y+2.0)*z +x*(y+1.0) -y -1.0);
      break;
   case 6:   // node 3
      if(x > y)
         f = 0.25*(x+z)*(z-y)*(-y+z-1.0)*(x+z-1.0);
```

```
      else
         f = 0.25*(x-z)*(y+z)*(z*(x-z-y+2.0) +x*(y+1.0) -y -1.0) +z*(x-y);
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      if(x > y)
         f = -0.5*(-y+z-1.0)*(x+z-1.0)*(-y+z+1.0)*(x-z);
      else
         f = 0.5*(-y+z-1.0)*(x-z)*(x-z-1.0)*(y+z-1.0);
      break;
   case 8:   // midpoint, between corner nodes 2 and 0
      if(x > y)
         f = -(-y+z-1.0)*(x+z-1.0)*((y-1.0)*x +z*(z-2.0*y+3.0) +y-1.0);
      else
         f = -(x-z+1.0)*(y+z-1.0)*(z*(-z+2.0*x-3.0) -x*(1.0+y) +y+1.0);
      break;
   case 9:   // midpoint, between corner nodes 4 and 0
      if(x > y)
         f = z*(y-z-1.0)*(x+z-1.0);
      else
         f = z*(x-z-1.0)*(y+z-1.0);
      break;
   case 10:   // midpoint, between corner nodes 4 and 1
      if(x > y)
         f = -z*((x+z+1.0)*(y-z-1.0) + 4.0*z);
      else
         f = -z*(x-z+1.0)*(y+z-1.0);
      break;
   case 11:   // midpoint, between corner nodes 4 and 2
      if(x > y)
         f = z*(y-z+1.0)*(x+z+1.0);
      else
         f = z*(x-z+1.0)*(y+z+1.0);
      break;
   case 12:   // midpoint, between corner nodes 4 and 3
      if(x > y)
         f = -z*(y-z+1.0)*(x+z-1.0);
      else
         f = -z*((y+z+1.0)*(x-z-1.0) +4.0*z);
      break;

   case 13:   // node 4
         f = z*(2.0*z-1.0);
      break;
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}


/*****************************************************************************
* Apply Extra symmetries to basis functions
*****************************************************************************/

double pyr14C1symBasis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
```

```
      f = pyr14C1basis(0,x,y,z) + pyr14C1basis(2,-x,y,z);
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr14C1basis(2,x,y,z) + pyr14C1basis(0,-x,y,z);
      f = f/2.0;
      break;
   case 4:   // node 2
      f = pyr14C1basis(4,x,y,z) + pyr14C1basis(6,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr14C1basis(6,x,y,z) +  pyr14C1basis(4,-x,y,z);
      f = f/2.0;
      break;
   case 13:   // node 4
      f = pyr14C1basis(13,x,y,z);
      break;
   case 1:   // midpoint, between corner nodes 1 and 0
      f = pyr14C1basis(1,x,y,z) + pyr14C1basis(1,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 2 and 1
      f = pyr14C1basis(3,x,y,z) + pyr14C1basis(7,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      f = pyr14C1basis(5,x,y,z) + pyr14C1basis(5,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      f = pyr14C1basis(7,x,y,z) + pyr14C1basis(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint on base, between corner nodes 2 and 0
      f = pyr14C1basis(8,x,y,z) + pyr14C1basis(8,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 4 and 0
      f = pyr14C1basis(9,x,y,z) + pyr14C1basis(10,-x,y,z);
      f = f/2.0;
      break;
   case 10:   // midpoint, between corner nodes 4 and 1
      f = pyr14C1basis(10,x,y,z) + pyr14C1basis(9,-x,y,z);
      f = f/2.0;
      break;
   case 11:   // midpoint, between corner nodes 4 and 2
      f = pyr14C1basis(11,x,y,z) + pyr14C1basis(12,-x,y,z);
      f = f/2.0;
      break;
   case 12:   // midpoint, between corner nodes 4 and 3
      f = pyr14C1basis(12,x,y,z) + pyr14C1basis(11,-x,y,z);
      f = f/2.0;
      break;
   default:
      printf("invalid case detected for pyr14C1basis functions\n");
   }
   return f;
}
```

```
/*************************************************************************
*
*                Case II for 14-Node Pyramidal Element
*
*************************************************************************/

double pyr14C2basis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      if(x > y)
         f = 0.25*(x+z)*(y-z)*(x+z-1.0)*(y-z-1.0);
      else
         f = 0.25*(x-z)*(y+z)*(x-z-1.0)*(y+z-1.0);
      break;
   case 1:   // midpoint node, between corner nodes 1 and 0
      if(x > y)
         f = -0.5*(x+z-1.0)*(((y-z-1.0)*(x+1.0)*y -z) +z*(2.0*x+1.0));
      else
         f = -0.5*(x-z+1.0)*(y+z-1.0)*(x-1.0)*y;
      break;
   case 2:   // node 1
      if(x > y)
         f = -0.25*(x+z)*(y-z)*((x+z+1.0)*(-y+z+1.0) -4.0*z) -z*(x-y);
      else
         f = -0.25*(x-z)*(y+z)*(-y-z+1.0)*(x-z+1.0);
      break;
   case 3:   // midpoint, between corner nodes 1 and 2
      if(x > y)
         f = -0.5*(y-z+1.0)*(((x+z+1.0)*(y-1.0)*x -z) +z*(2.0*y+1.0));
      else
         f = -0.5*(x-z+1.0)*(y+z-1.0)*(y+1.0)*x;
      break;
   case 4:   // node 2
      if(x > y)
         f = 0.25*(x+z)*(y-z)*(y-z+1.0)*(x+z+1.0);
      else
         f = 0.25*(x-z)*(y+z)*(x-z+1.0)*(y+z+1.0);
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      if(x > y)
         f = -0.5*(x+z-1.0)*(y-z+1.0)*(x+1.0)*y;
      else
         f = -0.5*(x-z+1.0)*(((y+z+1.0)*(x-1.0)*y -z) +z*(2.0*x+1.0));
      break;
   case 6:   // node 3
      if(x > y)
         f = 0.25*(x+z)*(y-z)*(y-z+1.0)*(x+z-1.0);
      else
         f = 0.25*(x-z)*(y+z)*((x-z-1.0)*(y+z+1.0) +4.0*z) +z*(x-y);
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      if(x > y)
         f = -0.5*(y-z+1.0)*(x+z-1.0)*(y-1.0)*x;
      else
         f = -0.5*(y+z-1.0)*(((x-z-1.0)*(y+1.0)*x -z) +z*(2.0*y+1.0));
      break;
```

```
   case 8:   // midpoint, between corner nodes 2 and 0
      if(x > y)
         f = (y-z+1.0)*(x+z-1.0)*(y-1.0)*(x+1.0)
           + z*(x+z-1.0)*(y-z+1.0)*(x-y+z+1.0);
      else
         f = (x-z+1.0)*(y+z-1.0)*(x-1.0)*(y+1.0)
           - z*(x-z+1.0)*(y+z-1.0)*(x-y-z-1.0);
      break;
   case 9:   // midpoint, between corner nodes 4 and 0
      if(x > y)
         f = z*(y-z-1.0)*(x+z-1.0);
      else
         f = z*(x-z-1.0)*(y+z-1.0);
      break;
   case 10:   // midpoint, between corner nodes 4 and 1
      if(x > y)
         f = -z*((x+z+1.0)*(y-z-1.0) + 4.0*z);
      else
         f = -z*(x-z+1.0)*(y+z-1.0);
      break;
   case 11:   // midpoint, between corner nodes 4 and 2
      if(x > y)
         f = z*(y-z+1.0)*(x+z+1.0);
      else
         f = z*(x-z+1.0)*(y+z+1.0);
      break;
   case 12:   // midpoint, between corner nodes 4 and 3
      if(x > y)
         f = -z*(y-z+1.0)*(x+z-1.0);
      else
         f = -z*((y+z+1.0)*(x-z-1.0) +4.0*z);
      break;

   case 13:   // node 4
         f = z*(2.0*z-1.0);
      break;
   default:
      printf("invalid case detected for basis functions: i = %d\n", i);
   }
   return f;
}

double pyr14C2basis_dx(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      if(x > y)
         f = 0.25*(y-z)*(y-z-1.0)*(2.0*x+2.0*z-1.0);
      else
         f = 0.25*(y+z)*(y+z-1.0)*(2.0*x-2.0*z-1.0);
      break;
   case 1:   // midpoint node, between corner nodes 1 and 0
      if(x > y)
         f = -0.5*((y-z-1.0)*(x+1.0)*y -z +z*(2.0*x+1.0))
                -0.5*(x+z-1.0)*((y-z-1.0)*y +2.0*z);
      else
         f = 0.5*y*(y+z-1.0)*(z-2.0*x);
      break;
```

```
case 2:   // node 1
   if(x > y)
      f = -0.25*(y-z)*(((x+z+1.0)*(-y+z+1.0)-4.0*z)+(x+z)*(-y+z+1.0))-z;
   else
      f = 0.25*(y+z)*(y+z-1.0)*(2.0*x-2.0*z+1.0);
   break;
case 3:   // midpoint node, between corner nodes 1 and 2
   if(x > y)
      f = 0.5*(-y+z-1.0)*(y-1.0)*(2.0*x+z+1.0);
   else
      f = 0.5*(y+1.0)*(y+z-1.0)*(z-2.0*x-1.0);
   break;
case 4:   // node 2
   if(x > y)
      f = 0.25*(y-z)*(y-z+1.0)*(2.0*x+2.0*z+1.0);
   else
      f = 0.25*(y+z)*(y+z+1.0)*(2.0*x-2.0*z+1.0);
   break;
case 5:   // midpoint node, between corner nodes 2 and 3
   if(x > y)
      f = 0.5*y*(-y+z-1.0)*(2.0*x+z);
   else
      f = -0.5*((y+z+1.0)*(x-1.0)*y -z +z*(2.0*x+1.0))
            -0.5*(x-z+1.0)*((y+z+1.0)*y +2.0*z);
   break;
case 6:   // node 3
   if(x > y)
      f = 0.25*(z-y)*(z-y-1.0)*(2.0*x+2.0*z-1.0);
   else
      f = 0.25*(y+z)*(((x-z-1.0)*(y+z+1.0)+4.0*z) + (x-z)*(y+z+1.0))+z;
   break;
case 7:   // midpoint node, between corner nodes 0 and 3
   if(x > y)
      f = 0.5*(y-1.0)*(-y+z-1.0)*(2.0*x+z-1.0);
   else
      f = 0.5*(y+z-1.0)*(y+1.0)*(-2.0*x+z+1.0);
   break;
case 8:   // midpoint node, between corner nodes 0 and 2
   if(x > y)
      f = (y-z+1.0)*((y-1.0)*(x+1.0)+(x+z-1.0)*(y-1.0)
                     +z*(x-y+z+1.0)+z*(x+z-1.0));
   else
      f = (y+z-1.0)*((y+1.0)*(x-1.0)+(x-z+1.0)*(y+1.0)
                     -z*(x-z-y-1.0)-z*(x-z+1.0));
   break;
case 9:   // midpoint node, between corner nodes 0 and 4
   if(x > y)
      f = z*(y-z-1.0);
   else
      f = z*(y+z-1.0);
   break;
case 10:   // midpoint node, between corner nodes 1 and 4
   if(x > y)
      f = -z*(y-z-1.0);
   else
      f = -z*(y+z-1.0);
   break;
case 11:   // midpoint node, between corner nodes 2 and 4
   if(x > y)
```

```
            f = z*(y-z+1.0);
         else
            f = z*(y+z+1.0);
         break;
      case 12:   // midpoint node, between corner nodes 3 and 4
         if(x > y)
            f = -z*(y-z+1.0);
         else
            f = -z*(y+z+1.0);
         break;
      case 13:   // node 4
         f = 0.0;
         break;
      default:
         printf("invalid case detected for basis functions: i = %d\n", i);
      }
      return f;
   }

   double pyr14C2basis_dy(int i, double x, double y, double z) {
      double f;

      switch(i)   {
      case 0:   // node 0
         if(x > y)
            f = 0.25*(x+z)*(x+z-1.0)*(2.0*y-2.0*z-1.0);
         else
            f = 0.25*(x-z)*(x-z-1.0)*(2.0*y+2.0*z-1.0);
         break;
      case 1:   // midpoint node, between corner nodes 0 and 1
         if(x > y)
            f = -0.5*(x+z-1.0)*(x+1.0)*(2.0*y-z-1.0);
         else
            f = 0.5*(x-1.0)*(-x+z-1.0)*(2.0*y+z-1.0);
         break;
      case 2:   // node 1
         if(x > y)
            f = -0.25*(x+z)*(((x+z+1.0)*(-y+z+1.0)-4.0*z) - (y-z)*(x+z+1.0)) +z;
         else
            f = 0.25*(x-z)*(x-z+1.0)*(2.0*y+2.0*z-1.0);
         break;
      case 3:   // midpoint node, between corner nodes 1 and 2
         if(x > y)
            f = -0.5*((x+z+1.0)*(y-1.0)*x -z +z*(2.0*y+1.0))
                     -0.5*(y-z+1.0)*((x+z+1.0)*x +2.0*z);
         else
            f = 0.5*x*(-x+z-1.0)*(2.0*y+z);
         break;
      case 4:   // node 2
         if(x > y)
            f = 0.25*(x+z)*(x+z+1.0)*(2.0*y-2.0*z+1.0);
         else
            f = 0.25*(x-z)*(x-z+1.0)*(2.0*y+2.0*z+1.0);
         break;
      case 5:   // midpoint node, between corner nodes 2 and 3
         if(x > y)
            f = 0.5*(x+1.0)*(x+z-1.0)*(z-2.0*y-1.0);
         else
            f = 0.5*(-x+z-1.0)*(x-1.0)*(2.0*y+z+1.0);
```

```
      break;
   case 6:   // node 3
      if(x > y)
         f = -0.25*(x+z)*(x+z-1.0)*(-2.0*y+2.0*z-1.0);
      else
         f = 0.25*(x-z)*(((x-z-1.0)*(y+z+1.0)+4.0*z) + (y+z)*(x-z-1.0)) -z;
      break;
   case 7:   // midpoint node, between corner nodes 0 and 3
      if(x > y)
         f = 0.5*x*(x+z-1.0)*(-2.0*y+z);
      else
         f = -0.5*((x-z-1.0)*(y+1.0)*x -z +z*(2.0*y+1.0))
                  -0.5*(y+z-1.0)*((x-z-1.0)*x +2.0*z);
      break;
   case 8:   // midpoint node, between corner nodes 0 and 2
      if(x > y)
         f = (x+z-1.0)*((y-1.0)*(x+1.0)+(y-z+1.0)*(x+1.0)
                           +z*(x-y+z+1.0)-z*(y-z+1.0));
      else
         f = (x-z+1.0)*((y+1.0)*(x-1.0)+(y+z-1.0)*(x-1.0)
                           -z*(x-z-y-1.0)+z*(y+z-1.0));
      break;
   case 9:   // midpoint node, between corner nodes 0 and 4
      if(x > y)
         f = z*(x+z-1.0);
      else
         f = z*(x-z-1.0);
      break;
   case 10:   // midpoint node, between corner nodes 1 and 4
      if(x > y)
         f = -z*(x+z+1.0);
      else
         f = -z*(x-z+1.0);
      break;
   case 11:   // midpoint node, between corner nodes 2 and 4
      if(x > y)
         f = z*(x+z+1.0);
      else
         f = z*(x-z+1.0);
      break;
   case 12:   // midpoint node, between corner nodes 3 and 4
      if(x > y)
         f = -z*(x+z-1.0);
      else
         f = -z*(x-z-1.0);
      break;
   case 13:   // node 4
      f = 0.0;
      break;
   default:
      printf("invalid case detected for basis functions: i = %d\n", i);
   }
   return f;
}

double pyr14C2basis_dz(int i, double x, double y, double z) {
   double f;

   switch(i)   {
```

```
case 0:   // node 0
   if(x > y)
      f = 0.25*(y-z-1.0)*((x+z)*(y-z)+(x+z-1.0)*(y-z)-(x+z-1.0)*(x+z))
                          -0.25*(x+z-1.0)*(x+z)*(y-z);
   else
      f = 0.25*(x-z-1.0)*((y+z)*(x-z)+(y+z-1.0)*(x-z)-(y+z-1.0)*(y+z))
                          -0.25*(y+z-1.0)*(y+z)*(x-z);
   break;
case 1:   // midpoint node, between corner nodes 0 and 1
   if(x > y)
      f = -0.5*((y-z-1.0)*(x+1.0)*y -z +z*(2.0*x+1.0))
                          -0.5*(x+z-1.0)*(-(x+1.0)*y +2.0*x);
   else
      f = 0.5*y*(x-1.0)*(-x+y+2.0*z-2.0);
   break;
case 2:   // node 1
   if(x > y)
      f = -0.25*(y-z)*((x+z+1.0)*(-y+z+1.0)-4.0*z) +0.25*(x+z)*
        (((x+z+1.0)*(-y+z+1.0) -4.0*z)-(y-z)*(-y+2.0*z+x-2.0))-x+y;
   else
      f = 0.25*(y+z)*(x-z+1.0)*(-y-z+1.0) -0.25*(x-z)*
        ((x-z+1.0)*(-y-z+1.0)-(y+z)*(-y-z+1.0)-(y+z)*(x-z+1.0));
   break;
case 3:   // midpoint node, between corner nodes 1 and 2
   if(x > y)
      f = 0.5*((x+z+1.0)*(y-1.0)*x -z +z*(2.0*y+1.0))
                    -0.5*(y-z+1.0)*((y-1.0)*x +2.0*y);
   else
      f = 0.5*x*(y+1.0)*(-x+y+2.0*z-2.0);
   break;
case 4:   // node 2
   if(x > y)
      f = -0.25*(x+z+1.0)*((x+z)*(y-z)-(y-z+1.0)*(y-z)+(y-z+1.0)*(x+z))
                       +0.25*(y-z+1.0)*(x+z)*(y-z);
   else
      f = -0.25*(y+z+1.0)*((x-z)*(y+z)+(x-z+1.0)*(y+z)-(x-z+1.0)*(x-z))
                       +0.25*(x-z+1.0)*(x-z)*(y+z);
   break;
case 5:   // midpoint node, between corner nodes 2 and 3
      if(x > y)
      f = 0.5*y*(x+1.0)*(-y+2.0*z+x-2.0);
   else
      f = 0.5*((y+z+1.0)*(x-1.0)*y -z +z*(2.0*x+1.0))
                    -0.5*(x-z+1.0)*((x-1.0)*y +2.0*x);
   break;
case 6:   // node 3
   if(x > y)
      f = 0.25*(y-z)*(x+z-1.0)*(y-z+1.0) -0.25*(x+z)*
           ((x+z-1.0)*(y-z+1.0)-(y-z)*(y-z+1.0)+(y-z)*(x+z-1.0));
   else
      f = -0.25*(y+z)*((x-z-1.0)*(y+z+1.0)+4.0*z) +0.25*(x-z)*
           (((x-z-1.0)*(y+z+1.0)+4.0*z)+(y+z)*(-y-2.0*z+2.0+x))+x-y;
   break;
case 7:   // midpoint node, between corner nodes 0 and 3
   if(x > y)
      f = 0.5*x*(y-1.0)*(-y+2.0*z-2.0+x);
   else
      f = -0.5*((x-z-1.0)*(y+1.0)*x -z +z*(2.0*y+1.0))
                          -0.5*(y+z-1.0)*(2.0*y-x*y-x);
```

```
      break;
   case 8:   // midpoint node, between corner nodes 0 and 2
      if(x > y)
        f = (y-z+1.0)*((y-1.0)*(x+1.0)+z*(x-y+z+1.0))+(x+z-1.0)*
          ((y-z+1.0)*(x-y+z+1.0)-z*(x-y+z+1.0)-(y-1.0)*(x+1.0)+z*(y-z+1.0));
      else
        f = (x-z+1.0)*((y+1.0)*(x-1.0)-z*(x-z-y-1.0))+(y+z-1.0)*
          (z*(x-z+1.0)-(y+1.0)*(x-1.0)-(x-z+1.0)*(x-z-y-1.0)+z*(x-z-y-1.0));
      break;
   case 9:   // midpoint node, between corner nodes 0 and 4
      if(x > y)
         f = (y-z-1.0)*(x+z-1.0) -z*(x+z-1.0) +z*(y-z-1.0);
      else
         f = (x-z-1.0)*(y+z-1.0) -z*(y+z-1.0) +z*(x-z-1.0);
      break;
   case 10:   // midpoint node, between corner nodes 1 and 4
      if(x > y)
         f = -(x+z+1.0)*(y-z-1.0) -4.0*z -z*(y+2.0-2.0*z-x);
      else
         f = -(x-z+1.0)*(y+z-1.0) +z*(2.0*z+y-2.0-x);
      break;
   case 11:   // midpoint node, between corner nodes 2 and 4
      if(x > y)
         f = (y-z+1.0)*(x+z+1.0) +z*(-2.0*z-x+y);
      else
         f = (x-z+1.0)*(y+z+1.0) -z*(2.0*z+y-x);
      break;
   case 12:   // midpoint node, between corner nodes 3 and 4
      if(x > y)
         f = -(y-z+1.0)*(x+z-1.0) -z*(-2.0*z-x+2.0+y);
      else
         f = -(y+z+1.0)*(x-z-1.0) -4.0*z -z*(x-2.0*z +2.0-y);
      break;
// new basis functions
   case 13:   // node 4
      f = 4.0*z - 1.0;
      break;
   default:
      printf("invalid case detected for basis functions: i = %d\n", i);
   }
   return f;
}

/*************************************************************************
* Apply Extra symmetries to basis functions
*************************************************************************/

double pyr14C2symBasis(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      f = pyr14C2basis(0,x,y,z) + pyr14C2basis(2,-x,y,z);
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr14C2basis(2,x,y,z) + pyr14C2basis(0,-x,y,z);
      f = f/2.0;
      break;
```

```
   case 4:   // node 2
      f = pyr14C2basis(4,x,y,z) + pyr14C2basis(6,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr14C2basis(6,x,y,z) +  pyr14C2basis(4,-x,y,z);
      f = f/2.0;
      break;
   case 13:   // node 4
      f = pyr14C2basis(13,x,y,z);
      break;
   case 1:   // midpoint, between corner nodes 1 and 0
      f = pyr14C2basis(1,x,y,z) + pyr14C2basis(1,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 2 and 1
      f = pyr14C2basis(3,x,y,z) + pyr14C2basis(7,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      f = pyr14C2basis(5,x,y,z) + pyr14C2basis(5,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      f = pyr14C2basis(7,x,y,z) + pyr14C2basis(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint on base, between corner nodes 2 and 0
      f = pyr14C2basis(8,x,y,z) + pyr14C2basis(8,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 4 and 0
      f = pyr14C2basis(9,x,y,z) + pyr14C2basis(10,-x,y,z);
      f = f/2.0;
      break;
   case 10:   // midpoint, between corner nodes 4 and 1
      f = pyr14C2basis(10,x,y,z) + pyr14C2basis(9,-x,y,z);
      f = f/2.0;
      break;
   case 11:   // midpoint, between corner nodes 4 and 2
      f = pyr14C2basis(11,x,y,z) + pyr14C2basis(12,-x,y,z);
      f = f/2.0;
      break;
   case 12:   // midpoint, between corner nodes 4 and 3
      f = pyr14C2basis(12,x,y,z) + pyr14C2basis(11,-x,y,z);
      f = f/2.0;
      break;
   default:
      printf("invalid case detected for pyr14C1basis functions\n");
   }
   return f;
}

double pyr14C2symBasis_dx(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      f = pyr14C2basis_dx(0,x,y,z) - pyr14C2basis_dx(2,-x,y,z);
```

```
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr14C2basis_dx(2,x,y,z) - pyr14C2basis_dx(0,-x,y,z);
      f = f/2.0;
      break;
   case 4:   // node 2
      f = pyr14C2basis_dx(4,x,y,z) - pyr14C2basis_dx(6,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr14C2basis_dx(6,x,y,z) -  pyr14C2basis_dx(4,-x,y,z);
      f = f/2.0;
      break;
   case 13:   // node 4
      f = pyr14C2basis_dx(13,x,y,z);
      break;
   case 1:   // midpoint, between corner nodes 1 and 0
      f = pyr14C2basis_dx(1,x,y,z) - pyr14C2basis_dx(1,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 2 and 1
      f = pyr14C2basis_dx(3,x,y,z) - pyr14C2basis_dx(7,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      f = pyr14C2basis_dx(5,x,y,z) - pyr14C2basis_dx(5,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      f = pyr14C2basis_dx(7,x,y,z) - pyr14C2basis_dx(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint on base, between corner nodes 2 and 0
      f = pyr14C2basis_dx(8,x,y,z) - pyr14C2basis_dx(8,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 4 and 0
      f = pyr14C2basis_dx(9,x,y,z) - pyr14C2basis_dx(10,-x,y,z);
      f = f/2.0;
      break;
   case 10:   // midpoint, between corner nodes 4 and 1
      f = pyr14C2basis_dx(10,x,y,z) - pyr14C2basis_dx(9,-x,y,z);
      f = f/2.0;
      break;
   case 11:   // midpoint, between corner nodes 4 and 2
      f = pyr14C2basis_dx(11,x,y,z) - pyr14C2basis_dx(12,-x,y,z);
      f = f/2.0;
      break;
   case 12:   // midpoint, between corner nodes 4 and 3
      f = pyr14C2basis_dx(12,x,y,z) - pyr14C2basis_dx(11,-x,y,z);
      f = f/2.0;
      break;
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}
```

```
double pyr14C2symBasis_dy(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      f = pyr14C2basis_dy(0,x,y,z) + pyr14C2basis_dy(2,-x,y,z);
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr14C2basis_dy(2,x,y,z) + pyr14C2basis_dy(0,-x,y,z);
      f = f/2.0;
      break;
   case 4:   // node 2
      f = pyr14C2basis_dy(4,x,y,z) + pyr14C2basis_dy(6,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr14C2basis_dy(6,x,y,z) +  pyr14C2basis_dy(4,-x,y,z);
      f = f/2.0;
      break;
   case 13:   // node 4
      f = pyr14C2basis_dy(13,x,y,z);
      break;
   case 1:   // midpoint, between corner nodes 1 and 0
      f = pyr14C2basis_dy(1,x,y,z) + pyr14C2basis_dy(1,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 2 and 1
      f = pyr14C2basis_dy(3,x,y,z) + pyr14C2basis_dy(7,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      f = pyr14C2basis_dy(5,x,y,z) + pyr14C2basis_dy(5,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      f = pyr14C2basis_dy(7,x,y,z) + pyr14C2basis_dy(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint on base, between corner nodes 2 and 0
      f = pyr14C2basis_dy(8,x,y,z) + pyr14C2basis_dy(8,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 4 and 0
      f = pyr14C2basis_dy(9,x,y,z) + pyr14C2basis_dy(10,-x,y,z);
      f = f/2.0;
      break;
   case 10:   // midpoint, between corner nodes 4 and 1
      f = pyr14C2basis_dy(10,x,y,z) + pyr14C2basis_dy(9,-x,y,z);
      f = f/2.0;
      break;
   case 11:   // midpoint, between corner nodes 4 and 2
      f = pyr14C2basis_dy(11,x,y,z) + pyr14C2basis_dy(12,-x,y,z);
      f = f/2.0;
      break;
   case 12:   // midpoint, between corner nodes 4 and 3
      f = pyr14C2basis_dy(12,x,y,z) + pyr14C2basis_dy(11,-x,y,z);
      f = f/2.0;
      break;
```

```
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}

double pyr14C2symBasis_dz(int i, double x, double y, double z) {
   double f;

   switch(i)   {
   case 0:   // node 0
      f = pyr14C2basis_dz(0,x,y,z) + pyr14C2basis_dz(2,-x,y,z);
      f = f/2.0;
      break;
   case 2:   // node 1
      f = pyr14C2basis_dz(2,x,y,z) + pyr14C2basis_dz(0,-x,y,z);
      f = f/2.0;
      break;
   case 4:   // node 2
      f = pyr14C2basis_dz(4,x,y,z) + pyr14C2basis_dz(6,-x,y,z);
      f = f/2.0;
      break;
   case 6:   // node 3
      f = pyr14C2basis_dz(6,x,y,z) +  pyr14C2basis_dz(4,-x,y,z);
      f = f/2.0;
      break;
   case 13:   // node 4
      f = pyr14C2basis_dz(13,x,y,z);
      break;
   case 1:   // midpoint, between corner nodes 1 and 0
      f = pyr14C2basis_dz(1,x,y,z) + pyr14C2basis_dz(1,-x,y,z);
      f = f/2.0;
      break;
   case 3:   // midpoint, between corner nodes 2 and 1
      f = pyr14C2basis_dz(3,x,y,z) + pyr14C2basis_dz(7,-x,y,z);
      f = f/2.0;
      break;
   case 5:   // midpoint, between corner nodes 3 and 2
      f = pyr14C2basis_dz(5,x,y,z) + pyr14C2basis_dz(5,-x,y,z);
      f = f/2.0;
      break;
   case 7:   // midpoint, between corner nodes 3 and 0
      f = pyr14C2basis_dz(7,x,y,z) + pyr14C2basis_dz(3,-x,y,z);
      f = f/2.0;
      break;
   case 8:   // midpoint on base, between corner nodes 2 and 0
      f = pyr14C2basis_dz(8,x,y,z) + pyr14C2basis_dz(8,-x,y,z);
      f = f/2.0;
      break;
   case 9:   // midpoint, between corner nodes 4 and 0
      f = pyr14C2basis_dz(9,x,y,z) + pyr14C2basis_dz(10,-x,y,z);
      f = f/2.0;
      break;
   case 10:   // midpoint, between corner nodes 4 and 1
      f = pyr14C2basis_dz(10,x,y,z) + pyr14C2basis_dz(9,-x,y,z);
      f = f/2.0;
      break;
   case 11:   // midpoint, between corner nodes 4 and 2
      f = pyr14C2basis_dz(11,x,y,z) + pyr14C2basis_dz(12,-x,y,z);
```

```
      f = f/2.0;
      break;
   case 12:   // midpoint, between corner nodes 4 and 3
      f = pyr14C2basis_dz(12,x,y,z) + pyr14C2basis_dz(11,-x,y,z);
      f = f/2.0;
      break;
   default:
      printf("invalid case detected for basis functions\n");
   }
   return f;
}
```

```
/*************************************************************************
* Author: Kevin Davies
* File solution.c
* Version: 4.0
* Date last modified: Dec 15, 2004
* Description: functions for true solution and its 2nd order derivatives.
*************************************************************************/

/*************************************************************************
*    True solution function
*
*************************************************************************/

double solution(double x, double y, double z)   {
   double u;

   // Possible alternate solutions
// alternate sol'n #1
//   u = sin(pi*x)*sin(pi*y)*sin(pi*z);

// alternate sol'n #2
   u = sin(pi*x)*sin(2.0*pi*y)*sin(3.0*pi*z);

// original sol'n
//   u = x*y*z*(1.0-x)*(1.0-y)*(1.0-z);

// alternate #4
   //   u = x*y*z*sin(pi*x)*sin(pi*y)*sin(pi*z);

// alternate #5
//   u = pow(x,2)*pow(y,2)*pow(z,2)*(1.0-x)*(1.0-y)*(1.0-z);
   return u;
}

/*************************************************************************
*    2nd order derivatives of true solution function
*
*************************************************************************/
double func(double x, double y, double z)  {
   double f;
   double x2,y2,z2,xm1,ym1,zm1;

   x2 = pow(x,2);
   y2 = pow(y,2);
   z2 = pow(z,2);
   xm1 = 1.0-x;
   ym1 = 1.0-y;
   zm1 = 1.0-z;
   double siny, sinz, sinx, cosx, cosy, cosz;
   siny = sin(pi*y);
   sinz = sin(pi*z);
   sinx = sin(pi*x);
   cosx = cos(pi*x);
   cosy = cos(pi*y);
   cosz = cos(pi*z);

   // Possible alternate solutions
// alternate sol'n #1
//   f = 3.0*pi*pi*sin(pi*x)*sin(pi*y)*sin(pi*z);
```

```
// alternate sol'n #2
   f = 14.0*pi*pi*sin(pi*x)*sin(2.0*pi*y)*sin(3.0*pi*z);

// original sol'n
//   f = 2.0*(z*(1.0-z)*(y*(1.0-y) + x*(1.0-x)) + x*y*(1.0-x)*(1.0-y));

// alterante sol'n #4
//   f = pi*(-2.0*y*z*cosx*siny*sinz+3.0*pi*x*y*z*sinx*siny*sinz
//            -2.0*x*z*sinx*cosy*sinz-2.0*x*y*sinx*siny*cosz);

// alternate #5
/*   f = -(2.0*y2*z2*xm1*ym1*zm1-4.0*x*y2*z2*ym1*zm1+2.0*x2*z2*xm1*ym1*zm1
        - 4.0*x2*y*z2*xm1*zm1+2.0*x2*y2*xm1*ym1*zm1-4.0*x2*y2*z*xm1*ym1);
*/
   return f;
}



/*************************************************************************
  Gaussian Elimination solver (without pivoting)
   Inputs: A - Matrix
           b - RHS vector
           x - vector to be solved for (must be allocated)
*************************************************************************/
void naiveGauss(SPMAT *A, VEC *b, VEC *x)    {
   int i,j,k,n;
   double sum, xmult;

   n = A->m;
   for(k = 0;k < n - 1; k++)    {
      for(i = k+1; i < n; i++)    {
         xmult = sp_get_val(A, i,k) / sp_get_val(A, k,k);
         sp_set_val(A, i,k, xmult);
         for(j = k+1; j < n; j++)    {
            sp_set_val(A,i,j,sp_get_val(A,i,j)-sp_get_val(A,k,j)*xmult);
         }
         b->ve[i] = b->ve[i] - b->ve[k] * xmult;
      }
   }
   x->ve[n-1] = b->ve[n-1]/sp_get_val(A, n-1,n-1);
   for(i = n-2; i >= 0; i--)    {
      sum = b->ve[i];
      for(j = i+1; j < n;j++)    {
         sum = sum - sp_get_val(A, i,j)*x->ve[j];
      }
      x->ve[i] = sum/sp_get_val(A, i,i);
   }
}

/*************************************************************************
  Gauss Seidel Method with relaxation
   Inputs:
       A - Matrix
       b - RHS vector
       x - vector to be solved for (must be allocated), and
              contain initial guess.
       maxIter - maximum number of iterations
```

```
        tol - tolerance: stop iterating if converges to within this value
        lambda - relaxation value: such that: 0 <= lambda <= 2
        used to weight average of results to control convergence,
           x_new = lambda*x_new + (1-lambda)*x_old
           Thus,
              0 <= lambda < 1: underrelaxation
              lambda = 1: No relaxation
              1 < lambda <= 2: overrelaxation
     Return values:
         0: Method converged to within tolerance
        -1: Method did not converge within specified number of iterations
        -2: Error, mothod halted due to error in inputs, etc.
*************************************************************************/

int GaussSeidel(SPMAT *A,VEC *b,VEC *x,int maxIter,
                                double tol,double lambda)    {
   int i, j, n, itr = 1, done = 0, retVal = 0;
   double temp, sum, oldx, err;

   n = A->m;
   for(i = 0; i < n;i++)    {
      temp = sp_get_val(A, i,i);
      for(j = 0; j < n; j++)
         sp_set_val(A, i,j, sp_get_val(A, i,j)/temp);
      b->ve[i] = b->ve[i]/temp;
   }
   for(i = 0; i < n; i++)  {
      sum = b->ve[i];
      for(j = 0; j < n; j++)  {
         if(i != j)
            sum = sum - sp_get_val(A, i,j) * x->ve[j];
      }
      x->ve[i] = sum;
   }
   while(!done)    {
      for(i = 0; i < n; i++)  {
         oldx = x->ve[i];
         sum = b->ve[i];
         for(j = 0; j < n; j++)  {
            if(i != j)
               sum = sum - sp_get_val(A, i,j) * x->ve[j];
         }
         x->ve[i] = lambda*sum + (1.0 - lambda)*oldx;
         if((!done) && (x->ve[i] != 0.0))
            err = fabs((x->ve[i]-oldx) / x->ve[i]);
            if(err < tol)
               done = 1;
      }
      itr++;
      if(itr > maxIter) {
         done = 1;
         retVal = -1;
      }
   }
   return (retVal);
}
```

```
//(Note: only the code for degree 7 quadrature is presented
// here, other cases follow a similar method.)

/*****************************************************************
* function to determine integration points using 31-point,
*   degree 7 quadrature for a tetrahedron.
* I/O: A: 3x4 matrix containing tet coordinate points as column
*           vectors.
*      p: the number of the point (1-31)
*      X: 1x3 vector that will contain coordinates for this point
*              (must have been previously allocated).
*      w: varable that will contain the weight for the point.
*****************************************************************/

void cubature7(MAT *A, int p, VEC* X, double *w)   {
   double alpha, beta, gamma;

   if ((p >= 1) && (p <= 6))   {
      // midside points(6)
      *w = 0.000970017636684296702;
      alpha = 0.5;
      beta = 0.0;
   }
   else
      if (p == 7)   {
         // centre point(1)
         *w = 0.0182642234661087939;
         alpha = 1.0/4.0;
      }
      else
         if ((p >= 8) && (p <= 11))   {
            // 1st set of vertex points(4)
            *w = 0.0105999415244141609;
            alpha = 0.765360423009044044;
            beta = 0.0782131923303186549;
         }
         else
            if ((p >= 12) && (p <= 15))   {
               // 2nd set of centre points(4)
               *w = -0.0625177401143299494;
               alpha = 0.634470350008286765;
               beta = 0.121843216663904411;
            }
            else
               if ((p >= 16) && (p <= 19))   {
                  // 3rd set of centre points(4)
                  *w = 0.00489142526307353653;
                  alpha = 0.00238250666073834549;
                  beta = 0.332539164446420554;
               }
               else   {
                  // edge-midsid points(12)
                  *w = 0.0275573192239850917;
                  alpha = 0.2;
                  beta = 0.1;
                  gamma = 0.6;
               }
   switch(p)   {
   // midside points (6)
```

```
case 1:
  X->ve[0]=alpha*(A->me[0][0]+A->me[0][1])+beta*(A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*(A->me[1][0]+A->me[1][1])+beta*(A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*(A->me[2][0]+A->me[2][1])+beta*(A->me[2][2]+A->me[2][3]);
  break;
case 2:
  X->ve[0]=alpha*(A->me[0][0]+A->me[0][2])+beta*(A->me[0][1]+A->me[0][3]);
  X->ve[1]=alpha*(A->me[1][0]+A->me[1][2])+beta*(A->me[1][1]+A->me[1][3]);
  X->ve[2]=alpha*(A->me[2][0]+A->me[2][2])+beta*(A->me[2][1]+A->me[2][3]);
  break;
case 3:
  X->ve[0]=alpha*(A->me[0][0]+A->me[0][3])+beta*(A->me[0][1]+A->me[0][2]);
  X->ve[1]=alpha*(A->me[1][0]+A->me[1][3])+beta*(A->me[1][1]+A->me[1][2]);
  X->ve[2]=alpha*(A->me[2][0]+A->me[2][3])+beta*(A->me[2][1]+A->me[2][2]);
  break;
case 4:
  X->ve[0]=alpha*(A->me[0][1]+A->me[0][2])+beta*(A->me[0][0]+A->me[0][3]);
  X->ve[1]=alpha*(A->me[1][1]+A->me[1][2])+beta*(A->me[1][0]+A->me[1][3]);
  X->ve[2]=alpha*(A->me[2][1]+A->me[2][2])+beta*(A->me[2][0]+A->me[2][3]);
  break;
case 5:
  X->ve[0]=alpha*(A->me[0][1]+A->me[0][3])+beta*(A->me[0][0]+A->me[0][2]);
  X->ve[1]=alpha*(A->me[1][1]+A->me[1][3])+beta*(A->me[1][0]+A->me[1][2]);
  X->ve[2]=alpha*(A->me[2][1]+A->me[2][3])+beta*(A->me[2][0]+A->me[2][2]);
  break;
case 6:
  X->ve[0]=alpha*(A->me[0][2]+A->me[0][3])+beta*(A->me[0][0]+A->me[0][1]);
  X->ve[1]=alpha*(A->me[1][2]+A->me[1][3])+beta*(A->me[1][0]+A->me[1][1]);
  X->ve[2]=alpha*(A->me[2][2]+A->me[2][3])+beta*(A->me[2][0]+A->me[2][1]);
  break;
// centeriod
case 7:
  X->ve[0]=alpha*(A->me[0][0]+A->me[0][1]+A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*(A->me[1][0]+A->me[1][1]+A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*(A->me[2][0]+A->me[2][1]+A->me[2][2]+A->me[2][3]);
  break;
// vertex-centroid points set#1 (4)
case 8:
  X->ve[0]=alpha*A->me[0][0]+beta*(A->me[0][1]+A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][0]+beta*(A->me[1][1]+A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][0]+beta*(A->me[2][1]+A->me[2][2]+A->me[2][3]);
  break;
case 9:
  X->ve[0]=alpha*A->me[0][1]+beta*(A->me[0][0]+A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][1]+beta*(A->me[1][0]+A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][1]+beta*(A->me[2][0]+A->me[2][2]+A->me[2][3]);
  break;
case 10:
  X->ve[0]=alpha*A->me[0][2]+beta*(A->me[0][0]+A->me[0][1]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][2]+beta*(A->me[1][0]+A->me[1][1]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][2]+beta*(A->me[2][0]+A->me[2][1]+A->me[2][3]);
  break;
case 11:
  X->ve[0]=alpha*A->me[0][3]+beta*(A->me[0][0]+A->me[0][1]+A->me[0][2]);
  X->ve[1]=alpha*A->me[1][3]+beta*(A->me[1][0]+A->me[1][1]+A->me[1][2]);
  X->ve[2]=alpha*A->me[2][3]+beta*(A->me[2][0]+A->me[2][1]+A->me[2][2]);
  break;
// vertex-centroid points set#2 (4)
case 12:
```

```
  X->ve[0]=alpha*A->me[0][0]+beta*(A->me[0][1]+A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][0]+beta*(A->me[1][1]+A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][0]+beta*(A->me[2][1]+A->me[2][2]+A->me[2][3]);
  break;
case 13:
  X->ve[0]=alpha*A->me[0][1]+beta*(A->me[0][0]+A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][1]+beta*(A->me[1][0]+A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][1]+beta*(A->me[2][0]+A->me[2][2]+A->me[2][3]);
  break;
case 14:
  X->ve[0]=alpha*A->me[0][2]+beta*(A->me[0][0]+A->me[0][1]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][2]+beta*(A->me[1][0]+A->me[1][1]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][2]+beta*(A->me[2][0]+A->me[2][1]+A->me[2][3]);
  break;
case 15:
  X->ve[0]=alpha*A->me[0][3]+beta*(A->me[0][0]+A->me[0][1]+A->me[0][2]);
  X->ve[1]=alpha*A->me[1][3]+beta*(A->me[1][0]+A->me[1][1]+A->me[1][2]);
  X->ve[2]=alpha*A->me[2][3]+beta*(A->me[2][0]+A->me[2][1]+A->me[2][2]);
  break;
// vertex-centroid points set#3 (4)
case 16:
  X->ve[0]=alpha*A->me[0][0]+beta*(A->me[0][1]+A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][0]+beta*(A->me[1][1]+A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][0]+beta*(A->me[2][1]+A->me[2][2]+A->me[2][3]);
  break;
case 17:
  X->ve[0]=alpha*A->me[0][1]+beta*(A->me[0][0]+A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][1]+beta*(A->me[1][0]+A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][1]+beta*(A->me[2][0]+A->me[2][2]+A->me[2][3]);
  break;
case 18:
  X->ve[0]=alpha*A->me[0][2]+beta*(A->me[0][0]+A->me[0][1]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][2]+beta*(A->me[1][0]+A->me[1][1]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][2]+beta*(A->me[2][0]+A->me[2][1]+A->me[2][3]);
  break;
case 19:
  X->ve[0]=alpha*A->me[0][3]+beta*(A->me[0][0]+A->me[0][1]+A->me[0][2]);
  X->ve[1]=alpha*A->me[1][3]+beta*(A->me[1][0]+A->me[1][1]+A->me[1][2]);
  X->ve[2]=alpha*A->me[2][3]+beta*(A->me[2][0]+A->me[2][1]+A->me[2][2]);
  break;
// edge-midsid points(12)
case 20:
  X->ve[0]=alpha*A->me[0][0]+gamma*A->me[0][1]+
           beta*(A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][0]+gamma*A->me[1][1]+
           beta*(A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][0]+gamma*A->me[2][1]+
           beta*(A->me[2][2]+A->me[2][3]);
  break;
case 21:
  X->ve[0]=alpha*A->me[0][0]+gamma*A->me[0][2]+
           beta*(A->me[0][1]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][0]+gamma*A->me[1][2]+
           beta*(A->me[1][1]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][0]+gamma*A->me[2][2]+
           beta*(A->me[2][1]+A->me[2][3]);
  break;
case 22:
  X->ve[0]=alpha*A->me[0][0]+gamma*A->me[0][3]+
```

```
            beta*(A->me[0][1]+A->me[0][2]);
  X->ve[1]=alpha*A->me[1][0]+gamma*A->me[1][3]+
            beta*(A->me[1][1]+A->me[1][2]);
  X->ve[2]=alpha*A->me[2][0]+gamma*A->me[2][3]+
            beta*(A->me[2][1]+A->me[2][2]);
  break;
case 23:
  X->ve[0]=alpha*A->me[0][1]+gamma*A->me[0][2]+
            beta*(A->me[0][0]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][1]+gamma*A->me[1][2]+
            beta*(A->me[1][0]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][1]+gamma*A->me[2][2]+
            beta*(A->me[2][0]+A->me[2][3]);
  break;
case 24:
  X->ve[0]=alpha*A->me[0][1]+gamma*A->me[0][3]+
            beta*(A->me[0][0]+A->me[0][2]);
  X->ve[1]=alpha*A->me[1][1]+gamma*A->me[1][3]+
            beta*(A->me[1][0]+A->me[1][2]);
  X->ve[2]=alpha*A->me[2][1]+gamma*A->me[2][3]+
            beta*(A->me[2][0]+A->me[2][2]);
  break;
case 25:
  X->ve[0]=alpha*A->me[0][2]+gamma*A->me[0][3]+
            beta*(A->me[0][0]+A->me[0][1]);
  X->ve[1]=alpha*A->me[1][2]+gamma*A->me[1][3]+
            beta*(A->me[1][0]+A->me[1][1]);
  X->ve[2]=alpha*A->me[2][2]+gamma*A->me[2][3]+
            beta*(A->me[2][0]+A->me[2][1]);
  break;
case 26:
  X->ve[0]=alpha*A->me[0][1]+gamma*A->me[0][0]+
            beta*(A->me[0][2]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][1]+gamma*A->me[1][0]+
            beta*(A->me[1][2]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][1]+gamma*A->me[2][0]+
            beta*(A->me[2][2]+A->me[2][3]);
  break;
case 27:
  X->ve[0]=alpha*A->me[0][2]+gamma*A->me[0][0]+
            beta*(A->me[0][1]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][2]+gamma*A->me[1][0]+
            beta*(A->me[1][1]+A->me[1][3]);
  X->ve[2]=alpha*A->me[2][2]+gamma*A->me[2][0]+
            beta*(A->me[2][1]+A->me[2][3]);
  break;
case 28:
  X->ve[0]=alpha*A->me[0][3]+gamma*A->me[0][0]+
            beta*(A->me[0][1]+A->me[0][2]);
  X->ve[1]=alpha*A->me[1][3]+gamma*A->me[1][0]+
            beta*(A->me[1][1]+A->me[1][2]);
  X->ve[2]=alpha*A->me[2][3]+gamma*A->me[2][0]+
            beta*(A->me[2][1]+A->me[2][2]);
  break;
case 29:
  X->ve[0]=alpha*A->me[0][2]+gamma*A->me[0][1]+
            beta*(A->me[0][0]+A->me[0][3]);
  X->ve[1]=alpha*A->me[1][2]+gamma*A->me[1][1]+
            beta*(A->me[1][0]+A->me[1][3]);
```

```
      X->ve[2]=alpha*A->me[2][2]+gamma*A->me[2][1]+
                  beta*(A->me[2][0]+A->me[2][3]);
      break;
    case 30:
      X->ve[0]=alpha*A->me[0][3]+gamma*A->me[0][1]+
                  beta*(A->me[0][0]+A->me[0][2]);
      X->ve[1]=alpha*A->me[1][3]+gamma*A->me[1][1]+
                  beta*(A->me[1][0]+A->me[1][2]);
      X->ve[2]=alpha*A->me[2][3]+gamma*A->me[2][1]+
                  beta*(A->me[2][0]+A->me[2][2]);
      break;
    case 31:
      X->ve[0]=alpha*A->me[0][3]+gamma*A->me[0][2]+
                  beta*(A->me[0][0]+A->me[0][1]);
      X->ve[1]=alpha*A->me[1][3]+gamma*A->me[1][2]+
                  beta*(A->me[1][0]+A->me[1][1]);
      X->ve[2]=alpha*A->me[2][3]+gamma*A->me[2][2]+
                  beta*(A->me[2][0]+A->me[2][1]);
      break;
    default:
       printf("error in cubature: invalid point #\n");
    }
}
```