Implementation of Polyhedral Finite Elements with application to heat transfer

THESEUS FE

theseus-fe.com

Advisors
Dr. Matthias Lambrecht
David Franke

Group Members
Zoltan Jozefik
Karthik Chittepu
Sheema Kooshapur

Abstract

Finite element procedures are now an important and frequently indispensable part of engineering analysis and design. Finite element computer programs are widely used in practically all branches of engineering. It is a flexible numeric procedure both for interpolation and approximation of the solution of partial differential equations. The PDE is approximated by a linear Galerkin finite element, which uses piecewise linear weight and basis functions. The resulting integral equation induces a system of linear equations, which needs to be solved with appropriate methods.

With the increase in the complexity on engineering problems the necessity for the extension of the standard elements has come into light. The arbitrary polyhedral finite elements lead to a greater flexibility in domain decomposition. A general simple local coordinate system, the natural element coordinates, was developed, which makes a formulation of interpolation functions on polyhedral elements possible that are independent of the dimension of the space, of the localization and vertex number.

The purpose of the project is to develop polyhedral finite elements, by implementing a numerical method to calculate the interpolation function at a particular point and integrating it with a finite element code for the calculating of unknowns. Reliability of these elements is tested by implementing them for heat transfer problems and visualizing the results and by comparing it with the analytical results.

Keywords: Finite element method, polyhedral elements, Voronoi decomposition, Heat transfer

Contents

List of Figures

Chapter 1

Introduction

Finite element procedures are now widely used in engineering analysis, and we can expect this use to increase significantly in the years to come. The development of finite element methods for the solution of practical engineering problems began with the advent of digital computer. That is, the essence of a finite element solution of problem is that a set of governing equations is established and solved, and it is effectively done through the use of digital computer. Finite element approximation is done by converting the continuous problem to a discrete problem. Approximation is done by subdividing the entire domain of a boundary value problem into sub–areas, so called finite elements. The approximation inside the element is done by Galerkin approximation method which uses linear weighting functions to approximate the continuous problem. The linear weighting functions are the shape functions in the case of Galerkin method. Shape functions for each node are calculated based on influence area of the node.

Many different types of elements have been developed for the wide range of applications. Most formulations are based on the decomposition in form of triangles or quadrilaterals. With the increase in the complexity of the engineering problem the necessity for the alternative elements has increased. Therefore, necessity for the arbitrary polyhedral has come into picture. The development of an arbitrary polyhedral element is done by using Voronoi decomposition strategies. Voronoi diagram is a simple mathematical object that determines the nearest neighbour decomposition for a set of points in Euclidean space.

Final goal of the finite element method is to calculate the unknowns. This is done by evaluating the stiffness matrix. The stiffness matrix is calculated by integrating the weak form of the differential equation which defines the problem. Integration is done by the Gaussian Quadrature. It is an approximation for the definite integration of a function, usually stated as a weighting sum of the values at specified points within the domain of integration. Therefore the shape functions are estimated at specified points called gauss points.

Finally the unknowns are determined for a heat transfer problem and visualized using GID. The calculating of shape functions code is written in C++ and Fem code is coded in FORTRAN. The Numerical results obtained from FEM are compared with the analytical solution.

Chapter 2

Finite Element Approximation of polyhedrals

We focus on the numerical approximation of the solution $u(x)$ of a partial differential equation with the finite element method. It is done by the Galerkin procedure,

$$u(x) \cong \tilde{u}(x) \equiv \sum_{i=1}^{n} u_i \phi(x) \tag{2.1}$$

Interpolation functions on classical finite elements are defined on triangles and quadrilaterals. Hence, the Interpolation functions are well known. The application of arbitrary polyhedrons as basic finite elements needs a generalized formulation of these interpolation functions. This is done numerically by Voronoi decomposition.

2.1 Voronoi Decomposition

Voronoi decomposition is a simple mathematical procedure that determines the nearest neighbour decomposition for a set of points in Euclidean space. Decomposition is done by drawing the perpendicular bisectors. Decomposition is of two orders,

2.1.1 Voronoi region of first order

First order decomposition is initial decomposition of the polyhedral, done by finding out influence area of each node. This is done by drawing the perpendicular bisector of two nodes as shown in the Figure 2.1. The same procedure is extended to the polyhedral as show in the Figure 2.2.

The area which is bounded by the blue lines indicates influence area of node 5. This influence area represents the set of all points which are closer (or at least equally close) to node 5 than they are to all other nodes of the element. This is mathematically given by,

$$VR(\varepsilon^i) := \left\{ p \in \Re^n : d(p, \varepsilon^i) \leq d(p, \varepsilon^j) \forall j \neq i \right\} \tag{2.2}$$

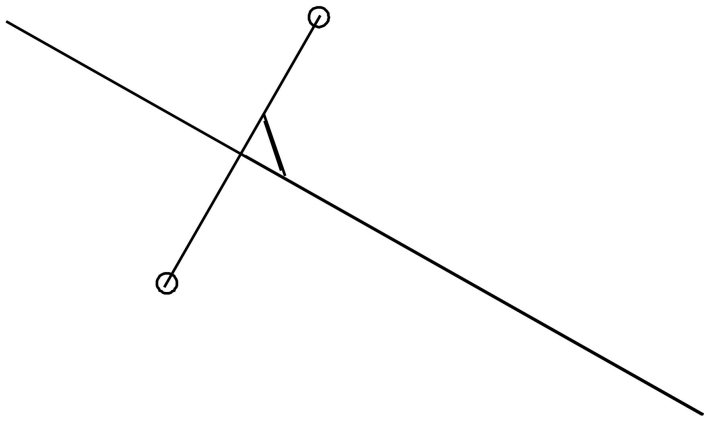

Figure 2.1: Voronoi decomposition of two nodes

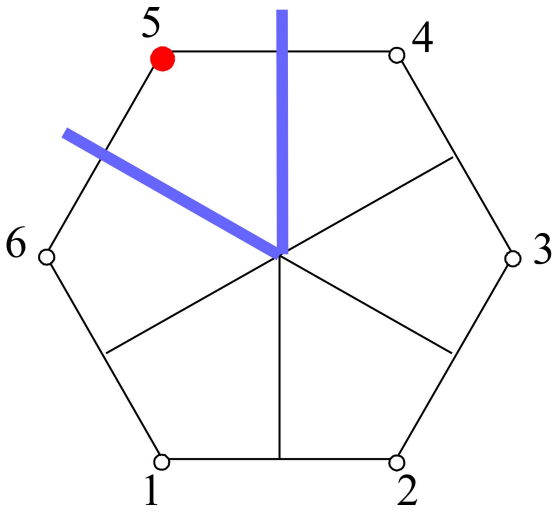

Figure 2.2: Voronoi decomposition of first order

2.1.2 Voronoi region of second order

The second order decomposition will result in an influence area of an arbitrary point inside the element. For this purpose the perpendicular bisectors between each node and the arbitrary point in the element are drawn. As shown in Figures 2.3 and 2.4 these bisectors will define an area around the point which is in fact the influence area of the point. Influence area is nothing but a set of points which are closer to the point of consideration than any other node.

In order to find the shape function value i of this point the intersection area of the influence region of the point and the influence region of node i is calculated as shown in Figure 2.5. Thus obtain area is a set of points which are closer to point of consideration than node i and also the set should be much closer to node i than any other node. This is given mathematically by equation 2.3.

$$VR(x, \varepsilon^i) := \left\{ p \in \Re^n : d(p, x) \leq d(p, \varepsilon^i) \leq d(p, \varepsilon^j) \forall j \neq i \right\} \tag{2.3}$$

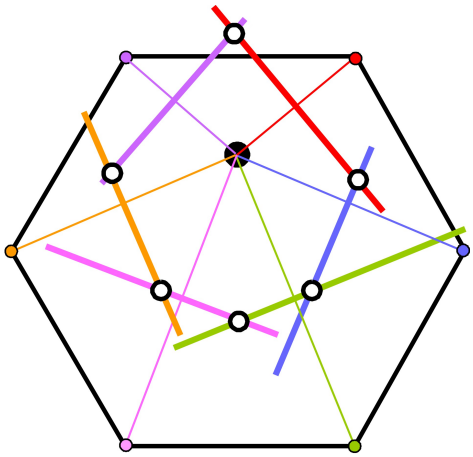

Figure 2.3: Voronoi decomposition of second order

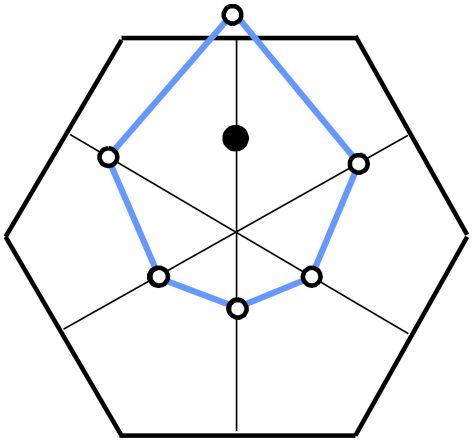

Figure 2.4: First and Second order voronoi regions

This area is then divided by the whole influence area of the point to obtain the shape function. This is shown in Figure 2.5 and can be represented by equation 2.4 (shape function). It can easily be seen that the sum of all the shape functions for a certain node is equal to one.

$$N_i = u(VR(x, \varepsilon^i)) / \sum u(VR(x, \varepsilon^i)) \tag{2.4}$$

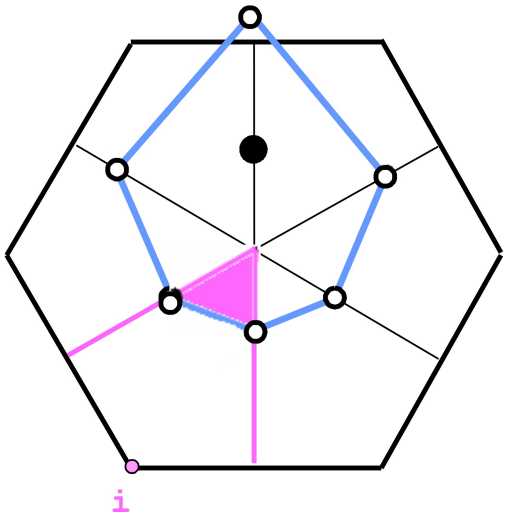

Figure 2.5: Intersection of Voronoi decomposition of first and second order

Shape Functions obtained from the following method are used in the FE code to calculate the unknowns by using new element types

2.2 Derivatives of shape functions

Derivatives of shape functions are necessary in the calculation of the Jacobian and strain displacement matrix. The derivatives are calculated using the central difference method as shown in the Figure 2.6 and mathematical equation is given in equation 2.5.

$$\frac{df}{dx} = (f(x+h) - f(x-h))/(2 \cdot h) \tag{2.5}$$

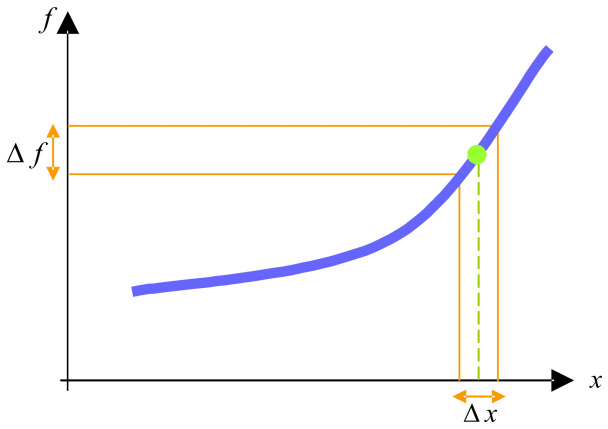

Figure 2.6: Central difference method

2.3 Procedure

The program written in C++ for finding the shape function values of a polyhedral element uses another code called voronoi which gives the vertices of the voronoi region of the second order for an arbitrary point inside the element. The procedure is as follows:

1. Calculation of shape function values begins with the program asking the user to enter the number of edges of the arbitrary polygon for which they have to be calculated.

2. Next the program reads a text file called '*duty text*' which is created by the user. This text file contains the value which is read by the program to know whither to do one gauss point per triangle or three or to do calculation for a grid point. The initial polyhedral is divided into triangles by connecting the vertices and the mid point of the circumscribed circle of the polyhedral for calculation of gauss points. If the value in the duty text is '*1*' then the program calculate shape function values for one gauss point per triangle and if it is '*3*' then it does for three gauss points per triangle else it is done for a grid point. The grid points are used to visualize the shape functions of every arbitrary polyhedral.

3. The header is divided into one class and two functions which contain the

- Polyhedral class which calculate the vertices, gauss points and shape function values and derivatives for a gauss point of a polyhedral.
- Grid function which calculate the grid points and shape functions and derivatives for each grid point of a polyhedral.
- GP function which writes the output file for the gauss points which contains shape function values and derivatives of each gauss point.

4. Class Polyhedral first constructs the vertices of the polyhedral on the circumscribe circle method using cosine and sin functions.

5. Then the gauss point (centroid in the case of one gauss point) is calculated for each triangle (it contain two adjacent vertices and the centre of circumscribed circle) by summing up the vertices and dividing it by three.

6. Thus calculated vertices and one specific gauss point each time is written in a text file called '*b.txt*'.

7. Then the shape function calls a program called `voronoi` which requires the '*b.txt*' file as the input file. This voronoi code decomposes the polyhedral and gives an output text file called '*hexagon.txt*'. This text file contains the vertices of second order decomposition, equation of the lines of second order decomposition, vertices of the polyhedral and the point for which it is calculated.

8. The obtained text file '*hexogon.txt*' is called by a function `shpfn` to save the coordinates of vertices and equations of lines in respective matrices.

9. Then the shape function values are calculated by calling a function `calculate`. This is done by first finding out the angles of the line going through the centre of circumscribed circle of the polyhedral and the vertices of second order decomposition and then arranging the vertices in the increasing order of there angles. Then the area of influence of gauss point for each vertex is calculated. Then the shape function is calculated by dividing the area of influence for a particular gauss point for a particular vertex by the total area of influence of that particular gauss point as shown in the Figure 2.5. It is given by equation 2.4.

10. Then the derivatives are calculated by central difference. Four points at a distance of .0001 from gauss point are taken and shape functions are calculated for these points and then the difference in shape function of each two point is calculated and divided by the distance between them to obtain the derivative as shown in Figure 2.6. this is given by equation 2.5.

11. To visualize the result of the shape function we call the shape function at every grid point by calling the function `grid`. This function calculates the grid points and shape function and derivatives of each grid point and gives out a text file called '*grid.flavia.msh*' which contains shape function and derivates of each grid point and coordinates of grid points and is written in such a way that can be visualized by using GID.

Shape function values, derivatives, gauss points and gauss weights thus obtained are inserted in the FE code. This code reads the geometry file, which contains the element type, nodal coordinates of elements and boundary elements. This data is used in estimating the Jacobean and strain–displacement matrices, which are then used in the calculation of the stiffness matrix using Gaussian quadrature. After finding the stiffness and force matrices they are solved by conjugate gradient method to obtain the unknowns.

To test the reliability of the elements, these elements are applied to a heat transfer problem. The heat transfer problem is solved numerically and compared with analytical solution to check the reliability.

2.4 Gaussian Quadrature

Calculation of the stiffness matrix involves the integration of the weak form of PDE of the problem. Since the definite integration is very cumbersome, it is approximated by Gaussian quadrature. This is an approximation for the definite integration of a function, usually stated as a weighting sum of the values at specified points within the domain of integration. Therefore the shape functions are estimated at specified points called gauss points. These are also the points at which result values (e.g. Stress, temperature, displacement...) can be recovered. The points are then multiplied by a weighting factor. The gauss weights represent the volume of influence of the gauss points. An n–point Gaussian quadrature rule is constructed to yield an exact result for polynomials of degree $2n-1$, by a suitable choice of the n points x_i and n weights w_i for the one dimensional case. A program that subdivides a polyhedral of radius one with an arbitrary N number of nodes into N triangles and within each triangle computes the location for one or three gauss points is written. For the single gauss point triangle, the point is located at the triangles centroid as shown in Figure 2.7. Mathematically this is represented by equation 2.6.

$$\int_{-1}^{1} f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{2.6}$$

As it is seen in the Figure 2.8 that the shape function at the node is one and it constantly decease as you move away from the node. These values at each gauss point are used as the weighting function in Gaussian quadratic method instead of integration.

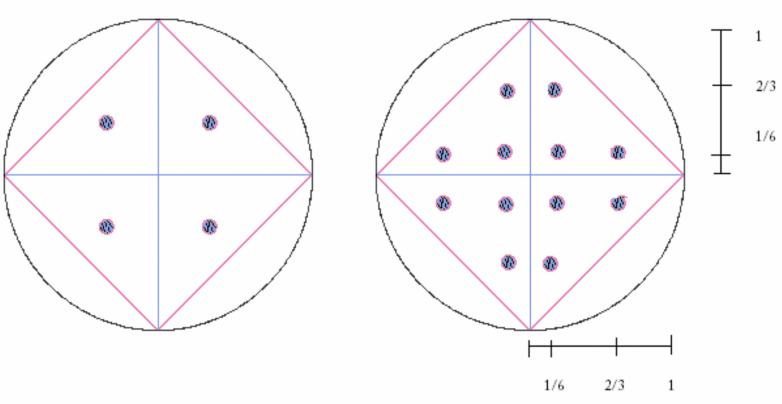

Figure 2.7: Gauss points for a 4 noded polyhedral

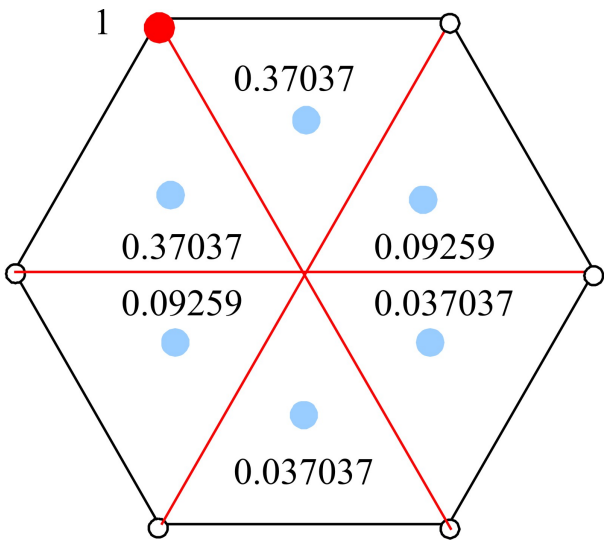

Figure 2.8: Weighting functions at each gauss point

Chapter 3

Visualization of shape function

For the visualization of the shape function for the 6–noded element C++ a program is written which places a fine grid over the element and provides the x and y coordinates of the grid points so that shape function values at these points can be calculated. This step was done purely to obtain a smooth visualization of the shape function and no other values are calculated at these grid points. The program works by taking advantage of the symmetry of elements and only the points in first quadrant are calculated. A function of the edge line is found from the provided nodes, and step incrementally along it. Since number of grid points is retained in each line, the mesh becomes finer as we move north and south away from the centre. This increase in mesh refinement, does not affect the accuracy of our results, because to even at the coarser places we have considered what is considered as a very fine mesh and secondly we are mostly interested in the values near the edges and nodes. Due to the way the FE code works, shape function values can't be calculated directly on the edges or the nodes. In the second, third and fourth quadrants, the values of the first quadrant are modified by giving suitable negative signs. In the Figure 3.1, the grid and the resulting shape function for the top right node is to be seen. At that node the value is one and at all other corner nodes the value is zero. The shape function is linear between edge nodes and it seems to have an exponential decay within the domain.

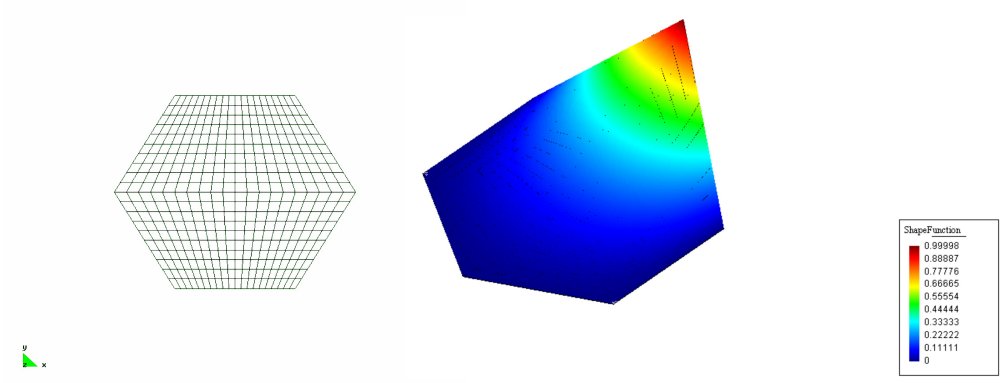

Figure 3.1: Visualization of shape function of 6–noded element

Chapter 4

Conjugate gradient

The FE code provided, previously used a commercial matrix solver called UMFPACK to solve the linear system of equation $A \cdot x = B$. UMFPACK uses a direct solver method and do to the memory allocation required, for large A matrixes computers ran out of allocable memory. For this reason a conjugate–gradient solver was written. The conjugate gradient method is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive definite. With the finite element method this requirement is always insured. The conjugate gradient method is an iterative solver, so it can be applied to sparse systems which are too large to be handled by direct method solvers such as UMFPACK. We use a Jacobi–preconditioner that sets the diagonal values of matrix A equal to 1. This helps lower the condition number and decrease computation time. To asses convergence, we check that the error norm and that the norm of the change in solution are less than $1.0 \cdot 10^{-9}$. After convergence is reached, we uncondition the solution vector to get the true values. The basic algorithm is presented below.

`Conjugate Gradient algorithm`

$$\begin{aligned} r_0 &:= b - A \cdot x_0 \\ p_0 &:= r_0 \\ k &:= 0 \end{aligned}$$

`repeat`

$$\begin{aligned} \alpha_k &:= \frac{r_k^T r_k}{p_k^T A p_k} \\ x_{k+1} &:= x_k + \alpha_k p_k \\ r_{k+1} &:= r_k - \alpha_k A p_k \end{aligned}$$

`If` $r_k + 1$ is 'sufficiently small' `then` exit loop `end if`

$$\begin{aligned} \beta_k &:= \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \\ p_{k+1} &:= r_{k+1} + \beta_k p_k \\ k &:= k + 1 \end{aligned}$$

```
end repeat
```

The result is x_{k+1}

Chapter 5

Mesh Generation

To obtain meshes, a program to mesh an arbitrary sized rectangle with adjustable mesh density for the six and eight–noded elements is written. Both the six and eight–noded meshers work by also using three and four noded elements (secondary elements) to fill in the domain in addition to the larger six or eight-noded primary elements used. The length and width parameters of the rectangle and an integer value that is then related to mesh density can be adjusted. The function then prints a *.nas* file that is read in by the FE code. Boundary condition should be manually added at the bottom of the mesh code. Meshes for the five, seven, nine and ten noded elements were provided using STAR-City with a fine and coarse mesh density. Figure 5.1 shows the visualization of the two different meshes. We used GID for the visualization, which only recognizes three and four noded elements, their fore we manually drew the six and eight noded shapes. In both meshers the primary elements are translated to new locations, but not rotated. There are many different possible ways to arrange the primary elements. Our goal was to limit the number of secondary elements and to have as much node to node and surface to surface contact between the primary elements. This is done to ensure that we get a true understanding of the behaviour of the element, its advantages or disadvantages without having the influences of the secondary elements.

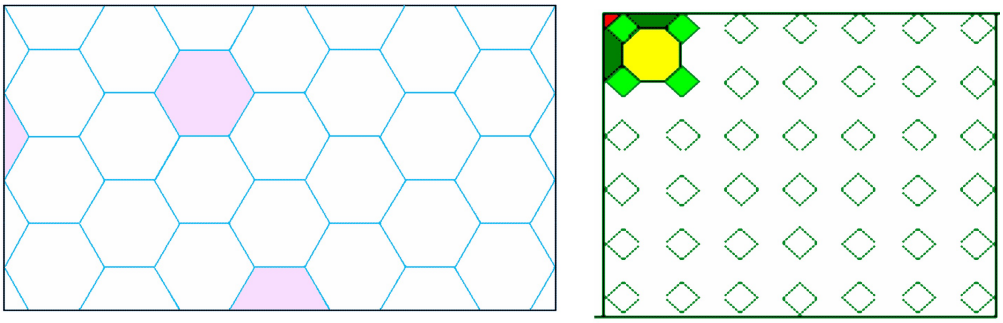

Figure 5.1: Visualization of 6 and 8 noded meshes

Chapter 6

Serendipity element

Serendipity elements are elements with mid side nodes. These elements lead to very accurate results at the edges. The main advantage of the serendipity elements is that since the internal nodes of the higher–order Lagrange elements do not contribute to the inter–element connectivity, the elimination of internal nodes results in reductions in the size of the element matrices.

In this project shape functions were found for hexagon shaped 6 noded serendipity elements. For each element 6 gauss points were considered. The output of the voronoi code in case of the serendipity element had to be read by the code and then corrected because the voronoi code produces some extra points which are not vertices of the voronoi region of second order. The vertices of the Voronoi region of second order should lie on the lines defining the decomposition of first order. However, the output of the Voronoi code is not always so; it sometimes has an error of about 10^{-6}. Also, finding the intersection area of the influence regions of a node and of a point in the element is more complicated in the case of serendipity elements because the influence area of each node of the serendipity element is more complicated in this case.

6.1 Procedure of the code

The number of nodes of the element and the coordinates of these nodes are written by the user in the '*a.txt*' file.

First the number of nodes should be written in the first line, and then the x and y coordinate of each node with a space in between. The nodes should be in a counter clockwise order.

The code starts by making a polyhedral object called *serendipity* using '*a.txt*'.

This object consists of a number of points which define the vertices of the element and an integer number which shows the number of vertices.

Then it calls the function `GP`: giving it as an input the object *serendipity*.

The function `GP` opens the text file '*resultsGP6*' for writing the results. Then it enters a *for loop*

In every step of this loop only one gauss point is dealt with.

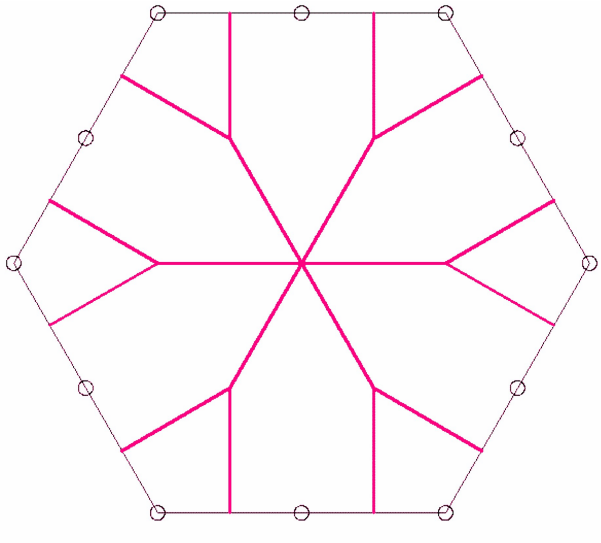

Figure 6.1: First order voronoi region

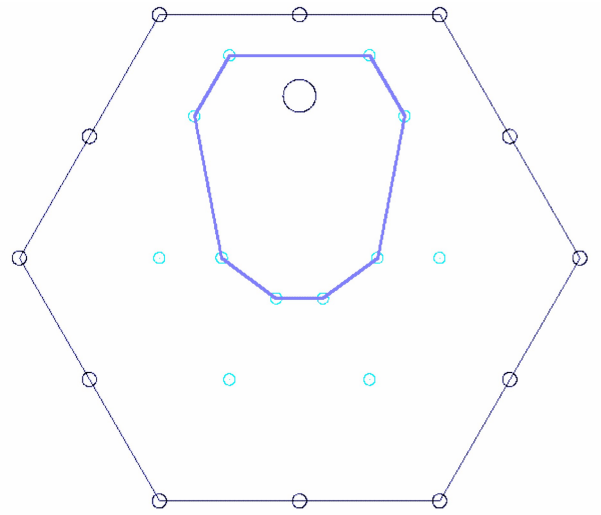

Figure 6.2: Second order voronoi region and the extra vertices given by the voronoi code for a point in the element

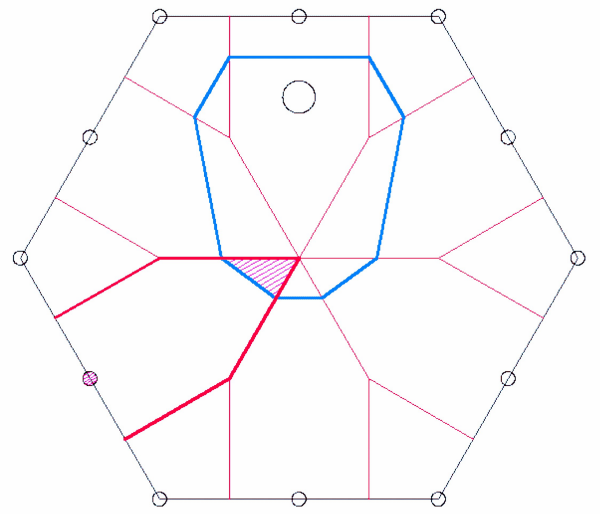

Figure 6.3: Intersection area of Second order voronoi region and the influence area of a certain node

In every step j of this loop first, the gauss point number j is calculated and the coordinates of it are written in '*resultsGP6.txt*'.

Then the code writes the coordinates of the nodes of the serendipity element and the coordinates of the gauss point together in a text file called '*b.txt*' in a certain order.

In the next step the voronoi code is called and the '*b.txt*' is given to it as input. The output is a text file called '*HEXAGON.txt*'.

Now another polyhedral object called '*second_decompose*' is constructed using a certain constructor which is capable of reading only the necessary information in the '*HEXAGON.txt*'.

The *second_decompose* polyhedral is then given to function `omit` which omits the extra points and only keeps the ones which can define the voronoi decomposition of second order. Then the new *second_decompose* is given to the function `putinorder` which orders the points in a counter clockwise manner.

In this way the second order decomposition for gauss point j is obtained as a set of points which define the vertices of this area.

Then the code enters another loop. In each iteration q of this loop, influence area q for gauss point j is calculated and stored in the array `ShapeFunction[q]` which is of type double. Also the sum of the influence areas for different q's will be found in this loop.

In this loop the function `ShapeFnc` is called. This function takes as input, the *second_decompose* polyhedral, the number of the shape function which has to be calculated q and the *serendipity* polyhedral which is in fact the actual serendipity element.

In the function `ShapeFnc` first an arbitrary polyhedral is constructed which is called *first_decompose*. Then this polyhedral together with *serendipity* and q is given to the void function `decompose1`. The function `decompose1` changes *first_decompose* in such a way as to make it the decomposition of first order for

node q of the element *serendipity*.

In the next step *second_decompose* and *first_decompose* are both given to function `find_intersect` which returns a polyhedral called *intersection* which is in fact the intersection between the two polyhedrals *second_decompose* and *first_decompose*.

Then the polyhedral *intersection* is also put in counter clockwise order using the function *putinorder*.

Finally by using the `PolygonArea` function the area of the polyhedral *intersection* is found. This is in fact the influence area of our gauss point j with respect to node q of our *serendipity* element. The area is then returned as a *double* value and we exit the function `ShapeFnc`. We also exit the loop we were in. In this loop the influence area of gauss point j was calculated with respect to every node q of the *serendipity* element. Also the sum of all influence areas was found.

Now the code enters another loop which finds shapefunction q in every iteration q by deviding the influence area `ShapeFunction[q]` by the sum of all influence areas. Then the `shapefunction` values obtained are written into text file '*resultsGP6.txt*'.

In the next part four points named 'a', 'b', 'c' and 'd' are made which are located on the left and right, above and below the gauss point j. the distance of these points to the gauss point is 0.001. The shape function values are found for each one of these points using the same procedure as the gauss point. Then using shape function values at these points the derivatives of shape function at gauss point j is computed using a central difference method. All the derivatives in x and y direction are then written into '*resultsGP6.txt*'.

Now all the operation for gauss point j is finished and all the shape functions and derivatives of shape functions are found. So the code goes back to the beginning of the first for loop, calculates the gauss point for the next j and repeats the same procedure.

Chapter 7

Results

To test our work and the finite element program we will compare three different meshes applied to heat transfer problem with analytical results. The first mesh is a regular 8–noded mesh. The second mesh contains a mix of four, five, six, seven and eight–noded elements. Our third mesh is a standard quad element mesh used to check if the new elements provide a difference. At this point our goal is not to assess which element, the four, five, six–noded, performs best, but to assess weather the elements work and produce meaningful results.

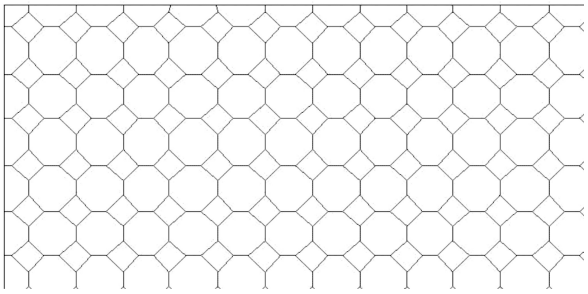

Figure 7.1: Regular 8–noded

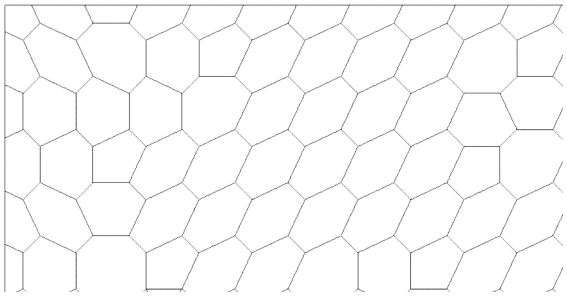

Figure 7.2: Mixed mesh

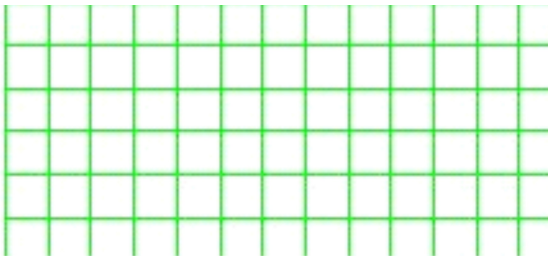

Figure 7.3: Mixed mesh

7.1 Example 1

First example is a 1D, steady-state heat conduction problem. We have a uniform wall with internal heat generation and temperature boundary conditions set at 20°C. We check results at three different locations in the wall and compare them to their corresponding analytic solution. We can see that the three different meshes provided similar accuracy of within plus/minus 1 degree of analytical solution. This is a relatively simple problem, but we can conclude that the finite element program and the different elements are working properly.

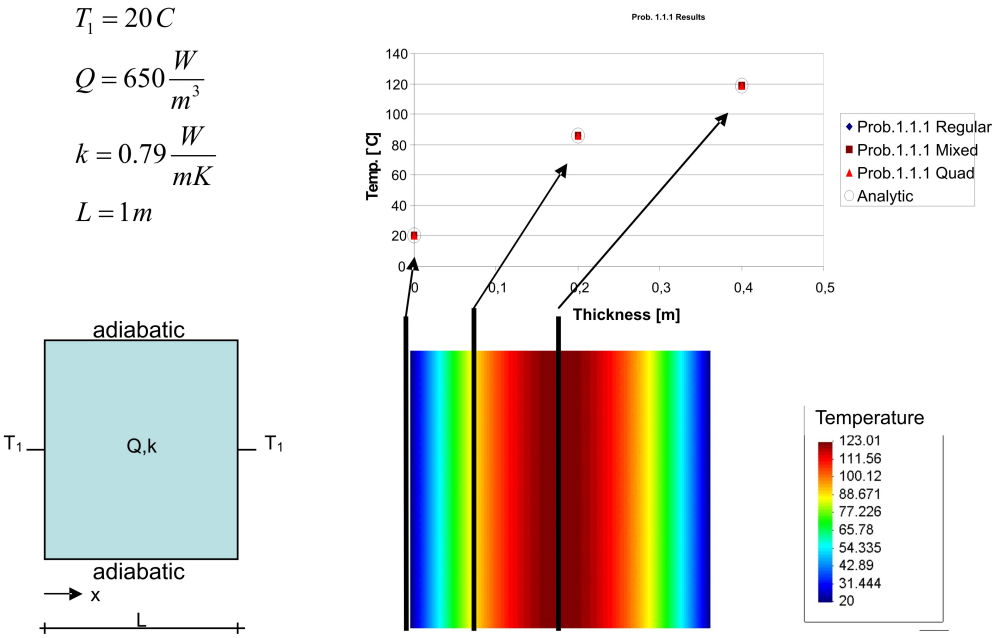

Figure 7.4: Heat transfer problem 1

7.2 Example 2

Second example is also a 1D, steady–state heat transfer problem. We have a composite wall with internal heat generation in the first section and convection boundary conditions set at the right side. Results checked at several different

locations in the wall and compare them to their corresponding analytic solution. Closely looking at the results, we can see that the quad mesh is more consistently centred within the analytical solution then the multi-noded element meshes. We don't draw any negative conclusions from this because all results are with in one degree accuracy and from this example we can further conclude that the finite element program and the different elements are working properly.

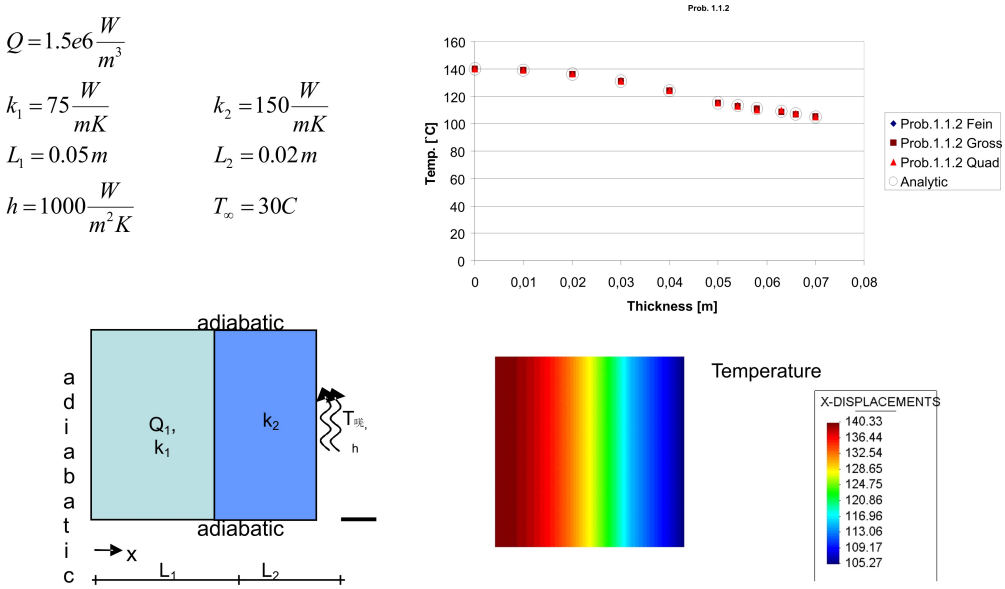

Figure 7.5: Heat transfer problem 2

7.3 Example 3

Final example is a 1D transient heat conduction problem. A homogeneous wall with initial temperature of 50°C is analyzed after 5 seconds as it cools to the temperature boundary conditions of 20°C. The motivation behind using higher–noded elements is that they are thought to perform better in regions of high gradients. In this example, the rapid temperature change with in the first few initial seconds produces a high temperature gradient, and we can see from the results that the higher–noded meshes are more accurate in the initial high gradient region than the quad mesh. We don't form yet any hard conclusions from this, but we take it as a positive sign and will need to make more comparisons to assess the validity of this observation.

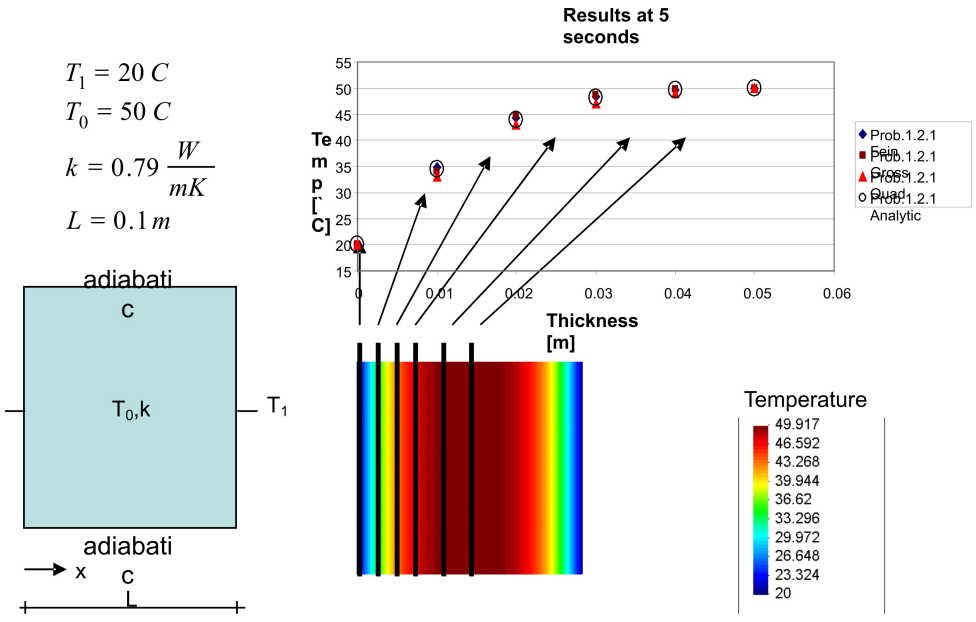

Figure 7.6: Heat transfer problem 3

Chapter 8

Conclusions

In conclusion, through this project we developed and implemented 5–10 noded polyhedral elements in a finite element program and checked their validity in heat transfer problems against analytic solutions. From the results, we found that the elements perform as accurately regular quad elements. To draw further conclusions about their performance, further careful testing is needed such as; adjusting mesh density, providing uniform meshes and testing them in different problems.