Applications of Generic Interpolants In the Investigation and Visualization of Approximate Solutions of PDEs on Coarse Unstructured Meshes

by

Hassan Goldani Moghaddam

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Applications of Generic Interpolants In the Investigation and Visualization of Approximate Solutions of PDEs on Coarse Unstructured Meshes

Hassan Goldani Moghaddam

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2010

In scientific computing, it is very common to visualize the approximate solution obtained by a numerical PDE solver by drawing surface or contour plots of all or some components of the associated approximate solutions. These plots are used to investigate the behavior of the solution and to display important properties or characteristics of the approximate solutions. In this thesis, we consider techniques for drawing such contour plots for the solution of two and three dimensional PDEs. We first present three fast contouring algorithms in two dimensions over an underlying unstructured mesh. Unlike standard contouring algorithms, our algorithms do not require a fine structured approximation. We assume that the underlying PDE solver generates approximations at some scattered data points in the domain of interest. We then generate a piecewise cubic polynomial interpolant (PCI) which approximates the solution of a PDE at off-mesh points based on the DEI (Differential Equation Interpolant) approach. The DEI approach assumes that accurate approximations to the solution and first-order derivatives exist at a set of discrete mesh points. The extra information required to uniquely define the associated piecewise polynomial is determined based on almost satisfying the PDE at a set of collocation points. In the process of generating contour plots, the PCI is used whenever we need an accurate approximation at a point inside the domain. The direct extension of the both DEI-based interpolant and the contouring algorithm to three dimensions is also

investigated.

The use of the DEI-based interpolant we introduce for visualization can also be used to develop effective Adaptive Mesh Refinement (AMR) techniques and global error estimates. In particular, we introduce and investigate four AMR techniques along with a hybrid mesh refinement technique. Our interest is in investigating how well such a 'generic' mesh selection strategy, based on properties of the problem alone, can perform compared with a special-purpose strategy that is designed for a specific PDE method. We also introduce an à posteriori global error estimator by introducing the solution of a companion PDE defined in terms of the associated PCI.

Dedication

I dedicate my thesis to my parents and family, for their understanding, support and unconditional love. Without these things this thesis could not have been possible.

Acknowledgements

This thesis would be incomplete without a mention of the support given me by my supervisor, Professor Wayne Enright. I would like to express my sincere appreciation for his guidance and insight throughout the research.

Special thanks go to my committee members, Professor Ken Jackson, Professor Christina Christara and Professor Tom Fairgrieve for their valuable suggestions and comments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Our physical world is often described by a set of laws that describes how a system evolves over time. These physical laws can often be stated in the form of Partial Differential Equations (PDEs) describing how the components representing the system change over time. Such PDEs can provide a precise mathematical model of physical phenomena, and they make up one of the most widely used tools of applied mathematics. For the vast majority of real world problems, the underlying PDE used to model the phenomenon does not have a closed form solution. In these cases, effective numerical methods can be used to approximate the solution at a discrete set of mesh points in the domain associated with the problem definition. After approximating the solution, one is often interested in visualizing some properties of the solution by rendering important properties or characteristics of the approximate solution. In such a case, a typical coarse mesh approximation is usually not sufficient to ensure that such visualization will appear continuous and smooth on a high resolution medium. One approach to address this difficulty is to determine approximations using the PDE solver on a mesh that is fine enough to render the solution in a 'smooth' fashion. However, this approach often results in much

more accuracy than what is needed and can be very expensive. An alternative approach that is often employed is based on the use of a multivariate spline to define a piecewise interpolant which can then be evaluated to determine the data values required for the fine mesh.

Although spline interpolation can work quite well in many cases, a more efficient approach to generate a piecewise approximate with comparable accuracy has been introduced by Enright to approximate the solution at arbitrary points in the domain of interest [12]. The approach, called Differential Equation Interpolant (DEI), is generic and can be applied to a large class of methods and problems in two and three dimensions. The idea is to associate an 'independent' (using local information only) multi-variate polynomial with each mesh element. The collection of such polynomials, over all mesh elements, defines a piecewise polynomial approximation. It assumes that accurate approximations to the solution and first-order derivatives exist or can be computed at a small additional cost at the set of discrete mesh points. In the case that extra information is required to uniquely determine the associated piecewise polynomial (on a given mesh element), the DEI approach does this by sampling the PDE at a small set of points and solving a small system of linear equations.

1.2 Objective

In the numerical solution of ODEs, once continuous extensions of discrete Runge-Kutta formulas were introduced (CRKs), they were used to develop reliable event location (points where the CRK, $S(x)$, takes on a prescribed value). These piecewise polynomials, $S(x)$ were then also used to develop mesh refinement schemes for BVPs, and global error estimates for ODEs. In this thesis, we investigate to what extent DEIs can be used to develop similar generic strategies or techniques that can produce effective visualization, mesh refinement and error estimation schemes that are suitable for use with some PDE

solvers on some problems.

In the ODE case, for the use of CRKs to be effective we assume that the exact ODE solution is at least p times differentiable (for a p^{th} order CRK). While the continuous piecewise polynomial extension, $S(x)$ needs to only be C^0 differentiable, it is often in C^1 and this can be an advantage. We make a similar assumption when DEIs are applied to investigate the approximate solution of PDEs. That is, they will be effective when the underlying exact solution is sufficiently differentiable but won't be effective if the true solution is not sufficiently differentiable.

Our main focus is to plot contour curves in 2D and level sets in 3D to visualize the approximate solution. We introduce three fast contouring algorithms for visualizing the approximate solution of two dimensional PDEs. We then investigate extending the approach to three dimensions and present a more detailed justification and implementation for one of them. Unlike standard contouring approaches, our contouring algorithms do not require a fine mesh approximation and work efficiently with the original scattered data.

In the numerical solution of Partial Differential Equations by standard methods the problem domain must be decomposed into a grid or mesh. A grid is a union of simple geometric elements such as quadrilaterals or triangles in two dimensions and tetrahedra in three dimensions. The spacing of the grid points affects the efficiency and the accuracy of the approximate solution. The spacing also determines the number of unknowns associated with the approximate solution and therefore the storage requirement which affects the cost of the computation. Although for well-behaved smooth problems a uniform mesh might give satisfactory results, there are problems where the solution is more difficult to approximate in some regions than in others. One could use a uniform grid with mesh elements fine enough to satisfy the required accuracy in these difficult regions. A better approach is to use a fine resolution in the difficult regions and a coarser resolution elsewhere. This approach is called Adaptive Mesh Refinement (AMR) and

can be very useful in saving computational resources particularly when solving problems modeling multi-scale phenomena.

The process of transforming a continuous model of a physical phenomena characterized by Partial Differential Equations into a discretized set of equations naturally looses information. No matter how appropriate a mathematical model is, the computational results will include some errors. The approximation error is often difficult to estimate or evaluate by heuristic approaches. However, it can be estimated (at least roughly) by a post-processing technique (an à posteriori error estimate) using a DEI and we will investigate an example of this approach.

1.3 Outline

In the following sections, we present an overview of the DEI generic approach followed by a detailed discussion of our two and three dimensional DEI-based interpolants for unstructured triangular/tetrahedronal meshes. In Chapter 2, we present an effective generic scheme for contouring in 2D and 3D based on DEI. We will first introduce three fast contouring algorithms for visualizing the approximate solution of two dimensional PDEs. We will then investigate extending the approach to three dimensions and present a detailed investigation and implementation for one of them. Unlike standard contouring approaches, our contouring algorithms do not require a fine mesh approximation and work efficiently with the original scattered data. In Chapter 3, we will discuss a 'generic' Adaptive Mesh Refinement approach which attempts to adapt the mesh based on the properties of the coarse mesh approximate solution (without requiring details of the underlying PDE method). In Chapter 4, we will introduce an à posteriori error estimator for the error associated with the DEI using the idea of a companion equation based on the original PDE. In Chapter 5, the numerical results will be presented. In Chapter 6, we will summarize the thesis and present some areas for future work.

1.4 Scattered Data Interpolation

Scattered data interpolation (SDI) is concerned with the approximation or representation of mathematical objects using samples taken at an unorganized set of discrete points, a scattered *point cloud*. The mathematical object may for instance be the boundary of a solid body, the graph of a scalar field, or the solution of a partial differential equation.

SDI refers to the problem of fitting a smooth surface through a scattered, or nonuniform, distribution of data samples. This subject is of practical importance in many science and engineering fields, where data is often measured or generated at sparse and irregular positions. The goal of interpolation is to reconstruct an underlying function (or a surface) that may be evaluated at any desired set of positions. This serves to smoothly propagate the information associated with the scattered data onto all positions in the domain. There are three common sources of scattered data: measured values of physical quantities, experimental results, and values computed from computer simulations. They are found in diverse scientific and engineering applications. For example, nonuniform measurements of physical quantities are collected in geology, meteorology, oceanography, cartography, and mining; scattered experimental data is produced in chemistry, physics, and engineering; and non-uniformly spaced computational values arise in the output from finite element solutions of partial differential equations, and various applications in computer graphics and computer vision. These fields require scattered data interpolation to determine values at arbitrary positions, not just those at which the data is available. This facilitates many useful operations for visualizing multidimensional data. For instance, in medical imaging, scattered data interpolation is essential to construct a closed surface from CT or MRI images of human organs. In geological applications, the derived interpolation function facilitates a contour map to be plotted.

The SDI problem can be defined in two dimensions as follows. Suppose we have a set

of n two-dimensional arbitrary distributed points

$$P_i = (x_i, y_i), \quad i = 1, 2, \cdots, n$$

over $\mathcal{R}^2$, and scalar values F_i associated with each point satisfying $F_i = F(x_i, y_i)$ for some underlying function $F(x, y)$. The problem is to find an interpolating function $\overline{F} \approx F(x, y)$ such that $\overline{F}(x_i, y_i) = F_i$ for $i = 1, 2, \cdots, n$ [34].

We assume that all the points P_i, referred to as mesh points are distinct. The formulation can be generalized into higher dimensions in a straightforward way. For the two-dimensional case, the scattered points P_i are on a plane, making the function $\overline{F}$ a bivariate function that defines a surface in three-dimensional space.

Given a particular use for the interpolating function $\overline{F}$, one may desire certain properties, such as smoothness, continuity or differentiability. For instance, smooth surfaces with C^1 continuity are sufficient for most of the applications that require visualization of surfaces. But if one is interested in visualizing a vector field, where the gradient of F is of relevance, higher order continuity might be preferred. We may also want the function to be expressed in explicit, implicit or procedural form.

Because there are particular attributes associated with each interpolating method, there is not a unique way to classify them. For example, the method could be global or local, based on the nodes needed to evaluate the function $\overline{F}$ at certain location (x, y). The form of the interpolating function could also be used for classification, since $\overline{F}$ could be a bivariate polynomial, a piecewise bivariate polynomial, or a rational function.

1.5 DEI: The Differential Equation Interpolant

In [12], Enright introduced the Differential Equation Interpolant (DEI) which approximates the solution of a PDE such that the approximations at off-mesh points have the same order of accuracy as those at mesh points provided by an underlying numerical PDE solver. The idea is to associate a multi-variate polynomial with each mesh element

and consequently, the collection of such polynomials over all mesh elements will define a piecewise polynomial approximation. In two dimensions, the DEI is a piecewise bivariate polynomial, $U(x, y)$, characterized by a number of unknown coefficients determined by the degree and type of the interpolant. The number of unknowns associated with the DEI is usually greater than the number of independent linear constraints defined by the information provided by the PDE solver at the discrete set of mesh points. While the standard approach in determining these additional constraints is based on enforcing continuity of higher derivatives of the piecewise polynomial at mesh points, with the DEI approach these additional constraints are based on introducing additional computation on each element to 'almost' satisfy the PDE at a prescribed set of local 'collocation' points.

Before reviewing details of how this is accomplished, for the particular DEIs we will consider, we note that constructing $U(x, y)$ from the data provided by the PDE method can be viewed as a post-processing task. The PDE method, in generating its discrete approximations has provided $O(h^p)$ accurate approximations to the solution $u(x, y)$ at the discrete mesh points. This information is then used (together with some additional mesh point approximations computed at some extra cost) to determine $U(x, y)$. How this is done is discussed in detail in [12]. For modest order methods ($p \leq 4$), such as those we investigated in this thesis, the cost required to generate this information is a small number of evaluations of the PDE (fewer than p sampled evaluations of the PDE for each element). Note that this post-processing approach to computing $U(x, y)$ is analogous to the determination of the CRK $S(x)$ in ODEs, when the cost of determining $S(x)$ for an underlying discrete Runge-Kutta formula requires computing additional sampled derivative values at each step (which can involve doubling the number of stages), and using these additional stages (together with the stages of the discrete Runge-Kutta formula) to determine $S(x)$.

The mesh elements can be rectangles or triangles depending on the distribution of the

associated mesh points. In two dimensions, for each element e, the bivariate polynomial $p_{d,e}(x, y)$ is represented by $(d+1)^2$ unknown coefficients,

$$p_{d,e}(x, y) = \sum_{i=0}^{d} \sum_{j=0}^{d} c_{ij} s^i t^j,$$

where

$$s = \frac{(x - x_1)}{D_1}, t = \frac{(y - y_1)}{D_2},$$

and D_1 and D_2 depend on the size of the mesh element e in the x and y direction, d is the degree and (x_1, y_1) is a mesh point associated with e (for rectangular elements, this is usually chosen to be the lower left corner of the rectangle).

In [12], the underlying PDE was assumed to be a two-dimensional, second-order problem of the form

$$Lu = g(x, y, u, u_x, u_y), \qquad (x, y) \in \Omega, \tag{1.1}$$

where L is a given semi-linear differential operator of the form

$$L = a_1(x, y)\frac{\partial^2}{\partial x^2} + a_2(x, y)\frac{\partial^2}{\partial y^2} + a_3(x, y)\frac{\partial^2}{\partial x \partial y}.$$

In the case that the underlying numerical method produces approximations on an unstructured mesh, the DEI can be considered to be a scattered data interpolant. In [34], Ramos and Enright presented an investigation of this problem. They introduced the Alternate Differential Equation Interpolant (ADEI), a piecewise polynomial interpolant associated with a scattered data set. Instead of choosing random points or fixed points, they developed an iterative algorithm to find suitable collocation points based on monitoring the magnitude of the coefficients of the resulting interpolant.

Although the ADEI provides a relatively smooth surface, it can still suffer from the appearance of discontinuities. Moreover, finding a suitable set of collocation points may require an excessive amount of computer time. In [22], we introduced the PCI, a pure cubic interpolant, to overcome these deficiencies. The PCI is globally continuous and

efficient in terms of time and error. Furthermore, in [23] we extended the definition of the PCI to a three-dimensional, trivariate interpolant and investigated the most efficient DEI-based interpolants in three dimensions.

1.6 PCI: The Pure Cubic Interpolant

The pure bivariate interpolant restricts the approximating polynomial to be a bivariate polynomial of total degree d. For each triangular element e, the interpolant, $p_{d,e}(x, y)$, is defined by

$$p_{d,e}(x, y) = \sum_{i=0}^{d} \sum_{j=0}^{d-i} c_{ij} s^i t^j, \tag{1.2}$$

where

$$s = \frac{(x - x_1)}{D_1}, \qquad t = \frac{(y - y_1)}{D_2}, \tag{1.3}$$

and, as can be seen in Figure 1.1, (x_1, y_1) is the lower left corner of the associated enclosing box of e and D_1 and D_2 are the dimensions of the box. For a pure interpolant of degree d, there are $\frac{(d+1)(d+2)}{2}$ unknown coefficients to be determined for each e.

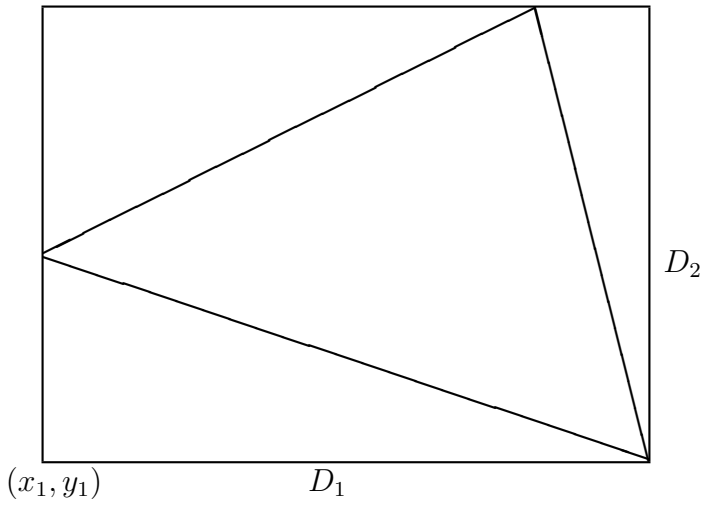


Figure 1.1: A triangular element and its associated enclosing box.

For the class of problems we are considering, fourth order accuracy seems to be standard and we have therefore chosen piecewise bi-cubics ($d = 3$) for our interpolant. In this case, we can represent the unknown coefficients by a matrix C of the form

$$C = \begin{pmatrix} 0 & 0 & 0 & c_{30} \\ 0 & 0 & c_{21} & c_{20} \\ 0 & c_{12} & c_{11} & c_{10} \\ c_{03} & c_{02} & c_{01} & c_{00} \end{pmatrix},$$

and therefore the cubic polynomial associated with e can be represented by,

$$p_{3,e}(x,y) = S^T C T = \begin{pmatrix} s^3 & s^2 & s & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & c_{30} \\ 0 & 0 & c_{21} & c_{20} \\ 0 & c_{12} & c_{11} & c_{10} \\ c_{03} & c_{02} & c_{01} & c_{00} \end{pmatrix} \begin{pmatrix} t^3 \\ t^2 \\ t \\ 1 \end{pmatrix}.$$

We call this interpolant the Pure Cubic Interpolant (PCI). For the PCI, we have 10 unknown coefficients. For each of the three mesh points of a triangle we assume we have 4^{th} order approximations to the function, u, and 3^{rd} order approximations to the derivatives, u_x and u_y. As we noted in the previous section, these approximations to u_x and u_y are not likely directly available from the PDE solver but are computed at a small additional cost as a post-processing task. To define the DEI, we require that the local interpolant (associated with each element) interpolates $\overline{u}$, $\overline{u_x}$, and $\overline{u_y}$ at the 3 mesh points associated with the element. This results in 9 linear equations, each of which is in one of the following forms:

$$\sum_{i=0}^{3} \sum_{j=0}^{3-i} c_{ij} \overline{s}^i \overline{t}^j = \overline{u}, \tag{1.4}$$

$$\sum_{i=0}^{3} \sum_{j=0}^{3-i} i c_{ij} \overline{s}^{i-1} \overline{t}^j = \overline{u_x}, \tag{1.5}$$

$$\sum_{i=0}^{3} \sum_{j=0}^{3-i} j c_{ij} \overline{s}^i \overline{t}^{j-1} = \overline{u_y}. \tag{1.6}$$

We still require another independent linear constraint to uniquely determine the 10 coefficients of $p_{3,e}(x,y)$. This can be accomplished by choosing one collocation point,

(x_m, y_m), inside the triangle e and imposing a condition that the interpolant almost satisfies the underlying PDE at that point. From (1.2) we have

$$\frac{\partial^2 p_{3,e}(x,y)}{\partial x^2} = \frac{1}{D_1^2}\sum_{i=0}^{3}\sum_{j=0}^{3-i} i(i-1)c_{ij}\Big(\frac{x-x_0}{D_1}\Big)^{i-2}\Big(\frac{y-y_0}{D_2}\Big)^{j}, \tag{1.7}$$

$$\frac{\partial^2 p_{3,e}(x,y)}{\partial y^2} = \frac{1}{D_2^2}\sum_{i=0}^{3}\sum_{j=0}^{3-i} j(j-1)c_{ij}\Big(\frac{x-x_0}{D_1}\Big)^{i}\Big(\frac{y-y_0}{D_2}\Big)^{j-2}, \tag{1.8}$$

$$\frac{\partial^2 p_{3,e}(x,y)}{\partial x \partial y} = \frac{1}{D_1 D_2}\sum_{i=0}^{3}\sum_{j=0}^{3-i} ijc_{ij}\Big(\frac{x-x_0}{D_1}\Big)^{i-1}\Big(\frac{y-y_0}{D_2}\Big)^{j-1}, \tag{1.9}$$

and therefore $Lp_{3,e}(x_m, y_m)$ is a linear combination of the unknown coefficients. From (1.1) for the point (x_m, y_m) we impose the additional linear constraint

$$Lp_{3,e}(x_m, y_m) = g(x_m, y_m, \hat{U}_m, \frac{\partial \hat{U}_m}{\partial x}, \frac{\partial \hat{U}_m}{\partial y}), \tag{1.10}$$

where $\hat{U}_m$ is any low order ($p = 2$) approximation to $U(x_m, y_m)$ and $\frac{\partial \hat{U}_m}{\partial x}$ and $\frac{\partial \hat{U}_m}{\partial y}$ are derivatives of $U(x_m, y_m)$ at the point (x_m, y_m) in the x and y directions, respectively (see [12] for details and justification of why this works). In the case when the order of $U(x_m, y_m)$ is 3, as it is for the PCI, $\hat{U}_m$ can be a linear approximation to U (and we will use this choice). Since we need only one collocation point, the obvious choice is the middle point of the triangle and we have investigated this choice. To determine the unknown coefficients, we then solve the system of linear equations that is formed by the 9 linear constraints (1.4), (1.5) and (1.6) and the additional linear constraint (1.10). This system can be written as

$$W\overline{c} = b \tag{1.11}$$

where W is a matrix that depends on the mesh points, the collocation point, and the definition of L; while $\overline{c} \in \mathcal{R}^{10}$ is the vector of unknown coefficients; and $b \in \mathcal{R}^{10}$ is a vector that depends on the mesh data and the linear approximations associated with the collocation point.

One of the advantages of the PCI is its continuity along the boundaries of mesh elements. The standard DEI finds an accurate interpolation over each element but might

suffer from discontinuity along the boundaries of the mesh elements. This will not be the case for the PCI, which always produces continuous results. This follows since the associated pure cubic polynomials for two neighbor triangles produce the same value for each arbitrary point on the shared side. For each of the two mesh points on the shared side, there are three pieces of information (U, U_x and U_y). If the side is a horizontal or vertical line, either U_x or U_y expresses the derivative of the function along that side. In general, where the side is an arbitrary line, we are able to state the derivative of the function in the direction of the side, U_μ, in terms of U_x and U_y by

$$U_\mu = \frac{\partial U}{\partial x} \cdot x_\mu + \frac{\partial U}{\partial y} \cdot y_\mu. \tag{1.12}$$

Therefore, for each mesh point we know the value of the function and the value of the derivative in the direction of each of the two adjacent sides. All this information is used to define the polynomial. Since along each side, the PCI is of degree 3 in μ, it can be characterized by four unknown coefficients. For the shared side we know exactly four pieces of information, the associated polynomials for both triangles produce the same value for each point located on the shared side. In this way, we have satisfied continuity, one of the most important aspects with regard to visualization.

In addition to continuity, the PCI is efficient in terms of time and accuracy. Since the linear system that must be solved has only 10 linear independent equations, the PCI needs less time than the DEI of degree 3, which has 16 linear independent equations, and much less time than the ADEI that may have to iterate many times to choose the best location for the collocation points.

It is worth noting that the PCI can also be applied to a high-order PDE solver and it can be effective to use a low-order interpolant for a high-order PDE solver. As an example, in COLSYS for ODE problems, the order of accuracy at mesh points is $O(h^{2k})$ compared with $O(h^{k+1})$ at off-mesh points.

1.7 Other Two Dimensional Approaches

Although the PCI generally produces excellent results in terms of time, accuracy and continuity, we are not sure if it is the best interpolant relative to the other DEIs for our problem. In order to obtain such confidence, we will investigate some other approaches based on using a DEI. We call these approaches DEIm-n where m is the degree of the corresponding interpolant and n is the number of the unknown coefficients. The first two alternative DEIs, DEI3-15 and DEI3-13, are based on the DEI of degree 3 but total degree 5 and 4 respectively (the standard DEI is basically of total degree 6) and the last one, DEI4-15, is a pure quartic interpolant of total degree 4. C_1, C_2 and C_3 are the matrices of unknown coefficients for DEI3-15, DEI3-13 and DEI4-15 respectively and are defined by

$$C_1 = \begin{pmatrix} 0 & c_{32} & c_{31} & c_{30} \\ c_{23} & c_{22} & c_{21} & c_{20} \\ c_{13} & c_{12} & c_{11} & c_{10} \\ c_{03} & c_{02} & c_{01} & c_{00} \end{pmatrix},$$

$$C_2 = \begin{pmatrix} 0 & 0 & c_{31} & c_{30} \\ 0 & c_{22} & c_{21} & c_{20} \\ c_{13} & c_{12} & c_{11} & c_{10} \\ c_{03} & c_{02} & c_{01} & c_{00} \end{pmatrix},$$

$$C_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & c_{40} \\ 0 & 0 & 0 & c_{31} & c_{30} \\ 0 & 0 & c_{22} & c_{21} & c_{20} \\ 0 & c_{13} & c_{12} & c_{11} & c_{10} \\ c_{04} & c_{03} & c_{02} & c_{01} & c_{00} \end{pmatrix}.$$

The results of the comparison between the PCI approach and these alternative DEI approaches will be presented in chapter 5. As we will observe, the PCI is the best approach in both time and accuracy. It is also the only approach that always generates

continuous results. We will use the interpolant generated by this approach for implementing our fast two-dimensional contouring algorithms.

1.8 Three Dimensional SDI

In [23], we focused on scattered data interpolation associated with the numerical solution of a three-dimensional second-order elliptic PDE of the form

$$Lu = g(x, y, z, u, u_x, u_y, u_z),$$

where L is a given differential operator of the form

$$L = a_1(x, y, z)\frac{\partial^2}{\partial x^2} + a_2(x, y, z)\frac{\partial^2}{\partial y^2} + a_3(x, y, z)\frac{\partial^2}{\partial z^2}.$$

We assume that there are some accurate numerical results (approximate solution values, $u(x, y, z)$, as well as approximate derivative values, $u_x(x, y, z)$, $u_y(x, y, z)$ and $u_z(x, y, z)$) at some mesh points that are scattered or unstructured in 3D. The mesh points partition the domain of the problem into a collection of mesh elements which are tetrahedra. Our approach is to associate with each mesh element e, a tri-variate polynomial $p_{d,e}(x, y, z)$ of degree d, which approximates $u(x, y, z)$ on mesh element e. In other words, one can determine a polynomial $p_{d,e}(x, y, z)$ that interpolates the data values associated with the mesh points of e and 'almost' satisfies the PDE at a predetermined set of collocation points of e. The number of collocation points depends on the degree d and type of interpolant (tensor product or pure). The collection of such polynomials over all mesh elements will then define a piecewise polynomial approximation $p_d(x, y, z)$, that is well defined for all (x, y, z) in the domain of interest.

Enright [12] has investigated some aspects of the performance of this approach for three dimensional problems. He reported that, for tetrahedron meshes, pure tri-cubic interpolants and tensor product tri-quadratic are the most appropriate candidates.

A pure tri-cubic polynomial for a mesh element e is defined by

$$p_{3,e}(x,y,z) = \sum_{i=0}^{3} \sum_{j=0}^{3-i} \sum_{k=0}^{3-i-j} c_{ijk} s^i t^j v^k,$$

where

$$s = \frac{(x-x_1)}{D_1}, t = \frac{(y-y_1)}{D_2}, v = \frac{(z-z_1)}{D_3},$$

and (x_1, y_1, z_1) is the corner of the associated enclosing box of e with the smallest values of (x, y, z); and D_1, D_2 and D_3 are the dimensions of the box.

The number of unknown coefficients, c_{ijk}, for a pure three-dimensional interpolant of degree d is $\frac{(d+1)(d+2)(d+3)}{6}$. Thus for a pure tri-cubic (where $d = 3$), there are 20 unknown coefficients. Since we already have 16 data values associated with e (u, u_x, u_y and u_z for each of the four nodes of the tetrahedron), we need to add at least 4 collocation points to uniquely determine the interpolant associated with e.

Alternatively a tensor product tri-quadratic polynomial can be defined by

$$p_{2,e}(x,y,z) = \sum_{i=0}^{2} \sum_{j=0}^{2} \sum_{k=0}^{2} c_{ijk} s^i t^j v^k.$$

For a tensor product three-dimensional interpolant of degree d, there are $(d+1)^3$ unknown coefficients. Therefore for a tensor product tri-quadratic, we have 27 unknowns to identify. Therefore at least 11 collocation points for each mesh element will be required to determine $p_{2,e}$.

Note that a pure tri-quadratic polynomial would have total degree 2 and have only 10 unknowns. Since the number of unknowns is less than the number of linear equations provided by the information at the four mesh points of e, it is not appropriate to investigate this type of interpolant. However we can consider a tri-quadratic polynomial of total degree 3 as follows:

$$\hat{p}_{2,e}(x,y,z) = \sum_{i=0}^{2} \sum_{j=0}^{\min(2,3-i)} \sum_{k=0}^{\min(2,3-i-j)} c_{ijk} s^i t^j v^k.$$

Since $\hat{p}_{2,e}$ has only 17 unknown coefficients, it requires less time to compute than $p_{2,e}$, and our testing has demonstrated that it also seems to generate more accurate results than $p_{2,e}$ in practice.

In section 5.2, we compare these three candidate interpolants and investigate their performance on a set of three-dimensional elliptic PDEs over an unstructured mesh. As we will observe, a pure tri-cubic interpolant, $p_{3,e}$, generates more accurate results than the tri-quadratic interpolants. This interpolant also is the best in terms of realistic, non distracting, visualization. However, none of these interpolants are globally continuous along the boundaries of the mesh elements as they provide continuity on the shared edges, but not necessarily on the shared faces.

The extension of DEI-based interpolants to more than three dimensions is theoretically possible. However, the algebra becomes more difficult and it requires more effort to avoid ill-conditioned linear systems, especially when determining higher order interpolants.

Chapter 2

Contouring Algorithms

2.1 Introduction

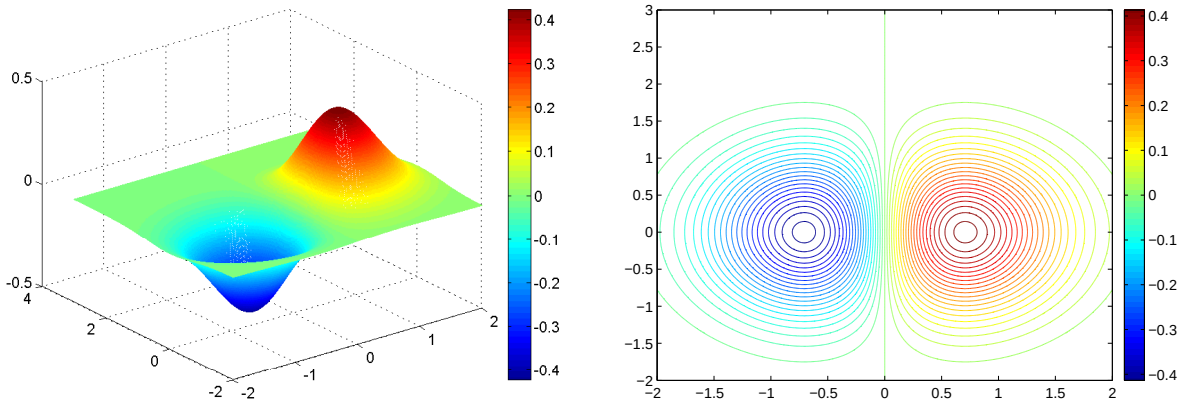
Most Partial Differential Equations (PDEs) that arise in practical applications do not have a closed form solution. In these cases, numerical methods can be used to approximate the solution at a discrete set of mesh points in the domain associated with the problem definition. After approximating the solution at a discrete set of mesh points by a numerical PDE solver, one might plot contour lines in order to visualize the solution. Standard contouring algorithms require knowing the function to be contoured on a regularly-spaced rectangular mesh. In cases where the original mesh is unstructured, a regular rectangular mesh must be introduced first. In addition, most standard contouring algorithms use linear interpolation inside each element. With these algorithms the more mesh points we have, the more accurate contour plot we obtain. To obtain a smooth contour plot, the numerical method must provide the approximations at a very fine mesh. In the case that we have an interpolant such as a DEI, we can obtain the refined mesh data directly from the DEI. Bradbury and Enright [7] investigated the application of the DEI to visualize the solution of PDEs when the underlying mesh is rectangular and structured. In addition, they introduced three fast algorithms to compute contour lines

efficiently directly from the DEI. In this chapter, we will introduce fast algorithms to compute contour lines directly for a particular DEI, the PCI [21]. In addition, we will extend the algorithm to three dimensions by introducing an algorithm to draw contour surfaces in three dimensions.

For a function of two variables, a contour curve is a curve along which the function has a constant value. In general, a level set of a real-valued function f of n variables is a set of the form

$$\{(x_1, x_2, \cdots, x_n) | f(x_1, x_2, \cdots, x_n) = c\}$$

where c is a constant. In other words, it is a set where the function takes on a given constant value. Figure 2.1 shows a surface plot and several different contour plots for the function $z = xe^{-(x^2+y^2)}$. Each of the color lines in Figure 2.1 corresponds to a different contour curve, $f(x, y) = c$ (for different values of c).



(a) The surface plot

(b) The contour plots

Figure 2.1: An example of a surface plot and contour plots in two dimensions.

Contour curves are of interest in a number of fields and problems such as:

- **Meteorology:** A curve connects points on a map that have the same temperature.

- **Cartography:** A curve joins points of equal elevation (height) above a given level.
- **Geography:** Contour curves denote elevation or altitude and depth on maps. From these contours, a sense of the general terrain can be determined.
- **Engineering:** Various types of graphs can be presented using contour curves. Such graphs are useful for visualizing or representing more than two dimensions on two-dimensional displays.

In particular, in computational science, one way to visualize the approximate solution obtained by a numerical PDE solver is to draw contour curves of the associated approximate solution, as an alternative to computing a standard surface plot (see Figure 2.1).

In the next section, some related previous investigations are presented.

2.2 Previous Work

In order to have a relatively smooth contour plot, a fine mesh approximation of the solution is needed. One way to obtain such a fine mesh approximation is to solve the problem by the numerical PDE solver for the requested accuracy on an associated coarse mesh and then generate a fine mesh approximation from the coarse mesh approximation using a DEI. Bradbury and Enright [7] investigated this approach (Matlab/DEI approach) to generate a fine mesh approximation and then plotted contour lines using a standard contouring algorithm like the built-in Matlab *contour* procedure. In addition, they introduced three fast direct contouring algorithms based on the use of the DEI that avoid the introduction of a fine mesh. To illustrate the fast contouring algorithms, they considered an underlying parabolic PDE of the form

$$Lu = g(x, y, u, u_x), \tag{2.1}$$

$$L = \frac{\partial}{\partial y} - \beta(x, y)\frac{\partial^2}{\partial x^2} \tag{2.2}$$

where the DEI is a piecewise bicubic polynomial and the underlying mesh is rectangular. Their results show that the fast contouring algorithms will be better than the Matlab/DEI approach if the refining factor (the ratio of the fine mesh to the coarse mesh) is relatively large. In section 2.3, we improve their fast contouring algorithms and extend them to unstructured triangular meshes. We will extend the algorithm to three dimensions in section 2.4.

2.3 Two Dimensional Contouring Algorithms

The user can specify either the contour level(s) explicitly or only the desired number of contour levels. In the latter situation, the contour levels are specified by equally dividing the range between global minimum and maximum values (This is the interface provided in the Matlab *contour* procedure). In this section, we will introduce three fast contouring algorithms for an unstructured triangular mesh that provide a similar interface. The results we will discuss, originally appeared in [20] and have been subsequently reported in [21].

Each algorithm has three stages and shares a common first two stages. These three stages consist of:

1. Computing the minimum and maximum values of the interpolant for each triangle. There are at least three advantages in computing the minimum and maximum values. First of all, it simplifies subsequent intersection tests for all triangles where we determine whether the triangle contains a segment of the contour line. Secondly, the situation that a contour curve lies completely inside a triangle can be easily distinguished (as there will be no intersection between the contour curve and the triangle's sides). Thirdly, global minimum and maximum values can easily be computed from this local information for each triangle and, if the contour levels are not specified by the user, suitable default values can be determined dynamically

based on this global information.

2. Identifying the intersection points between contour lines and the sides of the triangle and recursively dividing the triangles containing more or less than two such intersections into several triangles such that all triangles have two or zero intersections. A situation where the sides of a triangle have a total of exactly two intersections with a contour line is called a *desired* situation. Note that the heuristics we employ at this stage to accomplish this task do not directly deal with all possible situations. Only the most likely scenarios are addressed directly with the understanding that the less likely (or more pathological cases) may result in skipping a section of the contour curve if an 'undesirable' triangle is still present after six recursive refinement steps.

3. Computing a smooth and accurate contour line connecting the two intersection points for each triangle. We will introduce three fast and reliable algorithms to accomplish this task.

In the following, we will present each of these three stages in more detail.

2.3.1 Stage One: Computing the Minimum and Maximum Values

Finding the minimum and maximum values of the interpolant over each triangle can be the most expensive stage if we try to identify them to full machine accuracy [7]. The most accurate method is to find the points inside triangle e such that $(p_e)_x$ and $(p_e)_y$ are simultaneously zero but this can be very expensive in terms of computer time. A faster method is based on the assumption that the extreme values associated with the sides of a triangle are acceptable approximations to the extreme values associated with the whole triangle. This method is relatively fast (it involves only finding zeros of three polynomials for each triangle) and more reliable than the Matlab contouring procedure

that assumes the extreme values occur at the mesh points (or vertices of the triangle). We can also improve this technique to obtain better results by a simple idea. The idea is a recursive approach where the user can specify the number of recursive steps. Each recursive step consists of forming a new (smaller) triangle by joining the extreme points associated with each side and then determine the extreme points of this smaller triangle. This recursive algorithm is presented in Figure 2.2, and Figure 2.3 illustrates how the algorithm finds the approximate minimum (or maximum) values of the interpolant for a triangle through a two-level recursion. The cost of this algorithm is a multiple ($\leq 2*$ number of recursive steps) of the cost of the primary algorithm (locating the extreme values on a side of a triangle). Note that, in some cases, this algorithm might only find the extreme values along a triangle's sides and fail to find a more extreme value inside the triangle. However, when this happens, the only consequence is that a closed segment of the contour curve which lies entirely inside the triangle and which is associated with a contour level close to an extreme value may be skipped.

2.3.2 Stage Two: Identifying Intersection Points and Dividing Triangles

After computing an approximate minimum and maximum values for the interpolant over each triangle, we can determine whether a given contour line intersects a triangle by simply comparing the contour level with the minimum and maximum values. Then, for each triangle e, if the contour level is between the minimum and maximum values over e, an intersection test is performed for each of three sides of triangle e. The result of this test can then be used to classify the triangle in terms of total number of intersections and the most likely situations are:

- *No intersection*: the contour line lies completely inside the triangle.

```
for each triangle e of the mesh
  lMin(e) ← recFindMin(e,1)
  lMax(e) ← recFindMax(e,1)
end for
gMin ← min(lMin)
gMax ← max(lMax)

function recFindMin(e,level)
  recFlag ← False
  for each of three sides of triangle e (i=1, 2, 3)
    find the extreme of p_e(x, y) along the ith side
    minPoint(i) ← select the minimum point from the extreme points and two end points
    if minPoint(i) is selected from the extreme points
      recFlag ← True
    end if
  end for
  if level < MAX_LEVEL and recFlag=True
    newTriangle ← The triangle formed by connecting minPoint(i) (i=1, 2, 3)
    minValue ← recFindMin(newTriangle,level+1)
  else
    minValue ← min(value of p_e(x, y) at minPoint)
  end if
  return minValue
end function

function recFindMax(e,level)
⋮
```

Figure 2.2: The recursive algorithm to approximate the extreme values in 2D.

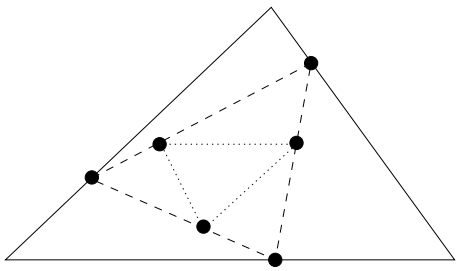


Figure 2.3: Finding the minimum (or maximum) values of the interpolant.

- *One intersection*: the contour line is either an inner or outer tangent to one side at the intersection point.
- *Two intersections*: a single contour line passes through the triangle connecting these two intersection points.
- *Three intersections*: a single contour line passes through the triangle and it is also tangent to one side.
- *Four intersections*: two contour lines pass through the triangle.
- *More than four intersections*: more than two contour lines pass through the triangle.

Our fast algorithms are based on drawing a contour line between two intersection points (stage three). Therefore, the 'two intersections' case is our desired situation and for all cases except *'two intersections'*, the triangle should be divided into two or more triangles such that each new triangle has exactly two intersections with the contour level (or zero if a new triangle contains no segment of the contour line). We implement this approach using a recursive function that takes a triangle and the contour level and other necessary parameters and then computes the contour line. Note that our classification scheme ignores pathological (and unlikely cases) such as two intersection points of a triangle corresponding to two tangent points (associated with different segments of the contour curve). Moreover, at most six recursive steps are applied and after that, any triangle which is not in a desired situation will be skipped and its contribution to the

overall contour ignored. This could happen in the regions in which the triangulation is too coarse to accurately resolve a curvy contour. For each of the above six cases, excluding the *'two intersections'* case, an appropriate strategy or heuristic is adapted which attempts to replace the undesirable triangle with a set of desirable triangles. The respective strategy we adopt is:

1. **For no intersection:** A segment of the contour curve will lie completely inside a triangle. The strategy is to find a line between a vertex of the triangle and its corresponding opposite side such that it intersects the contour curve at at least two points. This is easy to do since the location of the approximate maximum and minimum values of the interpolant (Max, Min respectively) on this triangle are known and the contour level v satisfies $Min \leq v \leq Max$. In this case, as there are no intersection with any side, we must have that the values of the interpolant at each vertex must all be less than v or they must all be greater than v. Figure 2.4 shows the situation where the value at each vertex is less than the contour value v and the triangle is divided into two triangles by the line drawn through one of the vertices and the location of Max. A similar division is made when the values at each vertex are all greater than v (only the dividing line passes through Min). We will then have, in all but some rare or pathological cases, two triangles in a desired situation (each has exactly two intersections with the contour curve) and we will determine the contour segments for each new triangle, separately. Note that, in the unlikely event that more than two intersections are detected, we recursively apply the appropriate stage 2 strategy to each of the new triangles.

2. **For one intersection:** Figure 2.5 illustrates the situations that can result for one intersection corresponding to an inner tangent or an outer tangent. There are two different treatments for these two situations. In order to distinguish between

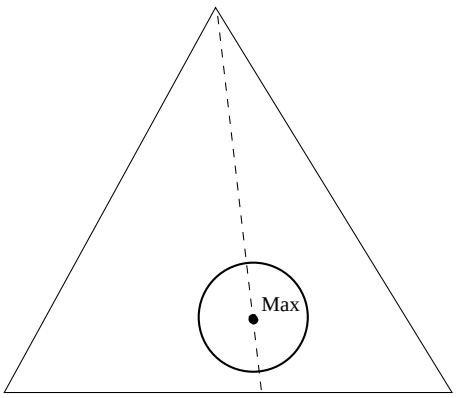
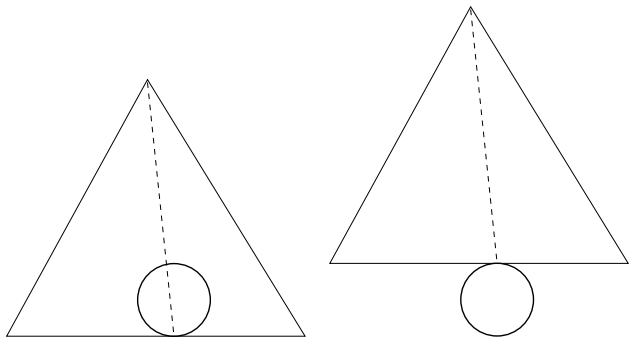


Figure 2.4: No intersection: Two new triangles.

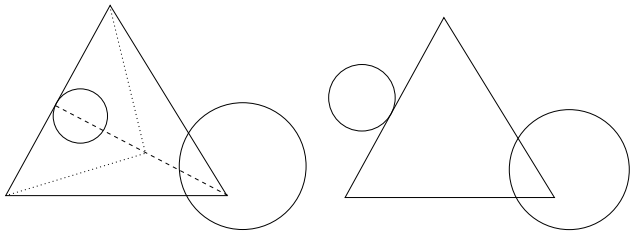
the inner tangent and outer tangent situations, we consider the line connecting the intersection point and the corresponding opposite vertex. Then, we compute the number of intersections between this line and the contour curve. In case that there is only one intersection (outer tangent), we will do nothing as the contour segment will be considered by the neighbor triangle. In the other case, when there are two or more intersection points (inner tangent), we divide the triangle into two new triangles (see Figure 2.5(a)) and then invoke the recursive function for each of them separately. In most cases, each new triangle will have exactly two intersections with the contour curve and consequently we have a desired situation for each of the new triangles. In a rare case, the contour can be curvy enough to create triangles with more than two intersections and, again, the recursive function of the appropriate stage 2 strategy will handle this situation.

3. **For three intersections:** The situation with three intersections can only arise if one of the intersections is a tangent and this case can be treated like the situation with one intersection if the tangent is inner (2.6(a)) and like the situation with two intersections if the tangent is outer (2.6(b)). In the former case (inner tangent), by connecting the tangent point to the corresponding opposite vertex, we will have two



(a) Inner tangent (b) Outer tangent

Figure 2.5: One intersection: Two situations, inner tangent and outer tangent.



(a) Inner tangent (b) Outer tangent

Figure 2.6: Three intersections: Two situations, inner tangent and outer tangent.

new triangles such that each one has two or more intersections with the contour curve. In the case that there are more than two intersections, we will call the recursive function again. In the latter case (outer tangent), the tangent point can simply be ignored and we have a desired situation. In order to determine which intersection point is the tangent point, we find the tangent line to the contour curve at all three points and then choose the point whose tangent is coincident with the corresponding side of the triangle. It is also easy to determine if the tangent is inner or outer by finding intersections between the contour curve and a line parallel to the tangent side but slightly shifted to be 'inside' the triangle. If there is no intersection, the tangent is exterior; otherwise it is interior. Note that since the situation with three intersections happens very rarely, it is not necessary to develop a very efficient strategy.

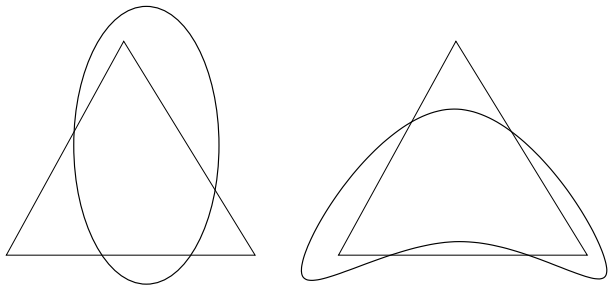


Figure 2.7: Four intersections: 2-1-1 situations.

4. **For four intersections:** There are several situations that can arise when there are four intersections with the contour curve. We classify these situations into four sub-categories according to the position of the intersection points as follows:

   (a) Two intersections on one side and one intersection on each of the other sides (2-1-1 situation): Figure 2.7 shows two such situations. We apply the following strategy to compute contour lines in these situations:

   > *Find the middle point of two intersections that are on the same side. Divide the triangle into two triangles by adding a line between this point and the corresponding opposite vertex and then call the recursive function for each of the two new triangles.*

   Figure 2.8 shows the final triangles created by applying this strategy. It works for the first situation where it results in two triangles that are in a desired situation. For the second situation, it creates two triangles, each of which has four intersections with the contour curve. It can be seen from the figure that both of these new triangles are of the first type and the contour lines will be computed by applying the strategy one more time.

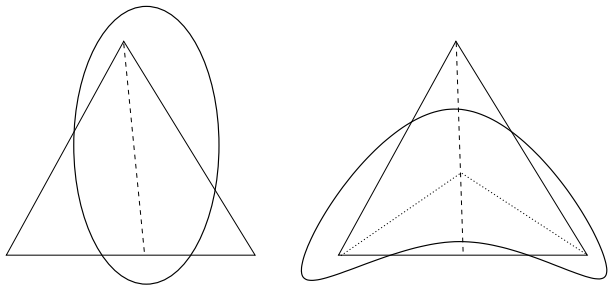


Figure 2.8: Four intersections: 2-1-1 situations (Final triangulation).

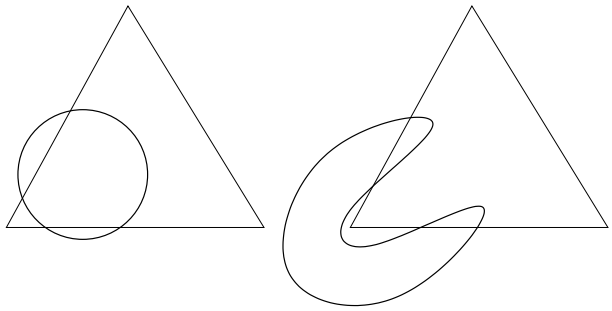


Figure 2.9: Four intersections: 2-2-0 situations.

(b) Two intersections on two sides and no intersection on the other side (2-2-0 situation): Figure 2.9 illustrates two such situations. We apply the following strategy to compute contour lines in these situations:

> *Find the middle points of the intersections that are on the same side. Draw a line between these two points and also draw a line between one of the middle points and the corresponding opposite vertex and then call the recursive function for each of three new triangles separately.*

In the first situation (of Figure 2.9), each of the three triangles has exactly two intersections with the contour curve. In the second situation, one triangle has two intersections with the contour curve and the other triangles have four

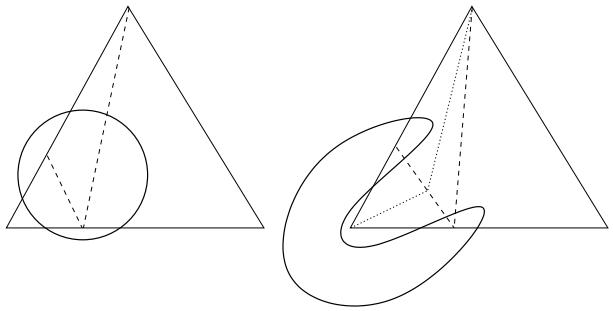


Figure 2.10: Four intersections: 2-2-0 situations (Final triangulation).

intersections with contour curve (2-1-1 situation) and we apply the recursive function one more time. Figure 2.10 shows the final triangles created by applying this strategy recursively.

(c) Three intersections on one side, one intersection on a second side and no intersection on the last side (3-1-0 situations): Figure 2.11 shows two such situations. We apply the following strategy to compute contour lines in these situations:

Find the middle point of two adjacent intersections that are on the same side. Draw a line between the middle point and the corresponding opposite vertex and then call the recursive function for each of two triangles separately.

Although this strategy does not directly convert the first case of Figure 2.11 into a desired situation, it converts a 3-1-0 situation into 2-1-1 and 2-2-0 situation, both of which have been discussed before. This strategy converts the first situation into two new triangles whose category depends on the position of two chosen adjacent intersections. Figure 2.12 shows the triangles that can be created by the first situation (of Figure 2.11).

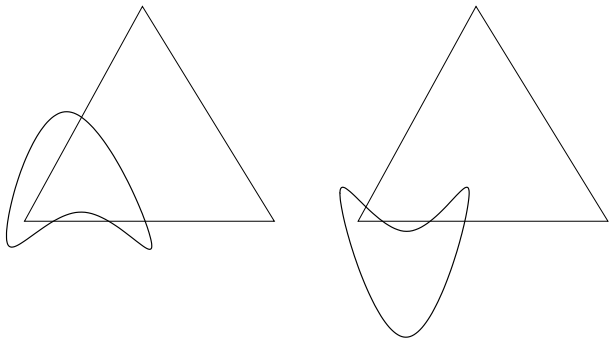


Figure 2.11: Four intersections: 3-1-0 situations.

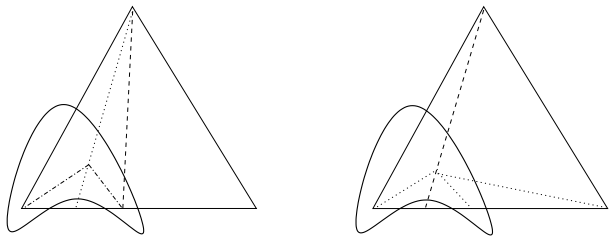


Figure 2.12: Four intersections: the first 3-1-0 situation (Final triangulation).

In the first case of Figure 2.12, the triangle is divided into two new triangles, one with two intersections and the other with four intersections of type 2-1-1 that can be converted to a desired situation by calling the recursive function two more times (as discussed before). Therefore, the first case is converted into a desired situation in at most three recursive steps. For the second case of Figure 2.12, the triangle is divided into two new triangles, both with four intersections with the contour curve, one of type 2-1-1 and the other of type 2-2-0. Since both new triangles can be obviously converted into a desired situation in one step, the second case can be converted into a desired situation in two recursive steps.

Figure 2.13 illustrates the two new cases that can be created by the second situation (of Figure 2.11). In the first case, the triangle is divided into two

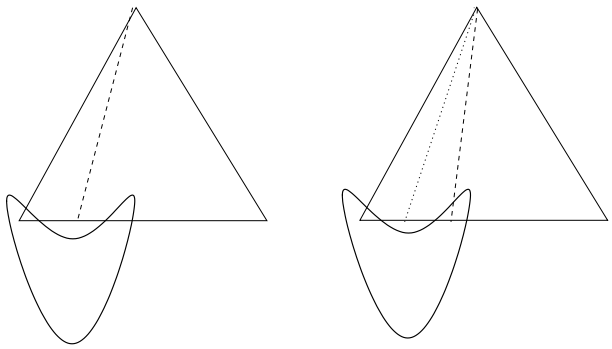


Figure 2.13: Four intersections: the second 3-1-0 situation (Final triangulation).

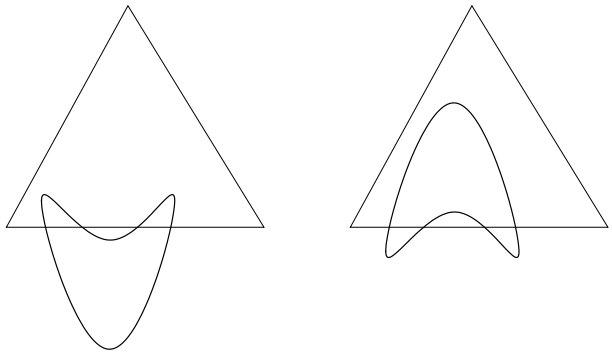


Figure 2.14: Four intersections: 4-0-0 situations.

new triangles, both in a desired situation. In the second case, the triangle is divided into two new triangles, one with four intersections of type 2-1-1 and the other with two intersections. Therefore, this second situation is also converted into a desired situation in at most two recursive steps.

(d) Four intersections on one side and no intersection on the other two sides (4-0-0 situations): Figure 2.14 shows the two such situations that can arise. We apply the following strategy to compute contour lines in these situations:

Find the middle point of the two middle intersection points. Draw a line between the middle point and the corresponding opposite vertex

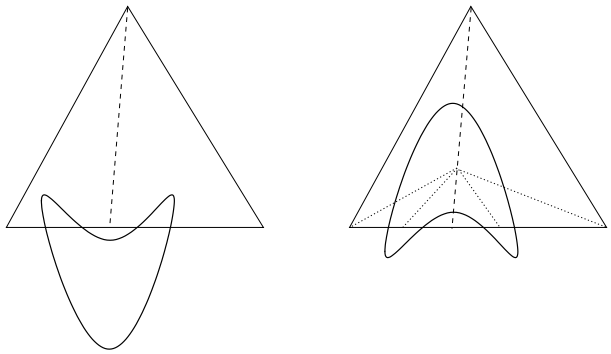


Figure 2.15: Four intersections: 4-0-0 situations (Final triangulation).

and then call the recursive function for each of the two new triangles.

In the first situation (of Figure 2.14), each of the two triangles has exactly two intersections with the contour curve. In the second situation, each of the two triangles has four intersections of type 2-2-0 and we apply the recursive step one more time. Figure 2.15 shows the final triangles created by applying this strategy recursively.

5. **For more than four intersections:** In the case that the triangulation is relatively coarse, we might encounter some triangles that have more than four intersections with the contour curve. We adopt a simple strategy in order to divide such triangles into a set of triangles each of which is of the form already considered (especially those triangles with two and four intersections).

   Quadrisect the triangle by connecting the midpoints of the sides.

   Figure 2.16 illustrate an example of such a situation. It shows a triangle with six intersections with the contour curve, and the triangles after the initial step. In most cases, our strategy converts the triangle into four new triangles such that three of them have only two intersections (desired situation) and one has no intersection.

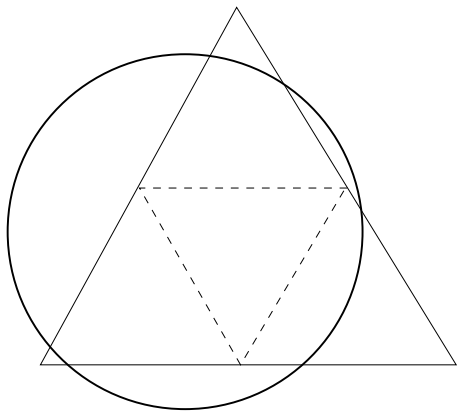


Figure 2.16: More than four intersections.

In case of a more complicated contour, the strategy might create triangles with more than two intersections which will be handled by subsequent iterations of the recursive step.

This classification seems to be appropriate for all the cases that can arise in contours. It is important to note that the recursive algorithm stops after at most six steps. Therefore, some rare cases might cause slight errors in the contour plot by omitting segments of contour curves.

2.3.3 Stage Three: Computing the Accurate Contour Lines

The last stage of our fast contouring algorithms is to compute the contour line between two points located on a triangle's sides. In the following, at first we introduce three techniques based on those implemented in Bradbury and Enright [7] for rectangular grids and then we try to improve each technique separately.

The Intercept Method

The basic idea of the Intercept method is to refine each element in only one direction, the x or y direction, depending on the location of the intersection points. We then find

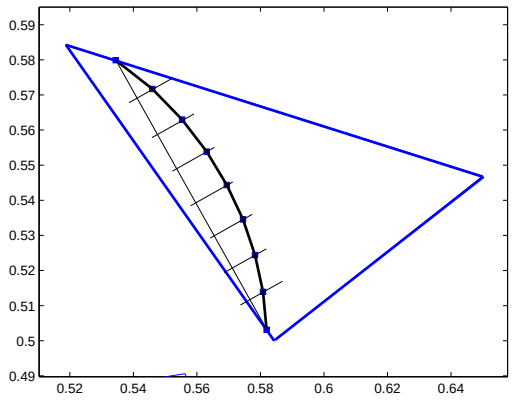


Figure 2.17: The contour curve created by the basic Intercept method. The points on the contour curve are the intersections between the contour curve and refinement lines.

intersections between new lines in the refined direction and the contour line and finally simply connect the intersection points [7]. Since each refined line is horizontal or vertical (i.e. parallel to the x or y axis), $p_e(x, y)$ reduces to a univariate cubic polynomial (along the refined line). For a triangular mesh, since we employ a PCI, the following strategy can be applied

> *Connect the two intersection points by a line and then consider some regularly-spaced points on this line. Draw perpendicular lines to this line at the considered points and then find the intersections between these perpendicular lines and the contour curve and connect the intersection points.*

Figure 2.17 illustrates how this strategy approximates the contour line with a refinement of eight.

The interpolant used for this problem, $p_e(x, y)$, is the Pure Cubic Interpolant (PCI). If we let $s = \frac{x-x_m}{D_1}$, $t = \frac{y-y_m}{D_2}$ where (x_m, y_m) is a vertex of e and D_1, D_2 are the dimensions of a rectangle that enclose e, then

$$p_e(x,y) = SCT = \begin{pmatrix} s^3 & s^2 & s & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & c_{30} \\ 0 & 0 & c_{21} & c_{20} \\ 0 & c_{12} & c_{11} & c_{10} \\ c_{03} & c_{02} & c_{01} & c_{00} \end{pmatrix} \begin{pmatrix} t^3 \\ t^2 \\ t \\ 1 \end{pmatrix}.$$

Although the refined lines are not necessary horizontal or vertical (i.e., parallel to the x or y axis), the polynomial restricted to each refined line will be a cubic univariate since the PCI is of total degree three. For each arbitrary line we have $s = at + b$ and if we replace s with $at + b$, we will have

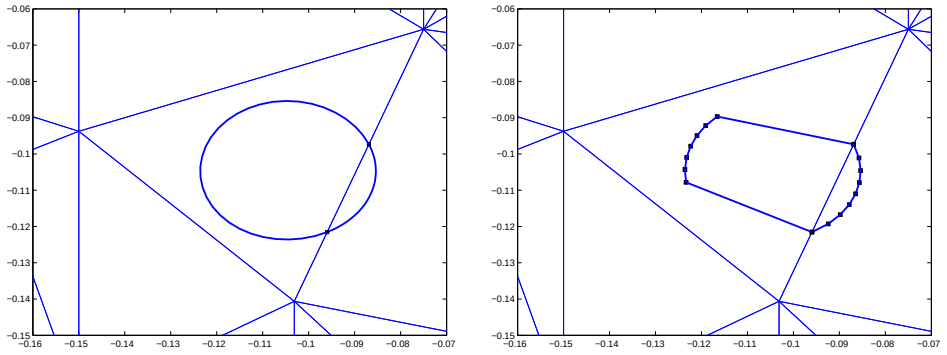
$$p_e(x,y) = \begin{pmatrix} a^3 & 3a^2b & a^2 & 3ab^2 & 2ab & a & b^3 & b^2 & b & 1 \end{pmatrix} \begin{pmatrix} c_{30} & 0 & 0 & 0 \\ 0 & c_{30} & 0 & 0 \\ c_{21} & c_{20} & 0 & 0 \\ 0 & 0 & c_{30} & 0 \\ 0 & c_{21} & c_{20} & 0 \\ c_{12} & c_{11} & c_{10} & 0 \\ 0 & 0 & 0 & c_{30} \\ 0 & 0 & c_{21} & c_{20} \\ 0 & c_{12} & c_{11} & c_{10} \\ c_{03} & c_{02} & c_{01} & c_{00} \end{pmatrix} \begin{pmatrix} t^3 \\ t^2 \\ t \\ 1 \end{pmatrix}, \tag{2.3}$$

and $p_e(x,y)$ will reduce to a cubic univariate polynomial in t. To determine the intersections between the refined line and the contour curve with contour level v, we find the roots of

$$k_3t^3 + k_2t^2 + k_1t + k_0 = v, \tag{2.4}$$

where the coefficients k_0, k_1, k_2, and k_3 are determined from (2.3).

This strategy works well as long as there is only one intersection between each refined line and the contour curve. Bradbury and Enright addressed this difficulty and showed



(a) True contour curve.

(b) The contour curve plotted by the basic Intercept method.

Figure 2.18: An illustration of the difficulty that can arise with the basic Intercept method.

a simple example where the whole contour curve lies inside one element [7]. We will not encounter this situation because in our algorithm there will be exactly two intersections between the contour curve and the sides of the triangle containing a segment of the contour. We might still have the situation where there is more than one intersection between a refined line and the contour curve. Note that the PCI can find at most three intersections. Figure 2.18(a) shows a simple situation and Figure 2.18(b) shows the result of our basic approach. As can be seen, we lose some sections of the contour curve due to the fact that the refinement is performed only between two intersections. On the other hand, refining the area outside of two intersections does not solve this difficulty because there is more than one intersection and it is not obvious how the intersection points should be connected. In order to resolve this difficulty, we introduce a recursive approach that attempts to find some equally spaced points located along the contour curve.

The idea is to first find the middle point of the contour curve between the initial two interception points and consider this point as one of the refined points (known to be on the contour curve) and then recursively treat each of the two new intervals (the intervals between the new point and each of the initial points). Figure 2.19 presents a

function recAddMiddlePoint(p_1,p_2,cnt,$\cdots$)

$p_m \longleftarrow = \frac{p_1+p_2}{2}$

$m_1 \leftarrow$ the angle of the tangent line of the contour curve at p_1

$m_2 \leftarrow$ the angle of the tangent line of the contour curve at p_2

$m \leftarrow \frac{m_1+m_2}{2}$

$p \leftarrow$ the intersection between the contour curve and the line passing p_m with an angle m

if cnt > 0

$p_L \leftarrow$ recAddMiddlePoint(p_1,p,cnt-1,$\cdots$)

$p_R \leftarrow$ recAddMiddlePoint(p,p_2,cnt-1,$\cdots$)

return $p_L + p + p_R$

else

return p

end if

end function

Figure 2.19: The recursive algorithm to find middle points.

more detailed description of this approach. If n is the number of recursive levels, we will have $2^n + 1$ points (including the two initial points) all lying on the contour curve.

Figure 2.20 illustrates how this approach draws contour curves by applying a three-level recursion. An important advantage of this approach is that it locates almost equally spaced points on the contour curve. However, there is still a remaining difficulty. As can be seen from Figure 2.20, although the points are located equally spaced inside each triangle, they are not totally equally spaced if we consider all triangles. In other words, the number of extra points should not necessarily be equal for all triangles. For example in Figure 2.20, the number of extra points (7 points) is not enough to give a suitable representation for the left triangle but more than enough for the right triangle. In order

to overcome this deficiency, instead of having a fixed level of recursion for all triangles, we consider a threshold for the distance between the extra points and try to locate the extra points such that the distance between each two points is less than the desired threshold. Figure 2.21 illustrates the effect of considering a threshold instead of the same level of recursion for all triangles. The value of the threshold can be specified by the user.

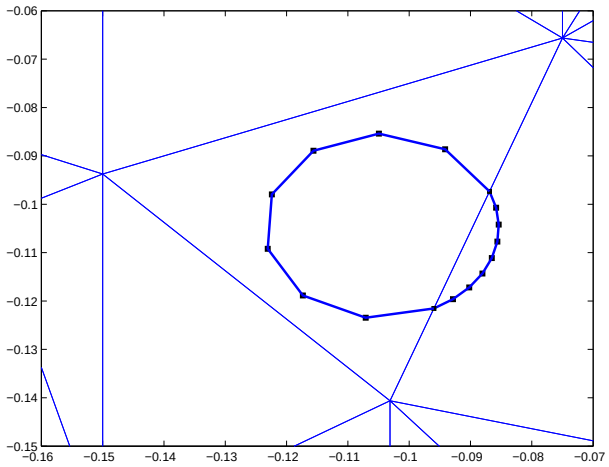


Figure 2.20: The contour curve created by the improved Intercept method using a fixed number of recursions, in this case $n = 3$, to reduce the chance of missing a segment of the contour curve.

The Simple ODE Method (SODE)

The contouring problem can be characterized as an initial value problem [7]. The problem for the contour level v is to solve

$$u(x, y(x)) = v, \tag{2.5}$$

where y is a function of x. If we differentiate both sides of this equation with respect to x, we obtain

$$u_x(x, y(x)) + u_y(x, y(x)) \cdot y_x(x) = 0, \tag{2.6}$$

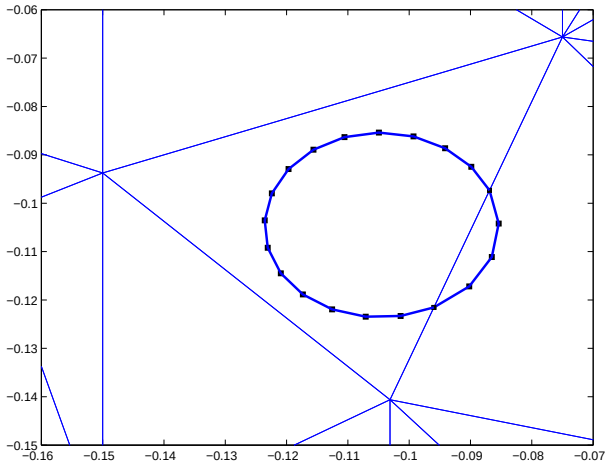


Figure 2.21: The contour curve created by the final Intercept method using a threshold value to control the spacing.

or

$$\frac{dy}{dx} = y_x(x) = -\frac{u_x(x, y(x))}{u_y(x, y(x))}. \tag{2.7}$$

This ODE is satisfied for $(x, y(x))$ lying on the contour curve. If we consider x as a function of y, we can similarly derive

$$\frac{dx}{dy} = x_y(y) = -\frac{u_y(x(y), y)}{u_x(x(y), y)}. \tag{2.8}$$

We can approximate $u(x, y)$,$u_x(x, y)$ and $u_y(x, y)$ at prescribed values of x and y using the PCI (and its partial derivatives). We can then solve equation (2.7) or (2.8) by applying a numerical IVP solver starting from a known intersection point. Furthermore, rather than applying an IVP solver, we can compute approximations to $y(x)$ and $y_x(x)$ (or $x(y)$ and $x_y(y)$) at prescribed values directly from the PCI. We can then interpolate the contour curve by using an appropriate order Hermite interpolant. The minimum number of extra points (in addition to the intersection points) necessary to define the Hermite interpolant is dependent on the contour curve and some properties of the two intersection points. Bradbury and Enright considered one middle point for all situations [7]. However, for some situations that can arise, we need no middle point and the contour curve can be approximated accurately with only the two intersection points.

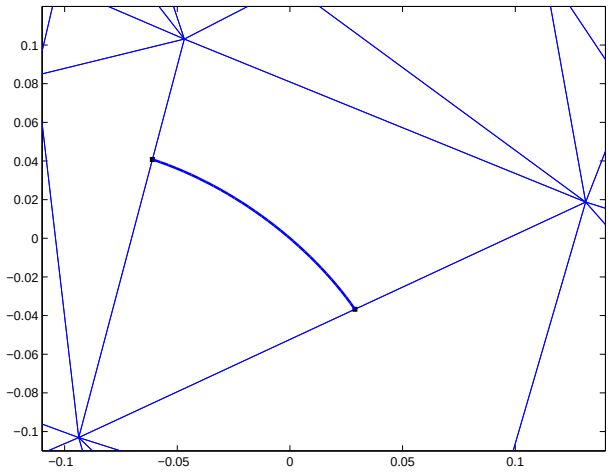


Figure 2.22: The common situation in contour curves.

In the case that the gradient of u at the two intersection points has the same sign (for both x and y directions), we need no middle points and the contour curve can be computed precisely by applying a cubic Hermite interpolant. Figure 2.22 shows such a situation. For the other cases, we will use the approach that will be introduced in the next method, the ODE with arclength. When the magnitude of the respective derivative, (y_x or x_y), is much larger than one, the Hermite interpolant may not approximate the solution of this IVP very well. We apply the ODE with arclength method for such situations.

The ODE with Arclength Method (ODEA)

In the SODE method, we assumed that y is a function of x (or x is a function of y). An alternative method is to parameterize with respect to arclength and assume both x and y to be functions of arclength [7]. In this case, we will have

$$u(x(s), y(s)) = v, \tag{2.9}$$

where v is the contour level and s is the arclength. Then if we differentiate (2.9) with respect to s, we will obtain

$$u_x \cdot x_s + u_y \cdot y_s = 0, \tag{2.10}$$

and if we consider a normalization condition for the arclength, we will get

$$x_s{}^2 + y_s{}^2 = 1. \tag{2.11}$$

From equations (2.10) and (2.11) we will obtain a system of ODEs,

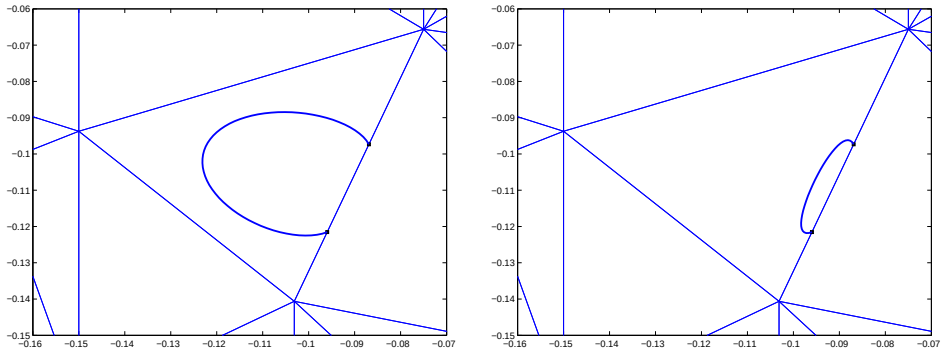
$$x_s = \frac{\pm u_y}{\sqrt{u_x{}^2 + u_y{}^2}}, \tag{2.12}$$

$$y_s = \frac{\mp u_x}{\sqrt{u_x{}^2 + u_y{}^2}}, \tag{2.13}$$

which is satisfied on the contour curve. Note that the sign of $x_s(\pm)$ differs from the sign of $y_s(\mp)$. In other words, if x_s and u_y have the same sign, y_s and u_x should have different sign and vice versa. In order to identify the proper sign in equations (2.12) and (2.13), an intersection test can be done along a single line of refinement. Like the SODE approach, we can obtain approximations of $x(s)$, $y(s)$, $x_s(s)$ and $y_s(s)$ at each arbitrary point directly from the PCI using Hermite interpolation to approximate $x(s)$ and $y(s)$ separately.

In most cases, the data ($x(s)$, $x_s(s)$, $y(s)$ and $y_s(s)$) from two intersection points is adequate to accurately approximate the contour curve. However in some cases, the Hermite interpolants fail to determine a suitable approximation and at least one extra point is required. Figure 2.23(a) illustrates one of these situations and Figure 2.23(b) shows the result of the Hermite interpolant if we apply it with only the two intersection points. By analyzing the difficulty and observing the result of some special situations, we realized that if either $x(s)$ or $y(s)$ has more than one extreme point on the contour curve, the Hermite interpolant may fail to accurately interpolate the contour curve. Note that since the PCI is of degree 3 in x and y, the corresponding polynomial can have at most two extreme points in each direction x or y (corresponding to one minimum and one maximum). We can assume that the middle point of the contour curve locates these two extreme points in two different sections.

In order to find the middle point of the contour curve, a similar approach to that employed in the SODE method can be used. After identifying the middle point we



(a) True contour curve.

(b) The contour curve plotted by the ODEA method with no middle point.

Figure 2.23: A situation where the ODEA method fails if no middle point is considered.

can use two cubic Hermite interpolants for computing the contour curve. Figure 2.24 illustrates the result of the situation corresponding to Figure 2.23 when we consider the middle point.

One advantage of the ODEA method is its relationship to Hermite interpolation of the exact contour curve. As mentioned in the previous section, the Hermite interpolant can fail to approximate the solution of the associated IVP (2.7) or (2.8) accurately if the derivative is much larger than one in magnitude. However, for the ODEA method, due to the normalization condition, it is guaranteed that x_s and y_s are both bounded by one in magnitude.

Since the ODEA method needs fewer middle points than the Intercept method, it computes contours much faster than the Intercept method. However, its results may not be as precise as those of the Intercept method since it is approximating the contours of the underlying Hermite interpolant of the contour curve. In order to obtain more accurate results with the ODEA method, we can add some extra 'middle points' and then apply a piecewise cubic Hermite defined over a finer mesh to compute the contour curve. We call this method, the Alternate ODE with Arclength method (AODEA). Like with the

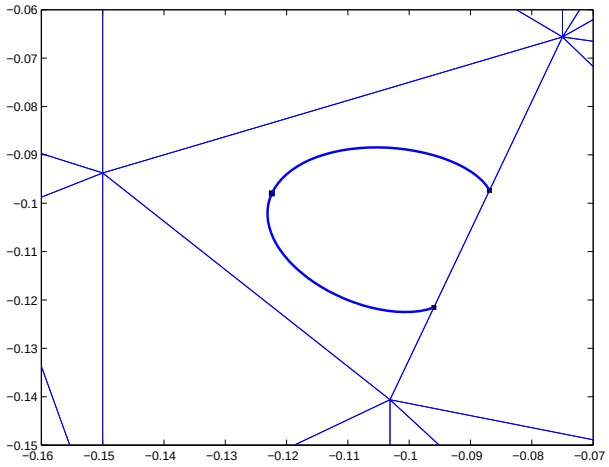


Figure 2.24: The contour curve plotted by the ODEA method with one middle point.

Intercept method, the user can specify a threshold value δ as the minimum distance separating neighbor points. For the Intercept method the contribution to the error in the contour segment will be $O(\delta^2)$ while, for the AODEA method the contribution to this error will be $O(\delta^4)$. The appropriate threshold value for use with AODEA can therefore be larger than that for the Intercept method.

2.4 Three Dimensional Contouring Algorithms

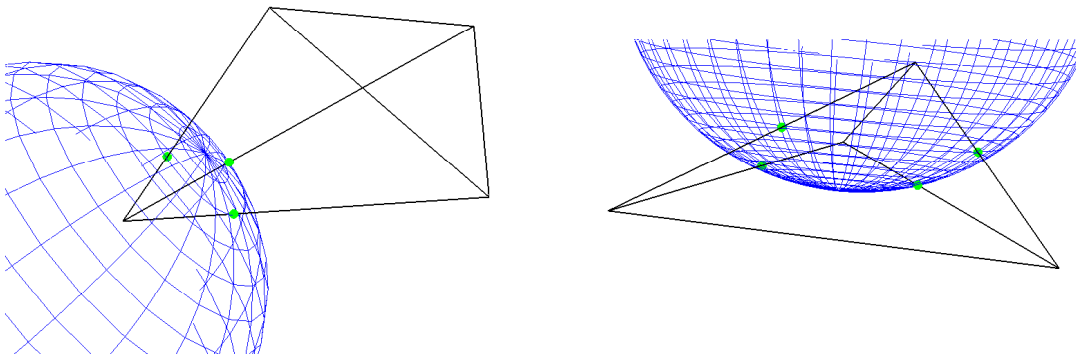
Three dimensional or volume visualization is associated with the graphical representation of data sets that are defined on three-dimensional grids. The algorithms presented in section 2.3 can be directly extended to plot three dimensional contour surfaces. Similar to the two dimensional case, the user can specify either the contour level(s) explicitly or only the desired number of contour levels. We assume that we are given an unstructured tetrahedral mesh associated with the approximate solution of a three dimensional PDE. For each tetrahedral element, e, we have a PCI, $p_e(x, y, z)$. The 3D algorithm has three stages as follows:

1. Computing approximate minimum and maximum values of the local interpolant for each tetrahedron. This stage helps in identifying the tetrahedra that include

some component of the contour surface. Moreover, the situation that a component of the contour surface lies completely inside a tetrahedron can be identified during this stage.

2. Identifying the intersection points between the contour surface and each of the six edges of a tetrahedron. A situation where the tetrahedron has 'three intersections on three different edges' or 'four intersections on four different edges' is called a *desired* situation (Figure 2.25). After determining the intersection points, a tetrahedron which is not in one of the desired situations will be recursively divided into several tetrahedra such that all tetrahedra are finally in a desired situation.

3. Computing a smooth and accurate contour surface using the intersection points identified in stage two for each tetrahedron.



(a) Three intersections with three different edges (b) Four intersections with four different edges

Figure 2.25: Desired situations in three dimensions. The tetrahedron e is displayed in black; the unknown contour surface is displayed (as a wire-frame surface) in blue; and intersection points with the edges of e are displayed as green dots.

2.4.1 Stage One: Computing the Minimum and Maximum Values

Similar to the two dimensional case, the problem of computing the strict minimum and maximum values of the interpolant over e would involve finding the points inside e such that $p_{e_x}(x, y, z)$, $p_{e_y}(x, y, z)$ and $p_{e_z}(x, y, z)$ are simultaneously zero. However, this would be very expensive in terms of required computer time. As with the 2D case, we will be satisfied with the computation of suitable approximations to these values for each e. We first observe that the extreme values that occur on the faces of tetrahedra can be found relatively easily. In order to approximate the extreme values on each face, one can apply the algorithm presented in Figure 2.2 for two dimensions to each of the four triangular faces. The resulting approximations to the maximum and minimum values on each face $((\overline{Max}_i, \overline{Min}_i), i = 1, 2, 3, 4)$ allow us to determine if there is an intersection of the contour surface with any of the faces. A simple improvement is to consider the extreme values on each face as four nodes of a virtual tetrahedron and then apply the idea recursively on this new tetrahedron. In the case that there are less than four distinct nodes (for example when two faces have a shared extreme point), the goal would change to find the extreme value on a 2D triangle (or even 1D line segment) in 3D space. This recursive algorithm is presented in Figure 2.26.

2.4.2 Stage Two: Identifying Intersection Points and Dividing Tetrahedra

In order to determine whether a contour surface at a given contour level has any intersection with a tetrahedron, one can simply compare the contour value with the minimum and maximum values of the interpolant over the selected element. For each tetrahedron e, if the contour level is between the minimum and maximum values of the interpolant over e, an intersection test is performed for each of six edges of e. The result of this test

for each tetrahedron e of the 3D mesh

lMin(e) ← recFindMin3D(e,1)

lMax(e) ← recFindMax3D(e,1)

end for

gMin ← min(lMin)

gMax ← max(lMax)

function recFindMin3D(e,level)

for each of four faces of tetrahedron e (i=1, 2, 3, 4)

minPoint(i) ← Find the point with the minimum value of $p_e(x, y, z)$ along the i^{th} face

end for

if level < MAX_LEVEL

newTetrahedron ← The tetrahedron formed by connecting minPoint(i) (i=1, 2, 3, 4)

minValue ← recFindMin(newTetrahedron,level+1)

else

minValue ← min(value of $p_e(x, y, z)$ at minPoint)

end if

return minValue

end function

function recFindMax3D(e,level)

⋮

Figure 2.26: The recursive approach to approximate the extreme values in 3D.

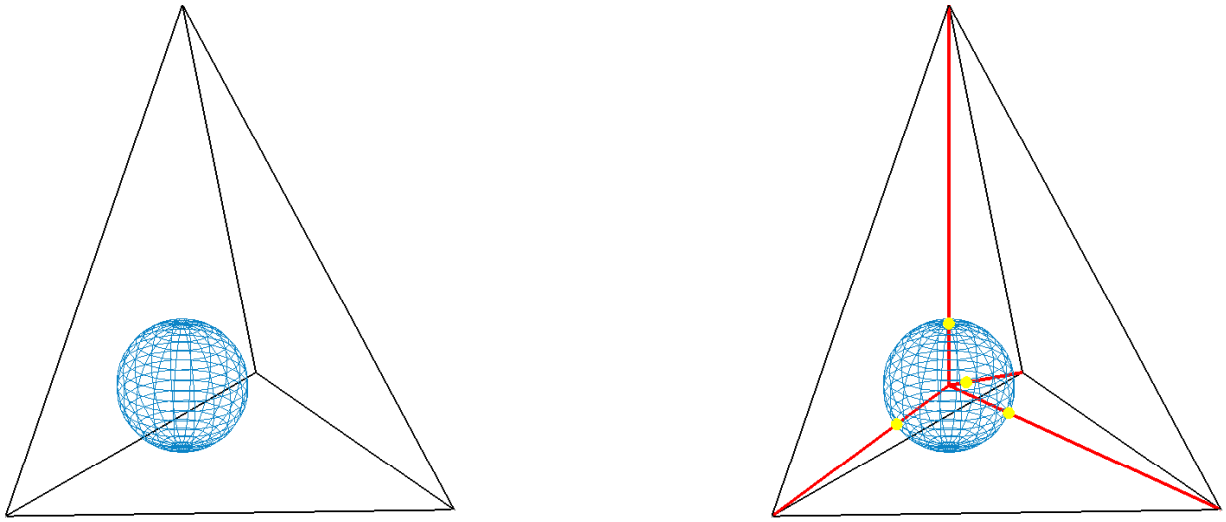
can then be used to classify those tetrahedra (that do contain a segment of the contour surface) in terms of the number and location of their edge-intersections. The possible situations are:

- *No intersection*: a component of the contour surface lies inside the tetrahedron.
- *Two or more intersections, two of them with a single edge*: the contour surface passes through a single edge of the tetrahedron in two locations and might have more intersections with other edges.
- *Three intersections*: the contour surface passes through three different edges of the tetrahedron.
- *Four intersections*: the contour surface passes through four different edges of the tetrahedron.

The 'three intersections' and 'four intersections' cases are our desired situations and an example of each is displayed in Figure 2.25. The green dots are the intersections between the contour surface and the edges of the tetrahedron. For other situations, the tetrahedron should be divided into two or more tetrahedra such that each new tetrahedron is in one of the two desired situations. A recursive approach could be applied if after one step the tetrahedron is still not in a desired situation. Based on our test results, the recursive approach seems to terminate after at most three or four recursive steps. For each of the two non-desirable situations, an appropriate strategy is adopted as follows:

1. **No intersection:** A component of the contour surface lies inside the tetrahedron. This could happen, for example, when the contour has a closed shape and the mesh is coarse. The strategy is to find an associated extreme point inside e and divide the tetrahedron into four new tetrahedra by connecting the extreme point to the four nodes of the original tetrahedron. The approximate extreme point to choose is the location of the local approximate maximum or minimum associated with e

(which must be an interior point of e as there are no edge-intersections with the contour surface). In the most likely situation, this will create four new tetrahedra such that each one would have exactly three intersections with the contour surface on three different edges, which is one of the desired situations. Figure 2.27 shows the 'no intersection' situation and the final tetrahedra after applying the strategy. The yellow dots show the intersections of the contour surface with the edges of four new tetrahedra.

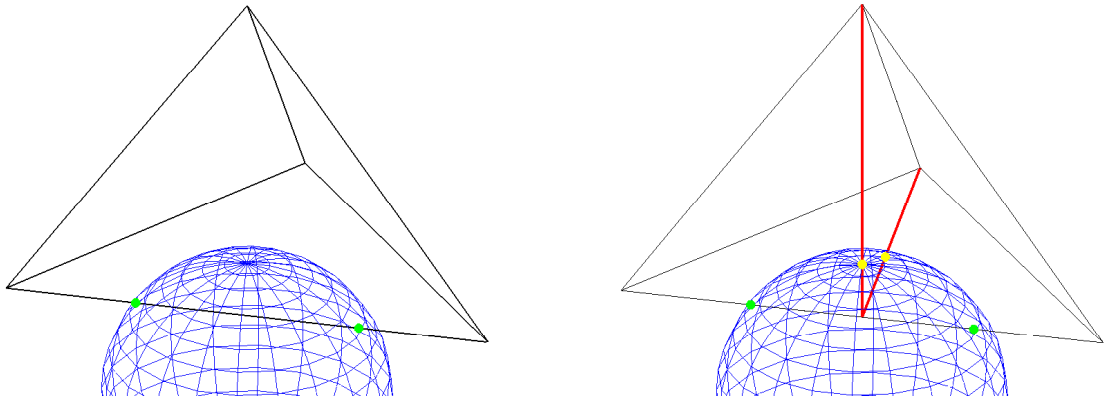


(a) Before applying the strategy (b) After applying the strategy (four new tetrahedra)

Figure 2.27: 'No intersection' situation in 3D.

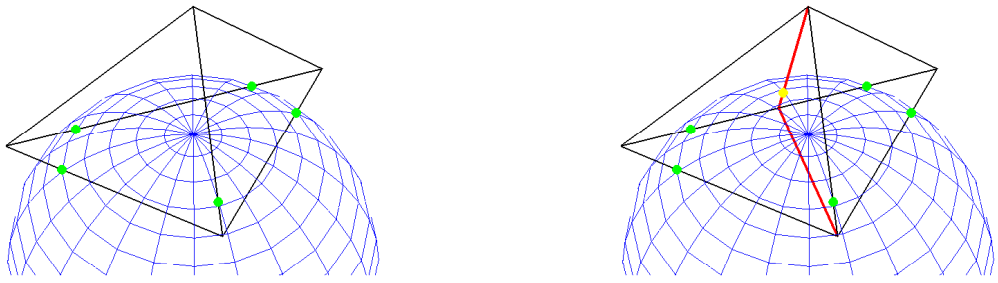
2. **Two or more intersections, two of them with a single edge:** This is the most likely situation when it is not in a desired situation. If there is one edge that has exactly two intersections with the contour surface, we apply a simple strategy, regardless of other intersections. In the case that there are other edges with two intersections, the algorithm will take care of them in a subsequent recursive step. The strategy is to find the middle point between the two intersections and then to divide the tetrahedron into two new tetrahedra by connecting the middle

point to the two opposite vertices of the original tetrahedron (i.e. the vertices not adjacent to the edge with two intersections). Figure 2.28 shows an example of this situation where there are only two intersections, both with the same edge. As can be seen after applying the strategy, we will obtain two new tetrahedra such that each has exactly three intersections with the contour surface on three different edges. Another situation which falls into this category is where there are five intersections with four edges (two of them with a single edge). Figure 2.29 shows how the strategy works with only one recursive step. Each of two new tetrahedra has exactly four intersections with the contour surface on four different edges which is again one of the desired situations. A more complex situation happens when there are four intersections with only two edges, two intersections with each. In this situation, by applying the same strategy for one edge, we will obtain two tetrahedra, one with three intersections (a desired situation) and the other one with five intersections with four edges, two of them with a single edge (An example of this situation is displayed in Figure 2.30(b)). In the next recursive step, the strategy will take care of the other edge with two intersections. Figure 2.30 illustrates this situation and the final tetrahedra after two recursive steps.



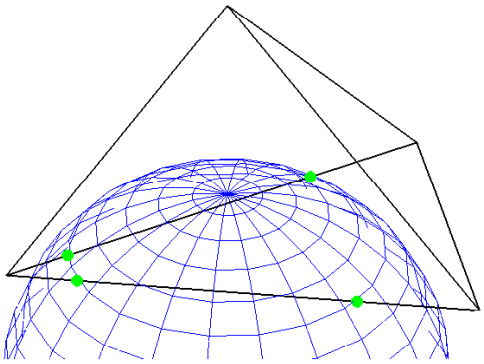
(a) Before applying the strategy (b) After applying the strategy

Figure 2.28: 'Two intersections with a single edge' situation in 3D.

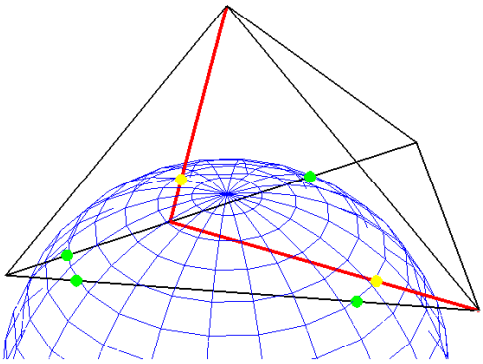


(a) Before applying the strategy (b) After applying the strategy

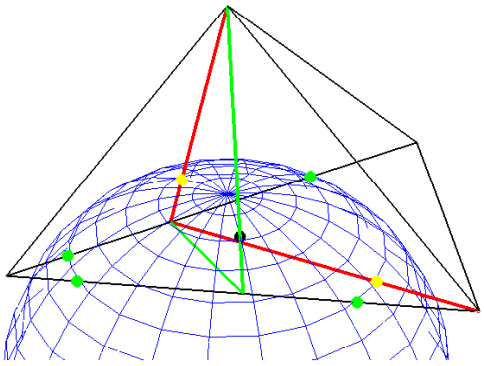
Figure 2.29: 'Five intersections with four edges' situation in 3D.



(a) Before applying the strategy



(b) After first recursive step



(c) After second recursive step

Figure 2.30: 'Four intersections with two edges' situation in 3D.

2.4.3 Stage Three: Computing the Accurate Contour Surfaces

The last stage of our contouring algorithm is to display components of the contour surface for each of the two desired situations. Although each of the three techniques discussed in section 2.3.3 for two dimensions is extendable into three dimensions, we have implemented only one of them in 3D, the *Intercept* method.

In two dimensions, the basic idea of the *Intercept* method was to simply connect intersections of the contour curve and the perpendicular lines from some regularly-spaced points on the straight line connecting two intersection points (see Figure 2.17). In this section, we will extend this basic idea into three dimensions for each of the two desired situations.

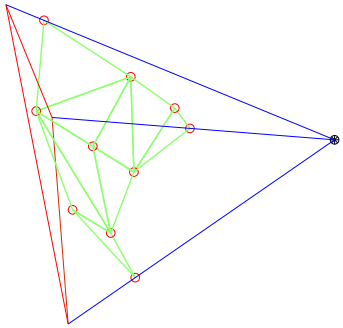
Three intersections with three different edges

As illustrated in Figure 2.25(a), these three intersection points lie on three different edges sharing a vertex, called the *apex*. We also call the triangle formed by connecting the other three vertices of the tetrahedron, the *base*. The following strategy can be applied for this situation

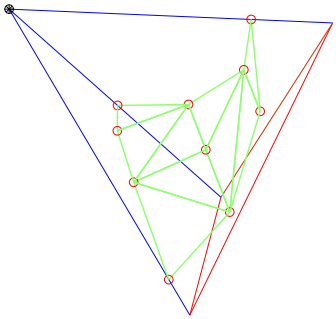
> *Consider some regularly-spaced collection of points covering the base triangle. Draw lines by connecting each of these regularly-spaced points to the apex and then find the intersections of these lines with the contour surface. A refined contour surface will be formed by connecting all the intersection points in an obvious way.*

Figure 2.31 shows how the strategy works to plot wire frame contour surfaces using 4 and 5 points on each side of the *base* (resulting in a total of 10 and 15 regularly-spaced points covering the *base*). We call this number *nSide*. Note that by filling or coloring the area of the refined triangles, one can obtain a rendered contour surface instead of a wire frame plot. As can be seen in the figure, the more regularly-spaced points we

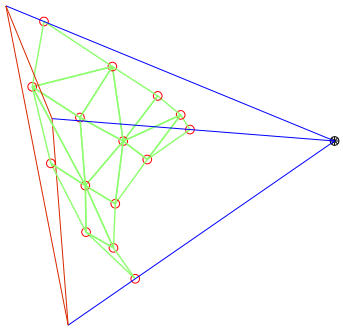
introduce in our *base* triangle, the more accurate contour we obtain and the more time required to plot the contour. On the other hand, in order to get a smooth contour plot, the number of extra points for a tetrahedron could be determined based on the amount of the contour surface which lies inside the tetrahedron.



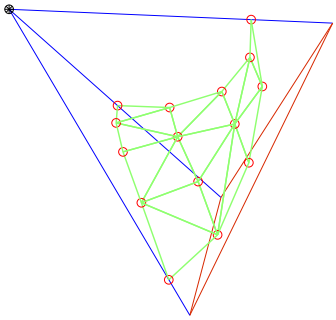
(a) $nSide = 4$ (total of 10 points)



(b) $nSide = 4$ (another view)



(c) $nSide = 5$ (total of 15 points)



(d) $nSide = 5$ (another view)

Figure 2.31: Two different views of drawing contour surface for 'three intersections with three different edges' situation in 3D. The black dot is the *apex* and red triangle is the *base*.

The interpolant used for this problem is $p_{3,e}(x, y, z)$, the pure tri-cubic interpolant introduced in Chapter 1, which is of the form

$$p_{3,e}(x, y, z) = \sum_{i=0}^{3} \sum_{j=0}^{3-i} \sum_{k=0}^{3-i-j} c_{ijk} s^i t^j v^k,$$

where

$$s = \frac{(x - x_m)}{D_1}, t = \frac{(y - y_m)}{D_2}, v = \frac{(z - z_m)}{D_3},$$

and (x_m, y_m, z_m) is the corner of the associated enclosing box of e with the smallest values of (x, y, z); and D_1, D_2 and D_3 are the dimensions of the box.

The parametric equation of a three dimensional line passing through (x_1, y_1, z_1) and (x_2, y_2, z_2) is of the form

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = \frac{z - z_1}{z_2 - z_1} = m$$

or

$$\begin{aligned} x &= x_1 + (x_2 - x_1)m \\ y &= y_1 + (y_2 - y_1)m \\ z &= z_1 + (z_2 - z_1)m \end{aligned} \tag{2.14}$$

In order to find the intersection of a line with the contour surface with a contour level v, we should solve

$$p_{3,e}(x, y, z) = v. \tag{2.15}$$

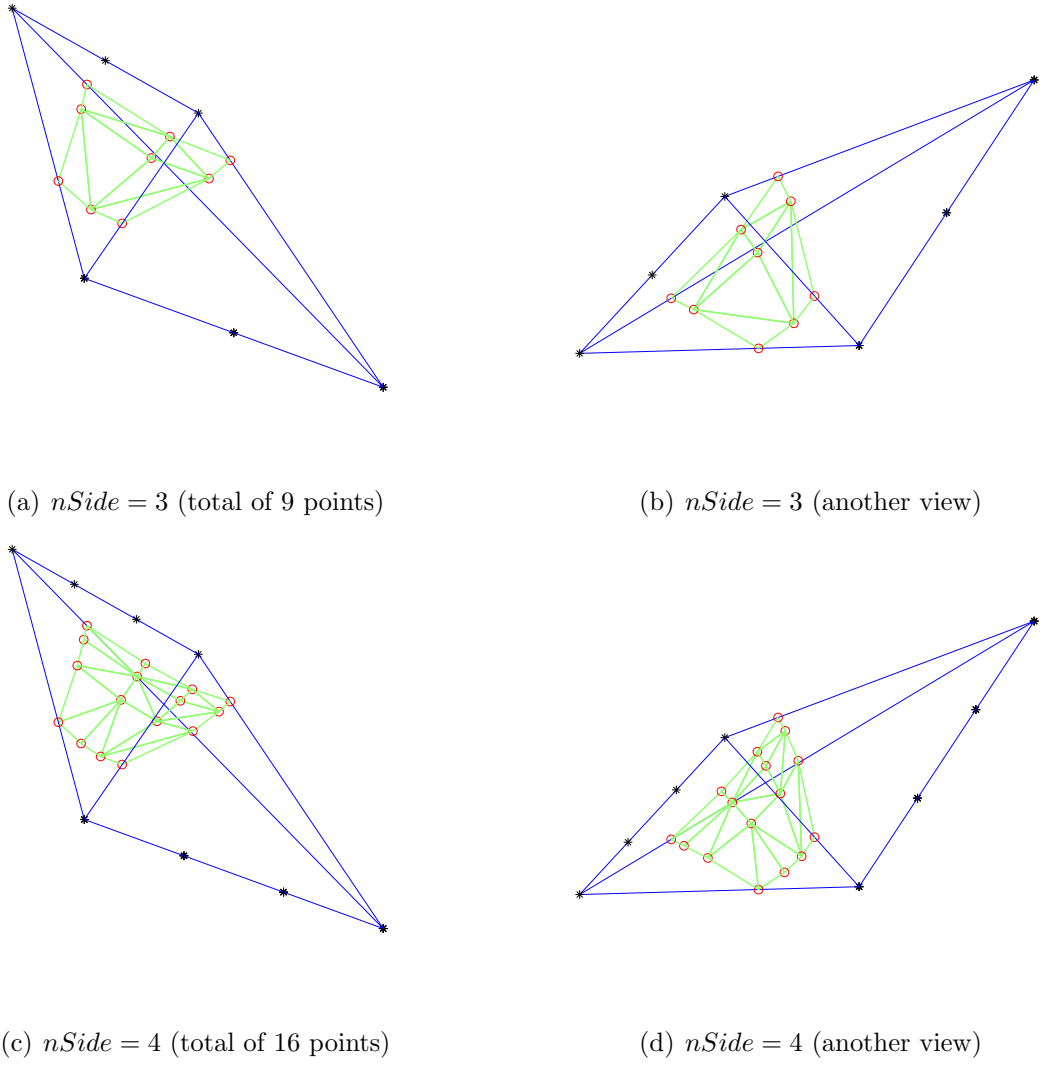
Since the total degree of the interpolant $p_{3,e}$ is three, after substituting for x, y and z in equation (2.15) with the expressions in equation (2.14), we will obtain a cubic polynomial in terms of m which could have up to three real roots. In our case, since we are looking for the intersections inside the tetrahedron, only the roots in between 0 and 1 are relevant and in most cases there will only be one such real root between 0 and 1. Note that it is easy to show that there must be at least one real root between 0 and 1.

Four intersections with four different edges

The situation is illustrated in Figure 2.25(b). As can be seen, four out of the six edges of the tetrahedron have exactly one intersection with the contour surface. In other words, exactly two edges have no intersection with the contour surface. The strategy to draw a wire frame contour surface in this situation is as follows

> *Consider some regularly-spaced points on each of the two edges with no intersection. Draw lines by connecting each regularly-spaced point from one edge to all the regularly-spaced points on the other edge and then find the intersections of these lines with the contour surface. A refined contour surface will be formed by connecting the intersection points in an obvious way.*

Figure 2.32 shows how the strategy works to plot wire frame contour surfaces using 3 and 4 points on each edge of the tetrahedron with no intersection (total of 9 and 16 points). Similar to the 'three intersections' situation, we can adjust number of selected points to have a smooth plot over all tetrahedra.



(a) $nSide = 3$ (total of 9 points)

(b) $nSide = 3$ (another view)

(c) $nSide = 4$ (total of 16 points)

(d) $nSide = 4$ (another view)

Figure 2.32: Two different views of drawing contour surface for 'four intersections with four different edges' situation in 3D. Black stars show the selected points on two edges with no intersection.

Chapter 3

Adaptive Mesh Refinement

3.1 Introduction

In the numerical solution of many scientific and engineering problems involving Partial Differential Equations (PDEs), the finite element and finite volume methods are two commonly used effective tools. Both techniques require a spatial decomposition of the computational domain into a mesh. A mesh is a union of simple geometric elements such as quadrilaterals or triangles in two dimensions and hexahedra or tetrahedra in three dimensions.

Adaptive mesh refinement is an important technique for saving computational resources particularly when solving problems modeling multi-scale phenomena. Rather than using a uniform grid, which would have to be at the resolution of the smallest feature of interest, adaptive grids permit one to use a fine resolution where it is needed, and a coarser resolution elsewhere. This can result in significant time and space savings over simply using a uniform grid.

As part of our work, we focus on adaptive mesh refinement techniques for two-dimensional partial differential equations. We introduce a class of adaptive mesh refinement algorithms that makes use of DEI-based interpolants to define the associated

monitor function. We investigate how well such 'generic' mesh selection strategies, based on properties of the problem alone, can perform when compared with special-purpose monitor functions that are designed for a specific PDE method.

In the following sections, we first present an overview of a standard three-stage algorithm. We then discuss how the DEI can be used to present an AMR approach which attempts to adapt the mesh based on the properties of the DEI and the PDE.

3.2 Previous and Related Work

In general, the domain of interest of a physical problem can be any irregularly shaped domain. In such a case, one usually wishes to create an initial mesh that conforms to the geometry of the problem. In order to generate such meshes, a three-stage algorithm can be used as follows:

- Stage 1: Generate a minimal (coarse) mesh to reflect the geometry of the domain and be consistent with the topology of the problem domain. The output of this stage would be a mesh M_1.

- Stage 2: Refine M_1 to obtain a mesh preserving the 'good' triangular shapes (M_2). For example, a mesh with the minimum angle greater than 25. The input of this stage is M_1 and its output is the mesh bvM_2.

- Stage 3: Refine M_2 which is assumed to be fine enough to support computing an approximate solution for which the monitor function used in this stage is in the asymptotic behaviour range. The input of this stage is M_2 and its output is the mesh M_3.

For the first two stages, Delaunay meshing has pretty well proven itself to be the method of choice. In the early 1990's a Delaunay refinement method was rigorously proven to produce meshes with no 'small' angles for Stage 2, and its practical performance

Start with the mesh generated by the second stage, M_2 (with corresponding triangles, T_0)
Solve the PDE on M_2 and form the associated piecewise polynomial (a DEI)
$k = 0$
while the termination condition is not satisfied **do**
(i) Determine a set of triangles, $S_k \subset T_k$, to refine based on a candidate monitor functions
(ii) Refine the triangles in S_k to form T_{k+1}
Solve the PDE on T_{k+1} and form the associated piecewise polynomial
$k = k + 1$
end do

Figure 3.1: Overview of Stage 3 of our adaptive mesh refinement algorithm.

and computational efficiency was established. The refinement uses the circumcircle center as the primary choice of insertion vertex. However, a more complex refinement is made if the circumcircle center lies near, or over, a boundary edge. One can refer to [9] [38] and [41] for a more detailed description and justification for the three-stage view.

Our focus and contribution to this large topic is on the third stage where we define and investigate alternative generic monitor functions that are each based on using DEIs. A monitor function tells us how appropriate an individual triangle is to be selected for refinement. It may rely on the quality of the triangle or be a direct measure of the error associated with the triangle. In any case, it must be easy and inexpensive to compute. It also can be used to identify if the termination criteria of the AMR algorithm is satisfied. Figure 3.1 illustrates our adaptive refinement algorithm for this stage. Two main steps are indicated by (i) and (ii) in the figure. In the mesh selection step, denoted by (i), we identify the mesh elements appropriate to refine based on one of the four monitor functions, 'defect', 'surface area', 'stepwise error', or 'interpolation error'. In this step, we identify a set of triangles where the monitor function is greater than a threshold value

(for example twice the average). It is these triangles that are refined on this step. In the case that the termination condition is an upper bound on the final number of mesh elements, we also take this into consideration in the mesh selection step. In section 3.3, the use of the DEI in the mesh selection step will be explained in more detail.

We chose MGGHAT [31] as an example of a standard PDE solver which solves elliptic PDEs and is callable as a Fortran subroutine. MGGHAT will return the solution on a mesh of at most K triangles using its own internal monitor function. We first apply MGGHAT to determine the MGGHAT 'benchmark' result for comparison. We can then determine what the results would have been if one of our monitor functions had been used for stage 3 by using MGGHAT to determine M_2 and then determining M_3 (without reference to MGGHAT) using the stage 3 algorithm of Figure 3.1 with the same stopping criteria (at most K triangles).

For the mesh refinement step, denoted by (ii) in Figure 3.1, a combined local refinement and local reconnection algorithm is used. In section 3.4, some specific details about our implementation for this step will be presented.

3.3 Mesh Selection Step

In [13], Enright suggested the use of approximations to 'arc length' and 'defect' to define monitor functions, based on CRKs, for mesh refinement in ODEs. Note that, in the case we are considering, in 1D (ODEs) a mesh element would be a line segment; while in 2D a mesh element would be a triangle; and in 3D a mesh element would be a tetrahedron. The mesh points can then be relocated based on 'equidistributing' this measure. For a two-dimensional problem, one can extend the notion of 'arc length' to 'surface area'. Moreover, as an alternative to 'defect' and 'surface area' measures, one can locally apply the difference between the approximations in two consecutive steps ('stepwise error') or the difference between the approximations using two different interpolants ('interpolation

error') as a monitor function. In the following, a more precise definition for each monitor function will be provided.

3.3.1 'Defect-based' Monitor Function

Consider the case where the underlying PDE is a two-dimensional second-order elliptic PDE of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x, y, u, u_x, u_y),$$

and $U(x, y)$ is the piecewise polynomial associated with the PCI approach. We define the associated defect or residual, $D(x, y)$, by

$$D(x, y) = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} - g(x, y, U, U_x, U_y).$$

$U_x(x, y)$ and $U_y(x, y)$ are also bivariate piecewise polynomials which are approximations to $u_x(x, y)$ and $u_y(x, y)$. The particular monitor function we use in our 'defect' mesh refinement scheme is the product of an estimated maximum defect (on the triangle) and the area of the triangle. For the mesh element t_i, let r_i be

$$r_i = A_i \times D_i \approx \int_{t_i} |D(x, y)| \, dxdy,$$

where D_i is the magnitude of a sampled defect on t_i and A_i is the area of t_i. Now since $\bigcup_i t_i = \Omega$ (The whole domain), we have

$$R \equiv \sum_{i=1}^{N} r_i \approx \int_{\Omega} |D(x, y)| \, dxdy,$$

and equidistributing the monitor function r_i is related to an effective control of $\int_\Omega |D(x, y)| dxdy$.

3.3.2 'Surface Area' Monitor Function

An approximation to the value of the surface area associated with each triangle, t_i, determined from $U(x, y)$, can also be used to define an appropriate monitor function. If

$t_i, i = 1, 2, \cdots, N$ are the triangles associated with the current mesh then the surface area associated with t_i and the approximate solution $U(x, y)$ is

$$A_{t_i} = \iint_{t_i} \sqrt{1 + (\frac{\partial U}{\partial x})^2 + (\frac{\partial U}{\partial y})^2} \ dxdy,$$

and this value can be used as the 'surface area' monitor function.

3.3.3 'Stepwise Error' Monitor Function

Using the difference between the approximations obtained in two consecutive refinement steps, one can define another measure of the error which can be used as a basis for a suitable monitor function. Starting with a coarse mesh along with a DEI, $U(x, y)$, and one can refine each triangle once and find the associated 'refined' piecewise polynomial interpolant, $\overline{U}(x, y)$. One can then define

$$SE_i = \frac{\sum_{k=1}^{K} |\overline{U}(x_k, y_k) - U(x_k, y_k)|}{K}, \quad i = 1, 2, \cdots, N$$

where N is the number of mesh elements in the refined mesh, K is the number of sample points used to find the average error in each mesh element and SE_i is the 'stepwise error' monitor function associated with t_i.

3.3.4 'Interpolation Error' Monitor Function

Another approach that uses DEIs is to use two different DEI approximations for each triangle and then define their difference as a monitor function [14]. That is, if $U(x, y)$ and $\tilde{U}(x, y)$ are two bivariate piecewise polynomials which approximate $u(x, y)$ differently, the 'interpolation error' monitor function for the element i is

$$IE_i = \frac{\sum_{k=1}^{K} |\tilde{U}(x_k, y_k) - U(x_k, y_k)|}{K}, \quad i = 1, 2, \cdots, N$$

where N is the number of mesh elements in the refined mesh and K is the number of sample points used to approximate the average error associated with each mesh ele-

ment. Note that $U(x, y)$ and $\tilde{U}(x, y)$ could be two interpolants of different order or two interpolants of the same order.

3.4 Mesh Refinement Step

In Appendix A.1, we review the advantages and disadvantages of the four most popular local refinement algorithms, Mid-point Insertion, Bisection, Regular Refinement and Newest-Node. In addition, in Appendix A.2, we introduced Delaunay triangulation as the most widely used local reconnection algorithm. Considering only the local refinement approach, one could insert a vertex inside the triangle. However, by combining a local refinement and a local reconnection approach, any vertex inside the circumcircle of the triangle can be selected since any triangle whose circumcircle includes that vertex will be refined. Intuitively, in order to get new well-shaped triangles by the node insertion, one may argue that the new node should be positioned as far from existing nodes as possible. The position inside the circumcircle of a triangle, which is as far from existing nodes as possible, is exactly at the center of the circumcircle of the triangle or simply at the 'circumcenter' of the triangle equidistant from its three nodes. As Hjelle argued in [24], it is also more distant from other nodes in the triangulation than it is from the three nodes of the triangle. Therefore, the circumcenter of a triangle seems to be a good choice since it improves the quality of the existing triangles. However, it is not always possible to insert the circumcenter of a triangle for refinement.

In some cases, the circumcenter may lie outside of the domain. In such cases, we simply apply the Mid-point algorithm. However, after adding the mid-point of the triangle, we still apply the local reconnection algorithm to make the triangulation Delaunay again. The Mid-point insertion algorithm might create triangles with small angle since it halves the existing angles. This could be resolved by the Delaunay algorithm which maximizes the minimum angle. However, if the triangle locates at a boundary, the small

angles might be left even after making the triangulation Delaunay, since Delaunay triangulation might not be able to remove poor triangles at boundaries. A solution is to use a simple 'Bisection' method when the triangle is a boundary triangle with the longest edge locating at the boundary. Thus, here is our mesh refinement algorithm:

- Find the most appropriate point p to add
  - ◇ Use circumcenter insertion if the circumcenter of the triangle lies inside the domain.
  - ◇ Use Bisection method if the longest edge of the triangle lies at the boundary.
  - ◇ Use Mid-point insertion otherwise.
- Insert p using an incremental approach of Delaunay triangulation.

Fortunately, inserting a point p into a Delaunay triangulation appears to be a local process. In most cases only a limited number of triangles near p needs to be rearranged when inserting p. More precisely, only the triangles whose circumcircle contains p in its interior will be modified.

Chapter 4

Error Estimation

4.1 Introduction

While an important task in numerical analysis is to develop reliable and efficient algorithms, it is also desirable to be able to estimate the error in the approximate solution when we solve a problem. An error estimate can help us know how much we can trust the solution or even the model itself. Our contribution in this area is to investigate the use of a DEI to introduce a companion equation based on the original PDE and then use the approximate solution of this companion equation to provide an à posteriori error estimator for the error associated with the DEI, $u - U$.

4.2 The Companion Equation

In order to illustrate our approach, we will consider a two-dimensional second-order elliptic PDE of the form

$$u_{xx} + u_{yy} = g(x, y, u, u_x, u_y), \tag{4.1}$$

defined over a uniform rectangular coarse mesh of size $n \times n$. (ie, $(n+1)^2$ mesh points and n^2 mesh elements.) We will assume that the DEI, $U(x, y)$, provides $O(h^p)$ off-mesh

approximations to the solution $u(x, y)$ for all (x, y) in the domain of interest, Ω (where $h = \frac{1}{n}$). $U(x, y)$ satisfies a perturbed PDE

$$U_{xx} + U_{yy} = g(x, y, U, U_x, U_y) + \delta_1(x, y) \qquad \|\delta_1(x, y)\| < TOL_1, \tag{4.2}$$

where the value of TOL_1 can be estimated by sampling the defect on each mesh element. (Note that $\delta_1(x, y)$ can be computed accurately at any point - it is only its maximum value that we are estimating by TOL_1.) Now, we know the exact global error for (x, y) in Ω is defined by

$$e(x, y) = u(x, y) - U(x, y). \tag{4.3}$$

From our assumption above $e(x, y)$ is $O(h^p)$. If we differentiate both sides of (4.3) with respect to x and y, we obtain

$$u_x(x, y) = U_x(x, y) + e_x(x, y), \tag{4.4}$$

$$u_y(x, y) = U_y(x, y) + e_y(x, y), \tag{4.5}$$

$$e_{xx}(x, y) = u_{xx}(x, y) - U_{xx}(x, y), \tag{4.6}$$

and

$$e_{yy}(x, y) = u_{yy}(x, y) - U_{yy}(x, y). \tag{4.7}$$

By considering equations (4.1), (4.2), (4.4) and (4.5), if we add equations (4.6) and (4.7), we observe that $e(x, y)$ satisfies the following PDE

$$\begin{aligned} e_{xx} + e_{yy} &= (u_{xx} + u_{yy}) - (U_{xx} + U_{yy}) \\ &= g(x, y, u, u_x, u_y) - g(x, y, U, U_x, U_y) - \delta_1(x, y) \\ &= g(x, y, U + e, U_x + e_x, U_y + e_y) - g(x, y, U, U_x, U_y) - \delta_1(x, y) \\ &\equiv h(x, y, e, e_x, e_y). \end{aligned} \tag{4.8}$$

We call equation (4.8), the companion equation which can be approximated locally on each mesh element, using the same PDE method in order to obtain an estimate of the

global error. For each local problem, we simply consider Dirichlet boundary conditions $e(x, y) = 0$. However, for the elements located adjacent to the boundary of the Ω, we can use the exact boundary condition $e(x, y) = u(x, y) - U(x, y)$ for the edge of the element that corresponds to the boundary of Ω. We then solve each local problem on a mesh of size 2×2 (see Figure 4.1) which corresponds to a global mesh of size $2n \times 2n$.

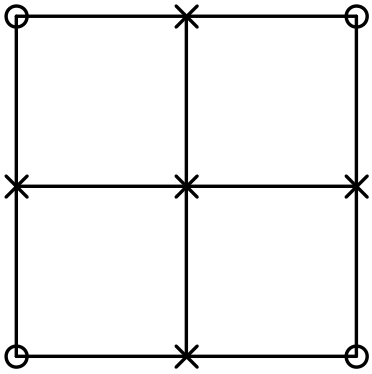


Figure 4.1: A mesh of size 2×2 on which we solve the companion equation for each local problem.

After doing this, we would also determine a bivariate piecewise polynomial, $E(x, y)$ which is an approximation to the exact error $e(x, y)$ for all (x, y) in the domain of interest,

$$E_{xx} + E_{yy} = h(x, y, E, E_x, E_y) + \delta_2(x, y) \qquad \|\delta_2(x, y)\| < TOL_2. \tag{4.9}$$

Note that the value of TOL_2 can be estimated by sampling the defect on each element associated with $E(x, y)$.

Moreover, we might be able to improve the accuracy of the solution of the original PDE by adding the solution of this companion equation (this technique is similar in spirit to the use of iterative improvement).

$$\tilde{U}(x, y) = U(x, y) + E(x, y). \tag{4.10}$$

Solving the companion equation for each mesh element can be a time-consuming process, but one can solve the equations in parallel since each one is a completely local computation. Furthermore, when we apply the underlying PDE solver to the companion equation on element e, we should be able to exploit the fact that we need only compute an approximate solution on the coarsest possible refinement of e (ie., with no error control or mesh refinement). In the next section, we will discuss a parallel implementation for solving the companion equation over each mesh element.

The estimates depend only on information that is locally available to each element and the local PDE associated with each element can be solved in parallel. The difficulty in using local information only to define the local PDE is that one does not have a natural definition of what the local boundary conditions should be. We have tried various approximations to these boundary conditions and some choices lead to a good estimate of the magnitude of the global error over the whole domain. Such estimates provide a good measure of the location and magnitudes of the maximum error over the complete domain, but they are not accurate enough to be used to improve overall accuracy of $U(x, y)$. This is in contrast with the ODE case where an error estimate based on integrating a companion ODE (defined in terms of the CRK, $S(X)$) has a well-defined local IVP associated with each step, and the estimate can be added to $S(x)$ to significantly improve the accuracy of approximation.

It is worth noting that there is another approach which is to solve the companion equation globally on a finer mesh (of size $2n \times 2n$ for example), by considering a simple Dirichlet boundary condition $e(x, y) = u(x, y) - U(x, y)$ for all (x, y) on the boundary of the whole domain. However this 'couples' all the 'local' problems and we lose the advantage of solving a sequence of independent local problems (for each element).

4.3 Parallel Implementation

Parallel computing is a form of computation in which many instructions are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently [1]. There are several different forms of parallel computing: bit-level parallelism, instruction-level parallelism, data parallelism, and task parallelism. Data parallelism focuses on distributing the data across different computing nodes where the same calculation is performed on different sets of data [10].

For example solving the companion equation on a rectangular mesh of size 20×20 or higher can be easily divided into several concurrent processes, each working on a part of the mesh. To do this in parallel, we need very little communication between processes. In fact, we only need to scatter the domain to the processes at the beginning and gather information from the processes at the end. In other words, we need no communication during the solution of each local PDE on each part of the domain.

We have used MPI subroutines [33] to implement data parallelism for solving the companion equations. In particular, we have used the MPI_SCATTER subroutine to scatter data across the processes and the MPI_GATHER subroutine to gather data from the processes.

Chapter 5

Numerical Results

5.1 Test Problems

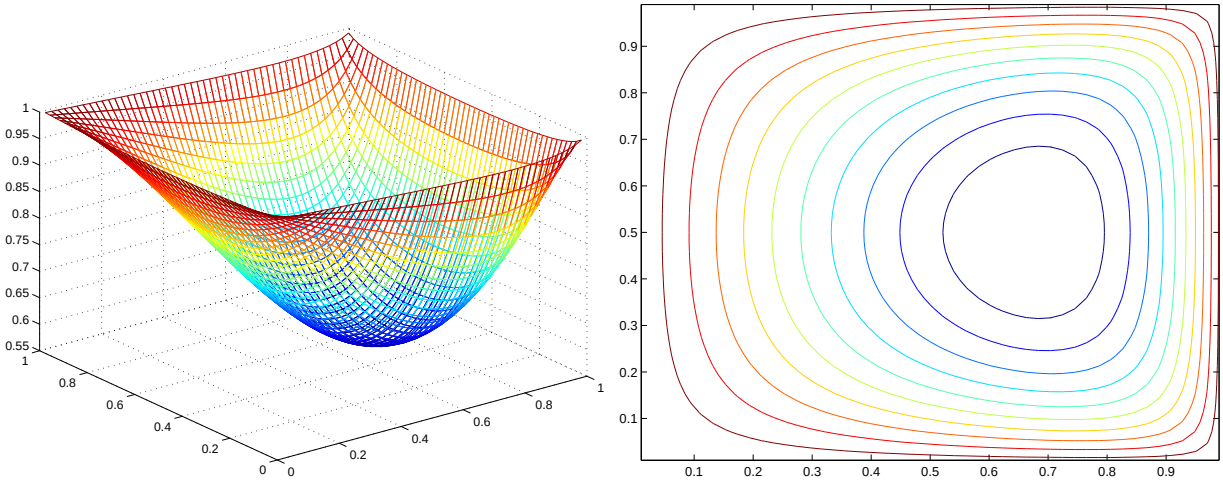
We have used several test problems to assess the relative performance of our algorithms. For the problems with no closed-form solution, in order to assess the accuracy, the approximation generated by a reliable method on a very fine mesh has been used. The first test problem (problem 39 in [35]) is a two-dimensional second-order elliptic PDE of the form,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \beta(1 - e^{2x})u = 0, \qquad (x, y) \in [0, 1] \times [0, 1],$$

with Dirichlet boundary conditions $u = 1$. Figure 5.1 shows its surface and contour plots generated using a reliable PDE solver for $\beta = 20$.

The second test problem (problem 10 in [35]) is a two-dimensional second-order elliptic PDE,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f, \qquad (x, y) \in [0, 1] \times [0, 1],$$



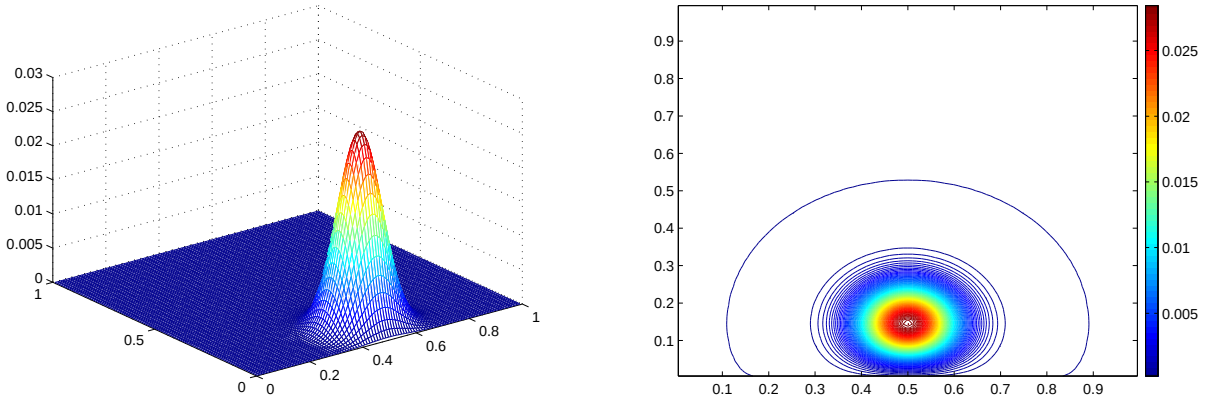
(a) The surface plot (b) The contour plots

Figure 5.1: First test problem: The surface plot and the contour plots for $\beta = 20$.

with Dirichlet boundary conditions $u = 0$. The function f is chosen to ensure the solution is

$$u(x, y) = (x^2 - x)(y^2 - y)e^{-\alpha[(x-.5)^2+(y-\beta)^2]}.$$

Figure 5.2 shows its surface and contour plots for $\alpha = 100$ and $\beta = .117$.



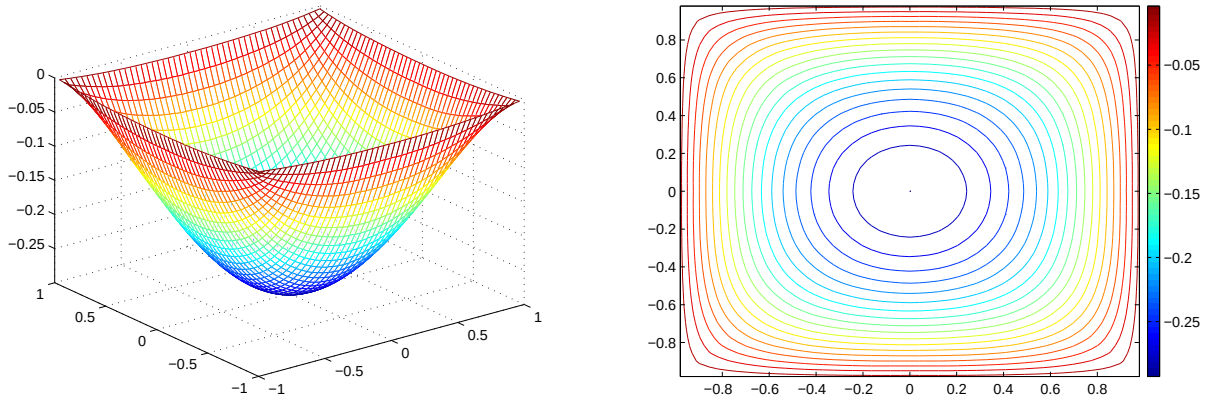
(a) The surface plot (b) The contour plots

Figure 5.2: Second test problem: The surface plot and the contour plots for $\alpha = 100$ and $\beta = .117$.

The third test problem (problem 28 in [35] for $\alpha = 1$) is also a two-dimensional second-order elliptic PDE of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 1, \qquad (x, y) \in [0, 1] \times [0, 1],$$

with Dirichlet boundary conditions $u = 0$. Figure 5.3 shows its surface and contour plots generated using a reliable PDE solver.



(a) The surface plot (b) The contour plots

Figure 5.3: Third test problem: The surface plot and the contour plots.

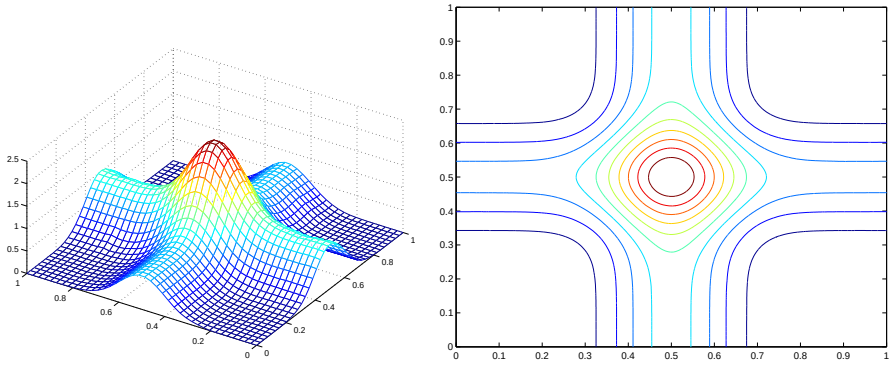
The fourth test problem is

$$\begin{aligned}\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= -100e^{(-\frac{(5-10x)^2}{2})} + (50 - 100x)u_x - 150e^{(-\frac{(5-10x)^2}{2})}e^{(-\frac{(5-10y)^2}{2})} \\ &\quad -75e^{(-\frac{(5-10y)^2}{2})} + (50 - 100y)u_y,\end{aligned}$$

on the domain $(x, y) \in [0, 1] \times [0, 1]$ and its closed-form solution is

$$u(x, y) = e^{(-\frac{(5-10x)^2}{2})} + 0.75e^{(-\frac{(5-10y)^2}{2})} + 0.75e^{(-\frac{(5-10x)^2}{2})}e^{(-\frac{(5-10y)^2}{2})}. \qquad (5.1)$$

Figure 5.4 shows its surface and contour plots.



(a) The surface plot (b) The contour plot

Figure 5.4: Fourth test problem: The surface plot and the contour plots.

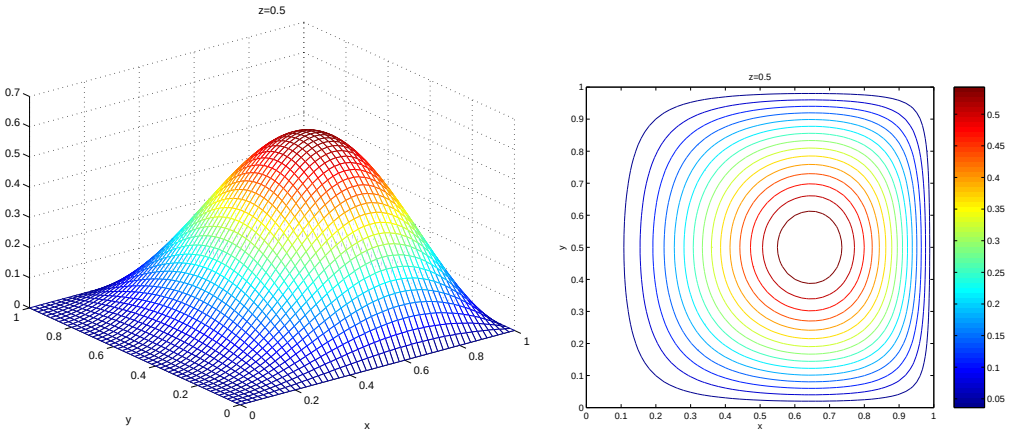
The fifth test problem is a three-dimensional second-order elliptic PDE:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 2\pi \cos(\pi x) \sin(\pi y) \sin(\pi z) - 3\pi^2 u$$

on the domain $(x, y, z) \in [0, 1] \times [0, 1] \times [0, 1]$ and its closed-form solution is

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

Figure 5.5 shows its surface and contour plots for fixed $z = 0.5$.



(a) The surface plot (b) The contour plot

Figure 5.5: Fifth test problem: The surface and contour plots for $z = 0.5$.

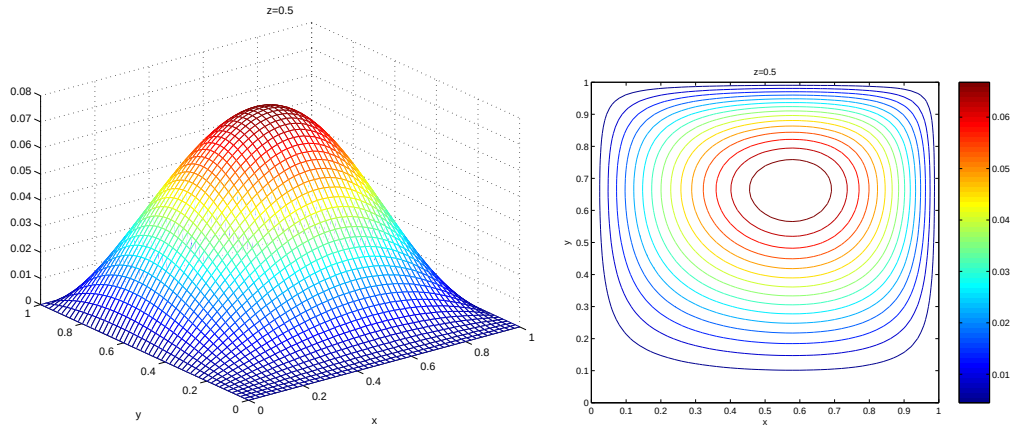
The sixth test problem is also a three-dimensional second-order elliptic PDE:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 6x(y^3 - y^2)(z^2 + 1) + (x^3 - x)(6y - 2)(z^2 + 1) + 2(x^3 - x)(y^3 - y^2),$$

where the domain is $(x, y, z) \in [0, 1] \times [0, 1] \times [0, 1]$ and its closed-form solution is

$$u(x, y, z) = (x^3 - x)(y^3 - y^2)(z^2 + 1).$$

Figure 5.6 shows its surface and contour plots for fixed $z = 0.5$.



(a) The surface plot (b) The contour plot

Figure 5.6: Sixth test problem: The surface and contour plots for $z = 0.5$.

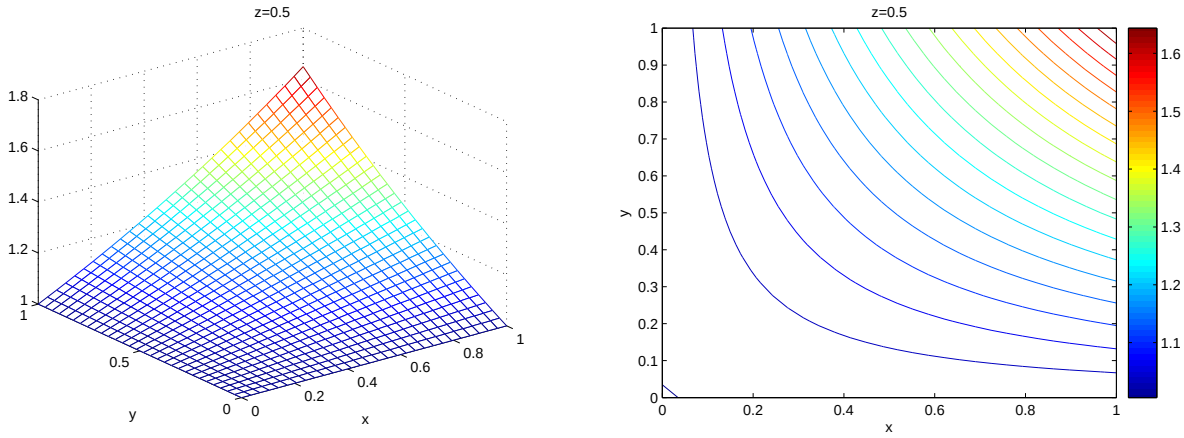
The seventh test problem is a three-dimensional second-order elliptic PDE:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = (x^2 y^2 + x^2 z^2 + y^2 z^2) exp(xyz)$$

on the domain $(x, y, z) \in [0, 1] \times [0, 1] \times [0, 1]$ and its closed-form solution is

$$u(x, y, z) = exp(xyz).$$

Figure 5.7 shows its surface and 2D contour plots for fixed $z = 0.5$.



(a) The surface plot

(b) The contour plot

Figure 5.7: Seventh test problem: The surface and 2D contour plots for $z = 0.5$.

5.2 Scattered Data Interpolation

5.2.1 Two Dimensional Case

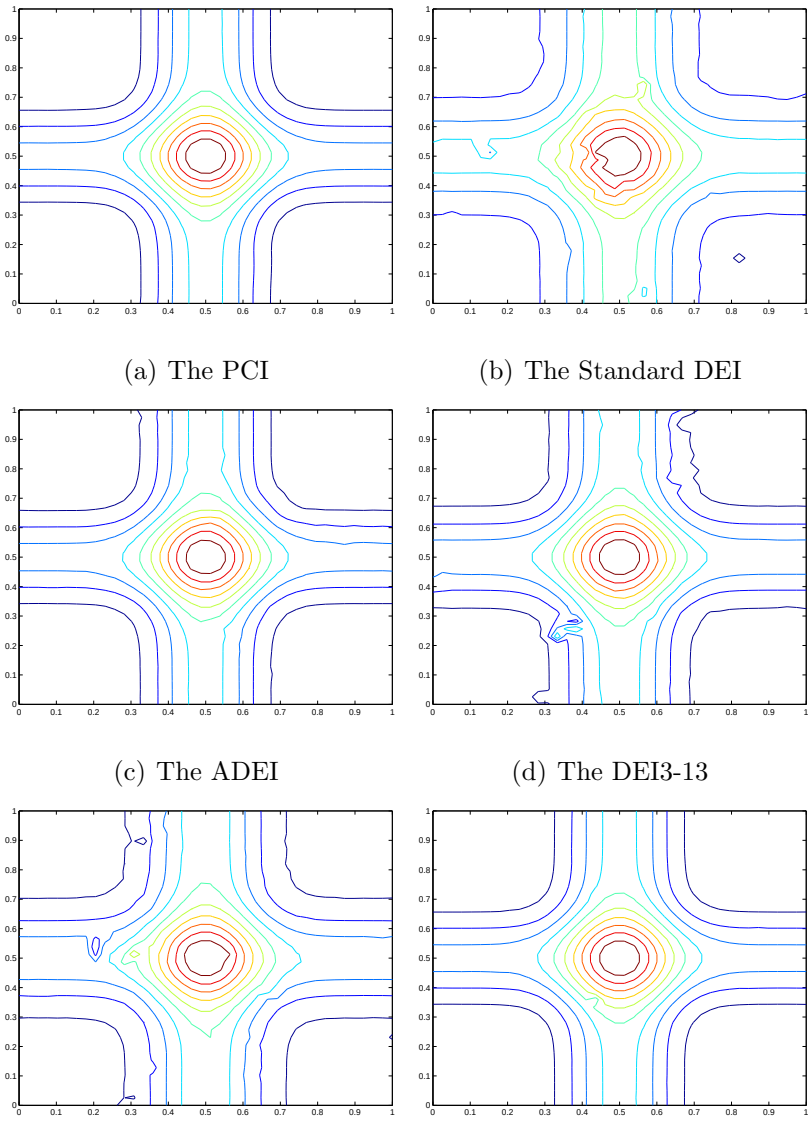
In this section, we compare the PCI with the standard DEI, the ADEI, DEI3-15, DEI3-13 and DEI4-15. In addition to time and accuracy, we consider visual appearance to show how the PCI generates a continuous contour or surface plot.

Table 5.1 shows the average error of the PCI and the other interpolants for the fourth test problem and different number of mesh points. It does not contain the average error of the ADEI for 2000 and 5000 mesh points due to the long time required for the computation. The scattered data is generated by using a random approach and triangularized by the built-in MATLAB function *delaunay*. As can be seen from the table, the PCI generates the smallest average error for all cases. Table 5.2 presents the corresponding required CPU time. The PCI is the most efficient interpolant in terms of time. It also can be seen that the ADEI, that is almost the second best one in terms of accuracy, needs much more time than the others because of the large number of iterations.

# Mesh Points	*Algorithm*					
	PCI	DEI	ADEI	DEI3-13	DEI3-15	DEI4-15
100	0.034112	0.105088	0.039668	0.901402	0.161128	0.036194
200	0.004943	0.292204	0.005256	0.005956	0.033420	0.007144
500	0.000703	0.004561	0.000876	0.001542	0.004836	0.000916
1000	0.000208	0.003134	0.000278	0.000426	0.001198	0.000210
2000	0.000071	0.001873	——	0.000917	0.000407	0.000085
5000	0.000009	0.000077	——	0.000019	0.000116	0.000011

Table 5.1: Average error for the PCI and the other interpolants on an unstructured triangular mesh with different number of mesh points for the fourth test problem.

Figures 5.8 and 5.9 show the contour plots generated by different interpolants for the fourth test problem on a triangular mesh with 500 and 2000 mesh points, respectively. The PCI is the only one that always generates continuous contour plots. The contour plots have been generated by the built-in MATLAB function *contour* on a fine grid of size 40*40.

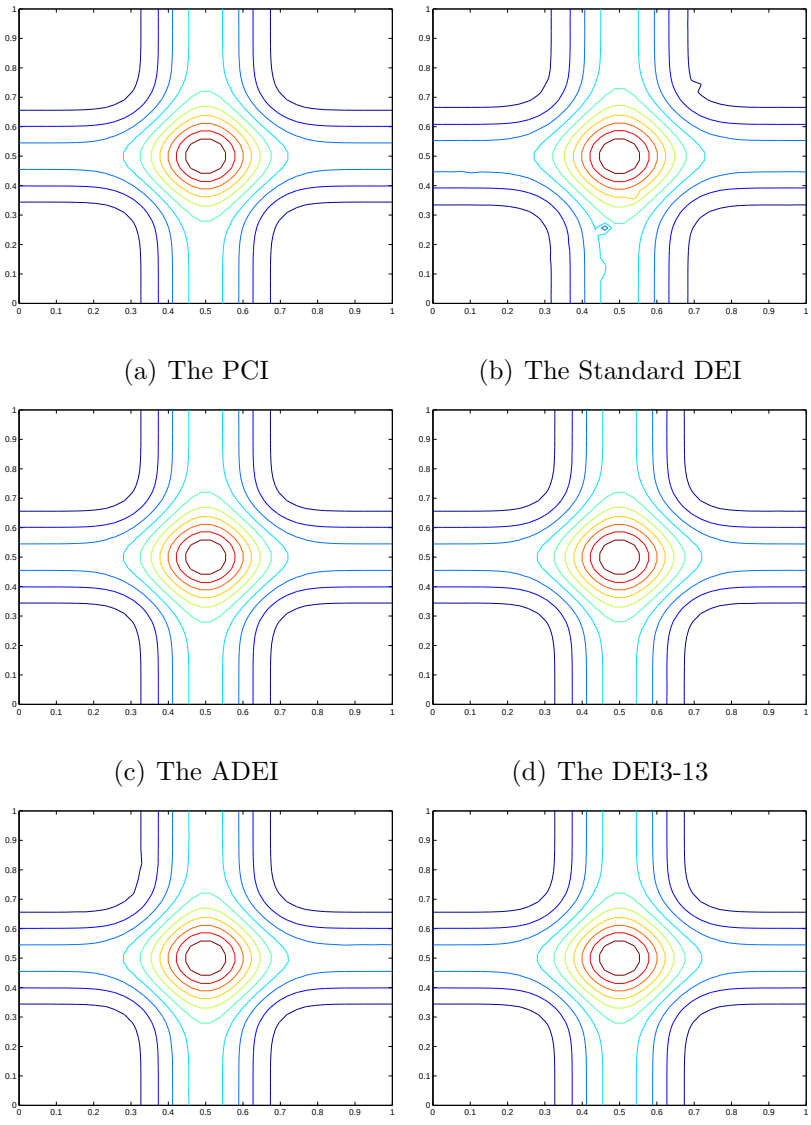


(a) The PCI (b) The Standard DEI

(c) The ADEI (d) The DEI3-13

(e) The DEI3-15 (f) The DEI4-15

Figure 5.8: The contour plots of the PCI and the other interpolants for the fourth test problem on a triangular mesh with 500 mesh points.



(a) The PCI (b) The Standard DEI

(c) The ADEI (d) The DEI3-13

(e) The DEI3-15 (f) The DEI4-15

Figure 5.9: The contour plots of the PCI and the other interpolants for the fourth test problem on a triangular mesh with 2000 mesh points.

# Mesh Points	*Algorithm*					
	PCI	DEI	ADEI	DEI3-13	DEI3-15	DEI4-15
100	0.38	1.21	95.63	0.77	1.10	1.10
200	0.77	2.48	628.13	1.59	2.25	2.19
500	2.31	6.54	3570.60	4.22	5.77	5.82
1000	3.85	13.35	19422.59	8.68	11.92	11.81
2000	7.58	26.64	——	17.25	23.72	23.51
5000	18.68	69.92	——	44.66	62.23	61.90

Table 5.2: Total required time (in terms of seconds) for the PCI and the other interpolants on an unstructured triangular mesh with different number of mesh points for the fourth test problem.

5.2.2 Three Dimensional Case

In this section, we present results for the three candidate interpolants in terms of the time required to generate them, the associated accuracy and their suitability for visualization. For accuracy, the average error over 1000 regular points in the domain has been computed. The scattered data is generated using a random distribution of points and triangularized by the built-in MATLAB *delaunay3* function.

	# Mesh Points	# Mesh Elements	*Interpolant*		
			$p_{3,e}$	$p_{2,e}$	$\hat{p}_{2,e}$
Fifth Test Problem	64	303	8.4×10^{-3}	1.065×10^{-1}	2.95×10^{-2}
	512	3146	4.97×10^{-4}	9.6×10^{-3}	5.8×10^{-3}
	4096	26880	3.76×10^{-5}	1.3×10^{-3}	1.31×10^{-4}
	32768	219273	5.56×10^{-6}	1.67×10^{-4}	4.04×10^{-5}
Observed Order			3.52	3.10	3.17
Sixth Test Problem	64	303	1.5×10^{-3}	2.03×10^{-2}	8.6×10^{-3}
	512	3146	5.42×10^{-5}	2.8×10^{-3}	2.3×10^{-3}
	4096	26880	4.76×10^{-6}	1.72×10^{-4}	3.14×10^{-5}
	32768	219273	4.77×10^{-7}	2.98×10^{-5}	1.02×10^{-5}
Observed Order			3.87	3.14	3.24

Table 5.3: Average error of the three dimensional DEI-based interpolants for the fifth and sixth test problems.

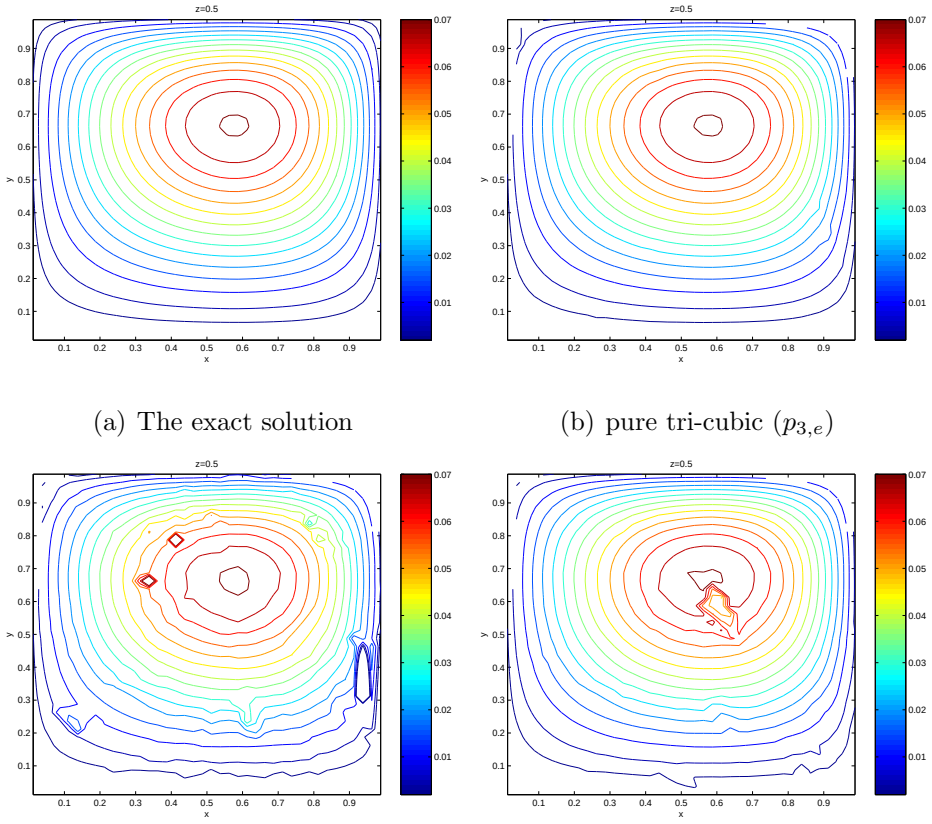
Table 5.3 shows the average error of the candidate interpolants for different number of mesh points. As expected, the pure cubic interpolant $p_{3,e}$ delivers the most accuracy for both test problems. Furthermore, Table 5.4 reports the corresponding required computer time for the candidate interpolants and both the fifth and sixth test problems. It can be seen that $\hat{p}_{2,e}$ needs less computer time than $p_{3,e}$ and $p_{2,e}$ as it has fewer unknown

coefficients to determine. Considering both test problems, the required time depends on the number of mesh elements and number of unknowns and is almost independent of the test problem.

	# Mesh Points	# Mesh Elements	*Interpolant*		
			$p_{3,e}$	$p_{2,e}$	$\hat{p}_{2,e}$
First Test Problem	64	303	1.021	2.193	0.621
	512	3146	10.71	24.68	6.098
	4096	26880	118.9	332.2	56.31
	32768	219273	3976	10682	1143
Second Test Problem	64	303	1.091	2.003	0.711
	512	3146	10.33	21.87	6.259
	4096	26880	119.3	291.8	52.81
	32768	219273	3995	10629	1016

Table 5.4: Total required time (in seconds) of the three dimensional DEI-based interpolants for the fifth and sixth test problems.

Unfortunately, none of these interpolants are globally continuous along the boundaries of the mesh elements. In fact, they provide continuity on the shared edges, but not necessarily on the shared faces. Figure 5.10 shows the contour plots associated with the different interpolants on a tetrahedron mesh with 512 mesh points for the sixth test problem. The contour plots have been generated by the built-in MATLAB *contour* procedure which requires the evaluation of the respective piecewise polynomials on a fine uniform grid of size $40 \times 40 \times 40$. The pure tri-cubic generates the most suitable results for visualization. As can be seen, tri-quadratic with total degree 3 generates better results than tensor product tri-quadratic with total degree 6.



(a) The exact solution

(b) pure tri-cubic ($p_{3,e}$)

(c) tensor product tri-quadratic ($p_{2,e}$)

(d) modified tri-quadratic ($\hat{p}_{2,e}$)

Figure 5.10: The contour plots of the exact solution and candidate interpolants for the sixth test problem for $z = 0.5$ on an unstructured tetrahedron mesh with 512 random mesh points.

5.3 Contouring Algorithms

5.3.1 Two Dimensional Case

We have compared our fast contouring algorithms, introduced in section 2.3, with the Matlab *contour* procedure and those addressed by Bradbury and Enright in [7]. In order to compare the methods, a uniform distribution of the data is considered because the Matlab *contour* procedure needs the data on a rectangular mesh. Both contouring algorithms implemented by Bradbury and Enright, FCINT and FCODE, are included for comparison. Table 5.5 shows the average error of applying the methods on a rectangular mesh with different number of mesh points for the fourth test problem. For each segment of the approximate contour curve that lies in a mesh element, a random point on the curve is considered as a sample point. Therefore the total number of sample points depends on the size of the underlying mesh and the number of contour curves. The error reported, for each sample point, is the absolute difference between the true value of $u(x, y)$ and the value of the contour curve. In addition, Table 5.6 shows the corresponding required computer time. From Tables 5.5 and 5.6, it can be inferred that our algorithms can be both more accurate and faster than the Matlab *contour* procedure, FCINT and FCODE.

Although the Intercept method is not as accurate as the SODE and ODEA methods for this rectangular mesh, it was more accurate than the others for the unstructured triangular meshes we investigated. However the total required time for the Intercept method is more. Tables 5.7 and 5.8 show the average error and corresponding required time of applying our fast contouring algorithms on an unstructured triangular mesh with 900, 1600 and 2500 mesh points for the fourth test problem. In section 2.3, an alternate ODEA algorithm, the AODEA, was introduced to improve the accuracy of ODEA method. The results of the AODEA method are also presented in Tables 5.7 and 5.8. As can be seen, the AODEA method is more accurate than the ODEA but

# Mesh Points	*Algorithm*					
	Matlab	FCINT	FCODE	Intercept	SODE	ODEA
30×30	0.3066	0.001900	0.002300	0.00001270	0.00001060	0.00001010
40×40	0.4522	0.000741	0.000811	0.00000798	0.00000291	0.00000235
50×50	0.4605	0.000273	0.000234	0.00000766	0.00000213	0.00000183

Table 5.5: Average error for the different methods for a rectangular mesh with 30×30, 40×40 and 50×50 mesh points for the first test problem.

# Mesh Points	*Algorithm*					
	Matlab	FCINT	FCODE	Intercept	SODE	ODEA
30×30	14.66	9.78	9.50	9.61	5.33	5.54
40×40	23.18	17.19	16.59	13.40	9.33	9.50
50×50	35.40	22.08	21.75	18.95	14.77	15.21

Table 5.6: Total required time (in terms of seconds) for the different methods for a rectangular mesh with 30×30, 40×40 and 50×50 mesh points for the fourth test problem.

needs more time as well. The accuracy of the AODEA method can be controlled by the value of the applied threshold.

# Mesh Points	*Algorithm*			
	Intercept	SODE	ODEA	AODEA
900	0.00000903	0.000687	0.000687	0.0000247
1600	0.00000655	0.0000299	0.0000226	0.0000156
2500	0.00000546	0.0000128	0.0000123	0.00000663

Table 5.7: Average error for the different methods for an unstructured triangular mesh with 900, 1600 and 2500 mesh points for the fourth test problem.

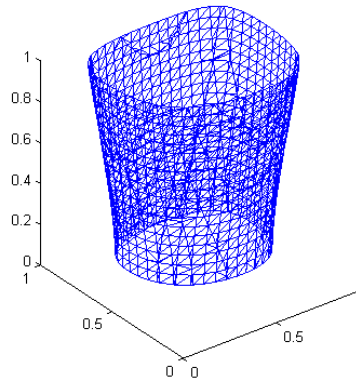
# Mesh Points	*Algorithm*			
	Intercept	SODE	ODEA	AODEA
900	10.98	5.43	5.55	6.65
1600	14.11	9.28	9.39	10.49
2500	18.35	13.62	13.79	15.54

Table 5.8: Total required time (in terms of seconds) for the different methods for an unstructured triangular mesh with 900, 1600 and 2500 mesh points for the fourth test problem.

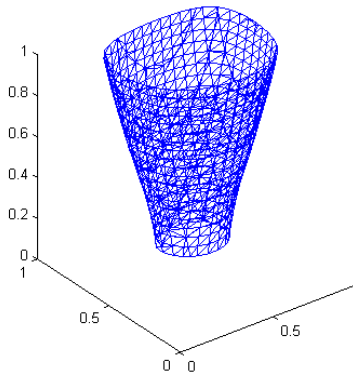
5.3.2 Three Dimensional Case

We have implemented the three dimensional contouring algorithm in MATLAB. MATLAB has a function called *patch(FV)* which can create a 3D patch using structure FV, that contains the vertices and faces information. Therefore, after constructing final refined triangles at the third phase of our algorithm, one can simply draw the triangles to have a 3D wire frame contour surface, or create a set of triangles (faces) and vertices and pass them to the *patch* routine to make a rendered graph. Figure 5.11 shows an example of both wire frame and rendered graphs for the sixth test problem for three different contour levels, $v = 0.03$, $v = 0.05$, and $v = 0.07$ generated by MATLAB *isosurface* routine.

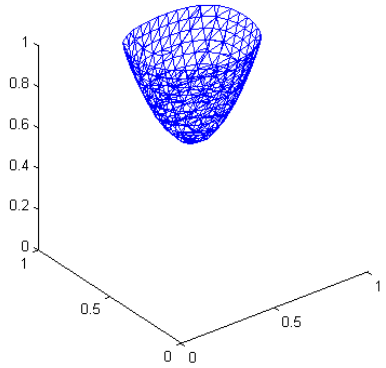
An important feature of our algorithm is its ability to refine the contour by considering extra points in the third stage. In other words, one can start with a coarse mesh and ask for a higher resolution of the contour. Figure 5.12 shows the contour surfaces generated using MATLAB *isosurface* routine and our algorithm with different number of extra points, for the sixth test problem. Note that $nSide$ is the number of points considered on each side of a triangle (in a three intersections situation) and is different from the actual number of extra points. For example, for $nSide = 2$, we consider two points on each side which means no extra points at all (three points in total). $nSide = 3$ means three points on each side or three extra points (six points in total). Table 5.9 shows the relation between $nSide$ and the number of points and triangles for each of the two desired situations.



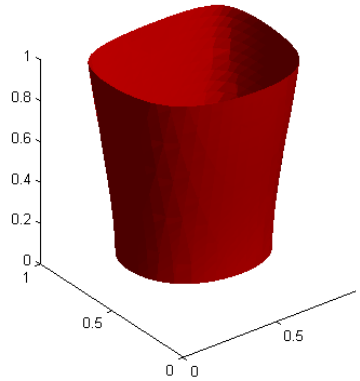
(a) Wire frame plot for $v = 0.03$



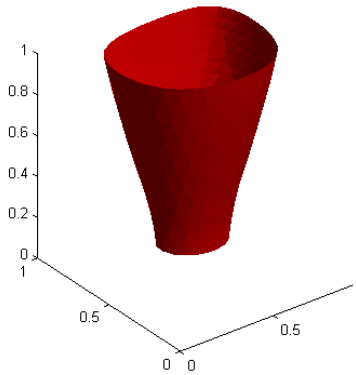
(b) Wire frame plot for $v = 0.05$



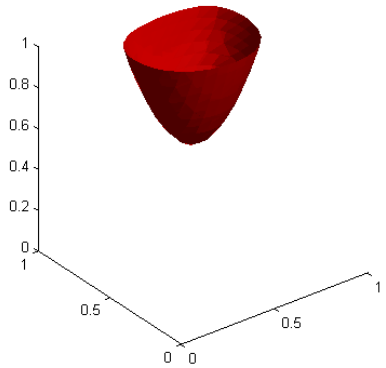
(c) Wire frame plot for $v = 0.07$



(d) Rendered plot for $v = 0.03$



(e) Rendered plot for $v = 0.05$



(f) Rendered plot for $v = 0.07$

Figure 5.11: The sixth test problem: Wire frame and Rendered contour plots for contour levels $v = 0.03, 0.05, 0.07$.

For a fixed value of $nSide$, the finer the initial grid is, the more accurate the computed contour will be. Figure 5.13 shows the contour plots generated using MATLAB *isosurface* routine and our algorithm when $nSide = 2$, starting with different initial grids. In Table 5.10, the average error of the computed contour for different initial coarse grids and value of $nSide$ has been reported. The same results have been represented in Figure 5.14 as a logarithmic graph. The error has been computed by averaging the absolute difference between the contour value (v) and the exact value of the function at the middle point

	Three Intersections			Four Intersections		
nSide	Extra Points	Total Points	Total Triangles	Extra Points	Total Points	Total Triangles
2	0	3	1	0	4	2
3	3	6	4	5	9	8
4	7	10	9	12	16	18
5	12	15	16	21	25	32
n	$\frac{n(n+1)}{2} - 3$	$\frac{n(n+1)}{2}$	$(n-1)^2$	$n^2 - 4$	n^2	$2(n-1)^2$

Table 5.9: Relation between *nSide* and the number of points and triangles for each of the two desired situations.

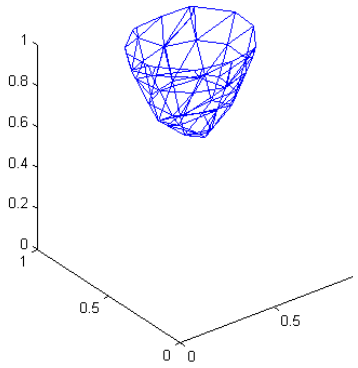
of each triangle. It also includes the same computed error for the MATLAB *isosurface* routine. Note that *isosurface* accepts only a structured grid as input, while our algorithm can start with any bounded unstructured initial grid. As can be seen in the table, the average error decreases as the value of *nSide* increases. It also decreases when we increase the number of points in the initial grid. Moreover, it shows that our algorithm is more accurate than *isosurface* even when no extra points are introduced.

	MATLAB		$nSide = 2$		$nSide = 3$		$nSide = 4$	
	Avg Err	#Δ	Avg Err	#Δ	Avg Err	#Δ	Avg Err	#Δ
$5 \times 5 \times 5$	4.53e-3	98	1.99e-3	357	1.01e-3	1617	7.93e-4	3608
$8 \times 8 \times 8$	1.76e-3	274	6.16e-4	955	2.17e-4	4299	1.57e-4	9677
$10 \times 10 \times 10$	1.15e-3	420	3.66e-4	1297	1.30e-4	5813	8.95e-5	13107
$15 \times 15 \times 15$	5.30e-4	962	1.45e-4	3372	4.67e-5	15113	2.74e-5	34003
$20 \times 20 \times 20$	2.95e-4	1722	8.08e-5	5842	2.44e-5	26241	1.28e-5	59051

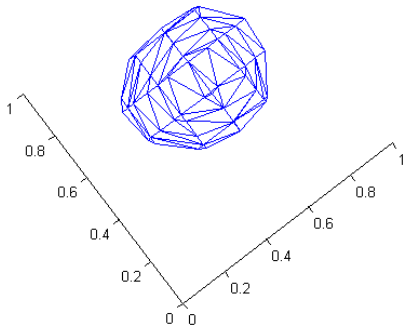
Table 5.10: The average error of the computed contour for different initial grids and different values of *nSide* for the sixth test problem ($v = 0.07$).

The contour surface generated by our algorithm can be passed to the MATLAB *patch* routine to create a rendered graph. Figure 5.15 represents such graphs generated by the MATLAB *isosurface* routine as well as our algorithm for $nSide = 2, 3, 4$ with an initial grid $8 \times 8 \times 8$ for the sixth test problem ($v = 0.07$). As can be seen, it seems that our algorithm suffers from discontinuity along the patches created by neighboring tetrahedral elements. However, this discontinuity is basically created by the interpolant we use to approximate the intersections in the second and third stages of the algorithm and has nothing to do with our contouring algorithm. As discussed in section 1.8, our DEI-based three dimensional interpolant has C^0 continuity along edges and is not necessarily continuous along faces.

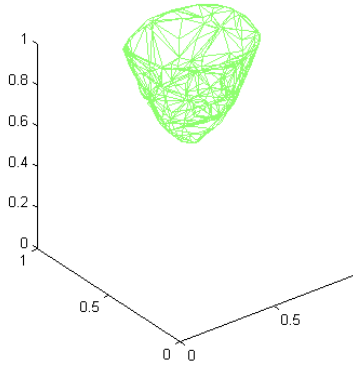
As discussed earlier, we represent the contour surface as a structure defined by a set of points and a set of triangles. One might be interested in identifying the number of final triangles representing the contour surface, before running the contouring routine. The number of points or triangles in the structure depends on three factors: Value of the contour level, v; size of initial grid; and number of extra points considered in the third stage (ie. the value of $nSide$). Unfortunately, this number cannot be identified before running the routine. However, after the first stage, we would know how many tetrahedra are participating in forming the contour surface. Therefore, if we could estimate the average number of refinements in the second stage (which makes every situation, a desired one), we would then set the value of $nSide$ in order to obtain a structure with a number of points or triangles as close as possible to the requested number by the user.



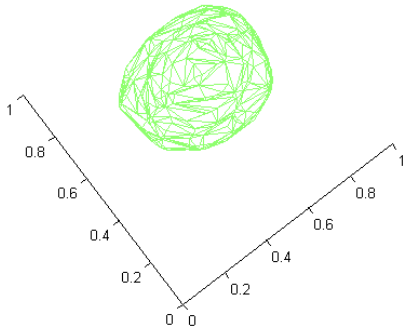
(a) Default view for MATLAB routine



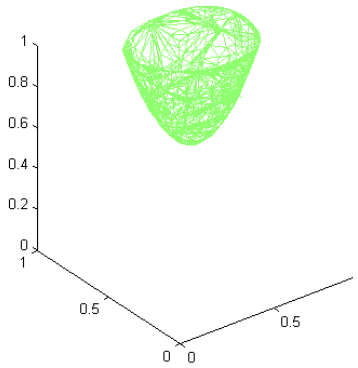
(b) Another view for MATLAB routine



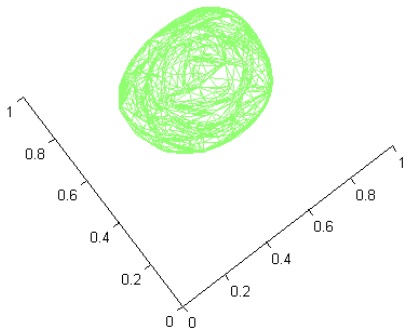
(c) Default view for $nSide = 2$



(d) Another view for $nSide = 2$

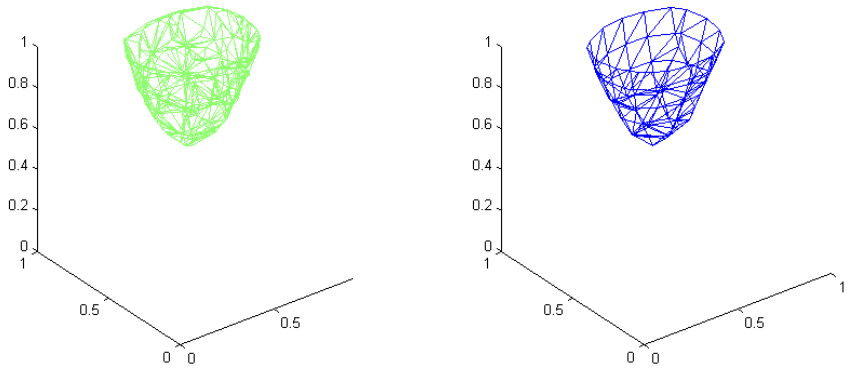


(e) Default view for $nSide = 3$

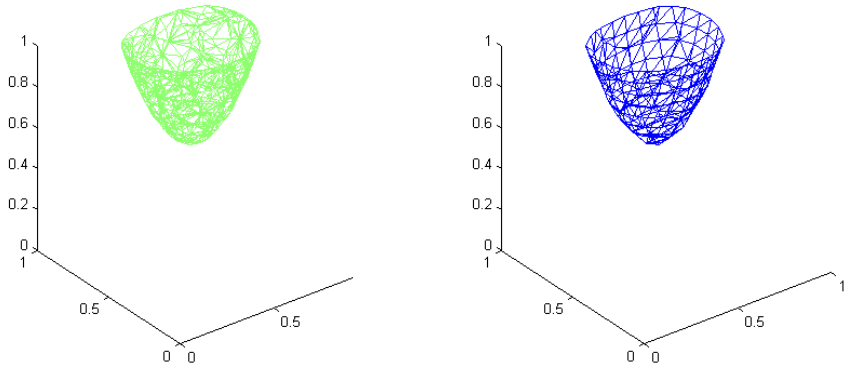


(f) Another view for $nSide = 3$

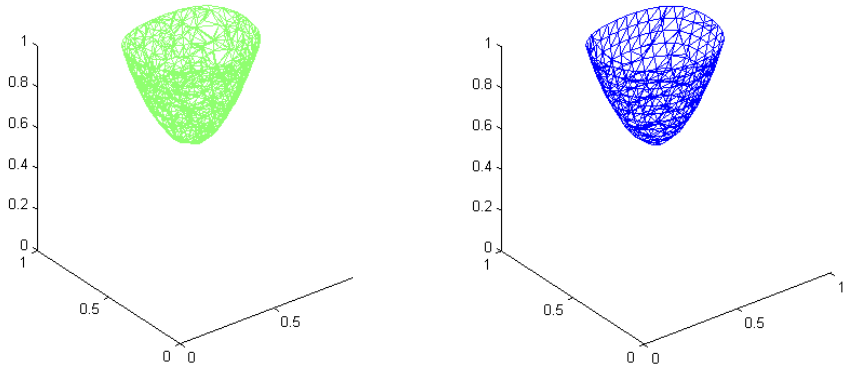
Figure 5.12: Contour plots for the sixth test problem ($v = 0.07$) generated using MATLAB *isosurface* routine and our algorithm with $nSide = 2, 3$ starting with a grid of size $8 \times 8 \times 8$.



(a) Our algorithm for $10 \times 10 \times 10$ (b) MATLAB *isosurface* for $10 \times 10 \times 10$



(c) Our algorithm for $15 \times 15 \times 15$ (d) MATLAB *isosurface* for $15 \times 15 \times 15$



(e) Our algorithm for $20 \times 20 \times 20$ (f) MATLAB *isosurface* $20 \times 20 \times 20$

Figure 5.13: Contour plots for the sixth test problem ($v = 0.07$) generated using MATLAB *isosurface* routine and our algorithm with $nSide = 2$ starting with different initial grids.

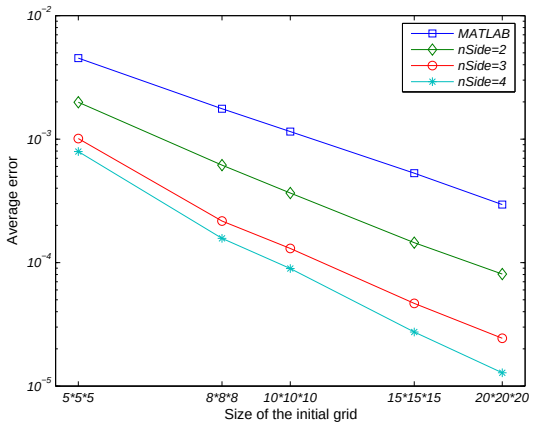
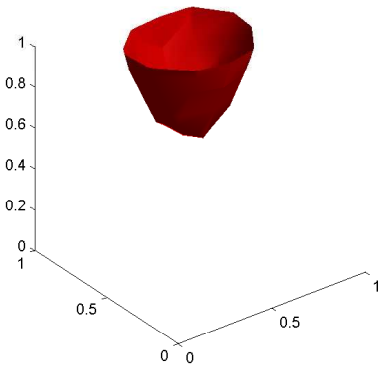
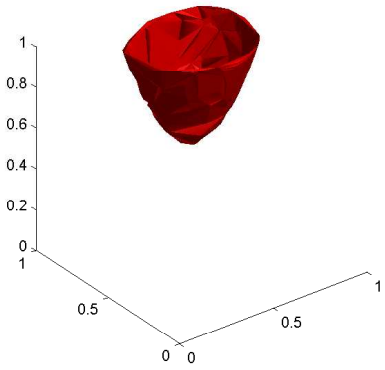


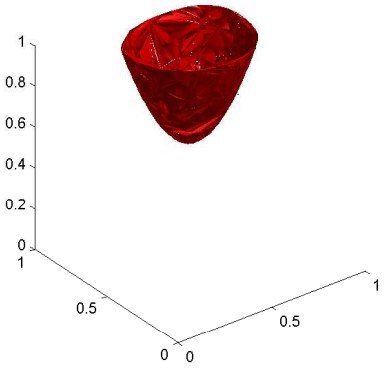
Figure 5.14: The average error of the computed contour for different initial grids for MATLAB *isosurface* routine and our algorithm with different values of $nSide$ for the sixth test problem ($v = 0.07$).



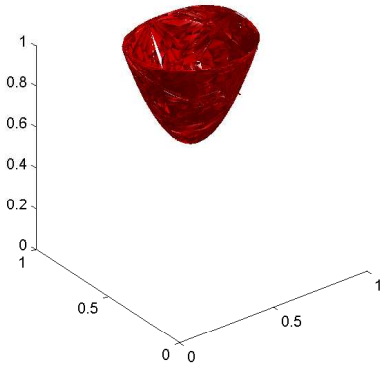
(a) MATLAB *isosurface* routine



(b) Our algorithm with $nSide = 2$



(c) Our algorithm with $nSide = 3$



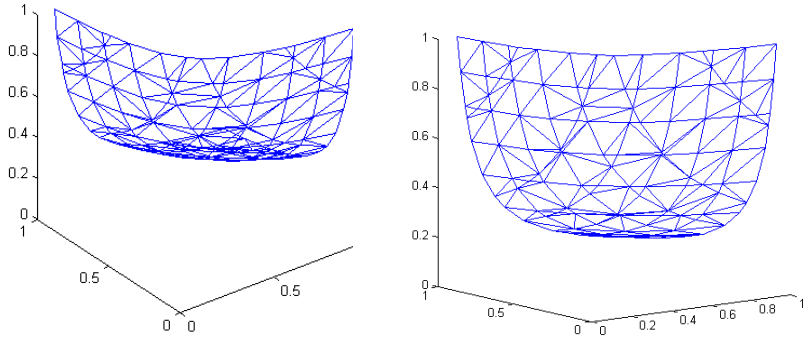
(d) Our algorithm with $nSide = 4$

Figure 5.15: Rendered contours generated by MATLAB *isosurface* routine and our algorithm with $nSide = 2, 3, 4$ for the sixth test problem ($v = 0.07$).

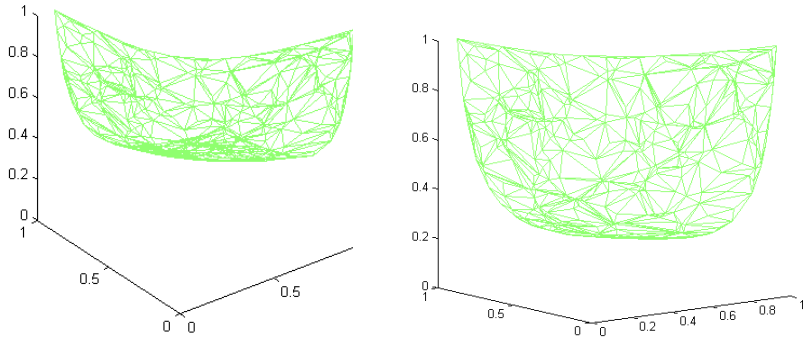
Figure 5.16 shows how the Intercept method works to generate contour surfaces for MATLAB *isosurface* routine and our algorithm with different values of $nSide$ for the seventh test problem. Similar to the sixth test problem, introducing extra points, by increasing the value of $nSide$, results in a plot with higher resolution. In Table 5.11 or Figure 5.18, one can see how much it can increase the accuracy of the computed contour surface. Furthermore, Figure 5.17 illustrates the graphs for MATLAB *isosurface* routine and our algorithm with $nSide = 2$ (no extra points), but different initial grids. Finally, Figure 5.19 represents such graphs generated by the MATLAB *isosurface* routine as well as our algorithm for $nSide = 2, 3, 4$ with an initial grid $8 \times 8 \times 8$ for the seventh test problem ($v = 1.1$).

	MATLAB		$nSide = 2$		$nSide = 3$		$nSide = 4$	
	Avg Err	$\#\Delta$	Avg Err	$\#\Delta$	Avg Err	$\#\Delta$	Avg Err	$\#\Delta$
$5 \times 5 \times 5$	4.58e-3	55	4.44e-3	167	1.41e-3	734	8.92e-4	1665
$8 \times 8 \times 8$	1.48e-3	145	1.25e-3	440	3.79e-4	1990	2.07e-4	4474
$10 \times 10 \times 10$	9.86e-4	235	6.94e-4	593	2.21e-4	2658	1.35e-4	6003
$15 \times 15 \times 15$	4.74e-4	529	3.21e-4	1513	9.67e-5	6783	5.26e-5	15251
$20 \times 20 \times 20$	2.59e-4	931	1.71e-4	2695	4.97e-5	12089	2.55e-5	27169

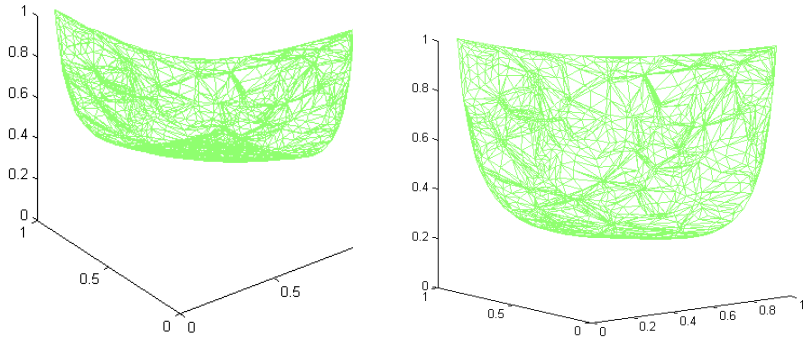
Table 5.11: The average error of the computed contour for different initial grids and different values of $nSide$ for the seventh test problem ($v = 1.1$).



(a) Default view for MATLAB routine (b) Another view for MATLAB routine

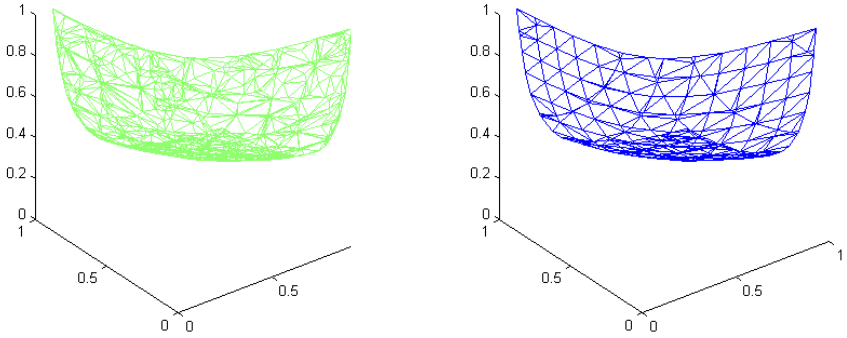


(c) Default view for $nSide = 2$ (d) Another view for $nSide = 2$

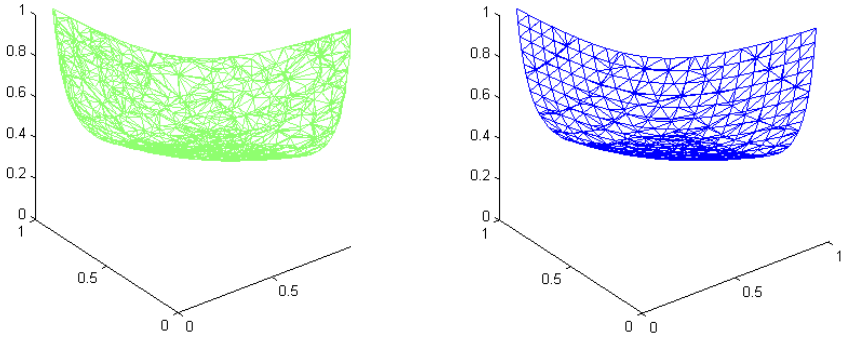


(e) Default view for $nSide = 3$ (f) Another view for $nSide = 3$

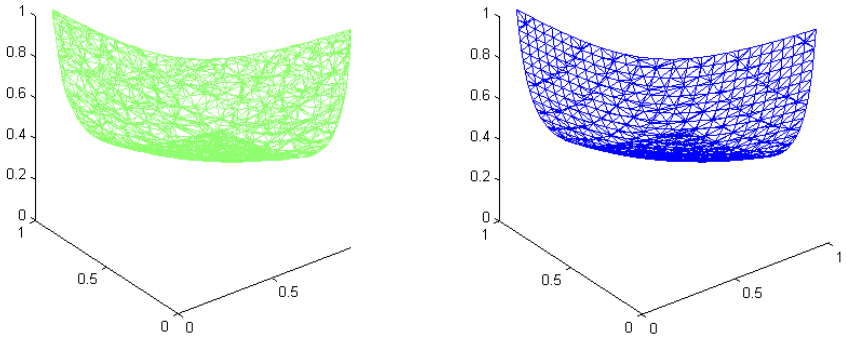
Figure 5.16: Contour plots for the seventh test problem ($v = 1.1$) generated using MATLAB *isosurface* routine and our algorithm with $nSide = 2, 3$ starting with a grid of size $8 \times 8 \times 8$.



(a) Our algorithm for $10 \times 10 \times 10$ (b) MATLAB *isosurface* for $10 \times 10 \times 10$



(c) Our algorithm for $15 \times 15 \times 15$ (d) MATLAB *isosurface* for $15 \times 15 \times 15$



(e) Our algorithm for $20 \times 20 \times 20$ (f) MATLAB *isosurface* $20 \times 20 \times 20$

Figure 5.17: Contour plots for the seventh test problem ($v = 1.1$) generated using MATLAB *isosurface* routine and our algorithm with $nSide = 2$ starting with different initial grids.

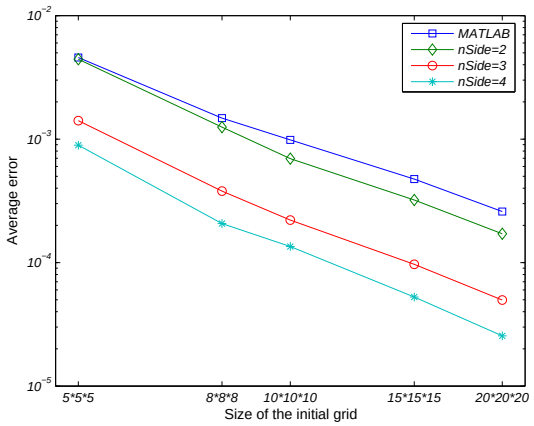
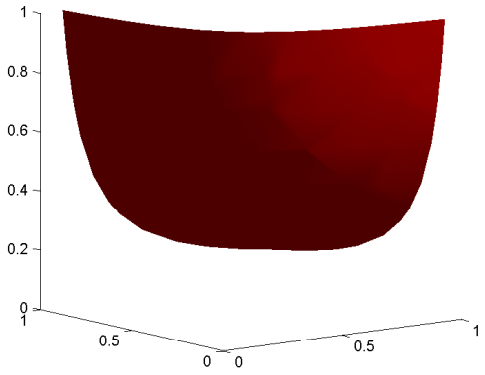
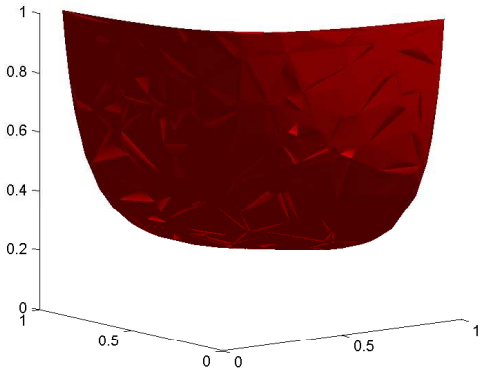


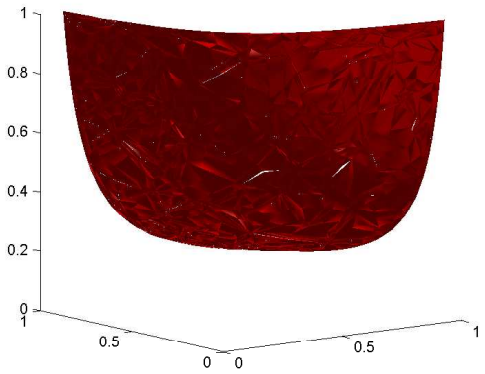
Figure 5.18: The average error of the computed contour for different initial grids for MATLAB *isosurface* routine and our algorithm with different values of $nSide$ for the seventh test problem ($v = 1.1$).



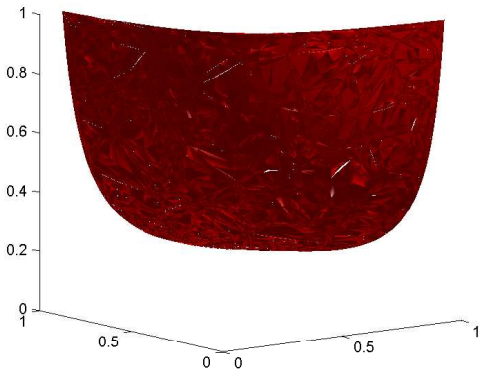
(a) MATLAB *isosurface* routine



(b) Our algorithm with $nSide = 2$



(c) Our algorithm with $nSide = 3$



(d) Our algorithm with $nSide = 4$

Figure 5.19: Rendered contours generated by MATLAB *isosurface* routine and our algorithm with $nSide = 2, 3, 4$ for the seventh test problem ($v = 1.1$).

5.4 Adaptive Mesh Refinement

The DEI adaptive mesh refinement algorithms have been implemented in both Matlab and Fortran. In order to evaluate our algorithms for problems with no analytical solution, we need a reliable PDE solver. After a survey of available packages, we chose MGGHAT [31] which is written for the solution of second order linear elliptic PDEs. It solves an elliptic PDE by the finite element method which is callable as a Fortran subroutine. Having an internal adaptive refinement procedure (based on the Newest-Node algorithm) makes it possible to qualify the relative performance of our adaptive refinement algorithms. MGGHAT starts with a 'regular' coarse mesh and takes advantage of the bisection method to keep the triangulation regular during refinement.

Figures 5.20 and 5.21 show the final triangulations using all the techniques discussed in section 3.3 for the first test problem, for 900 and 2500 mesh points, respectively. Tables 5.12 and 5.13 report the average error and the maximum defect of the approximated solution for the first test problem using several mesh selection techniques for 900, 1600, 2500 and 3600 points. Note that, since the mesh refinement section of our algorithm incrementally inserts points in the current mesh, it is possible to set the stope criterion to be the final number of points (for example 900). In addition, Figures 5.22 and 5.23 show the same information in semi-log plots. The errors have been computed over a 100×100 uniform grid inside the domain. The implemented techniques are:

- **Uniform Grid**: A uniform mesh with no adaptive mesh refinement.
- **Surface Area - PQI**: 'Surface Area' approximation of $U(x, y)$ is used in the mesh selection step and the PQI is used as the DEI.
- **Defect - PCI**: 'Defect' approximation of $U(x, y)$ is used in the mesh selection step and the PCI is used as the DEI.
- **Stepwise - PCI**: 'Stepwise Error' measure is used in the mesh selection step and

the PCI is used as the DEI.

- **FQI - PCI**: 'Interpolation Error' measure is used in the mesh selection step and the FQI is used as the DEI. The second polynomial DEI in this technique is the PCI.
- **MGGHAT**: The MGGHAT's internal adaptive mesh refinement is used.

What we can see in the table and figures is:

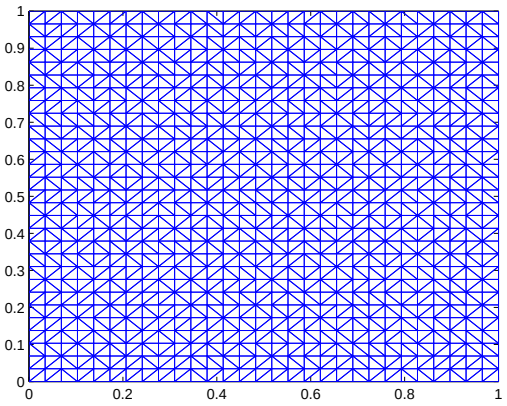
- All techniques try to generate a finer mesh close to the boundaries (specially on the right side) which is consistent with the behavior of the underlying function.
- While all DEI-based techniques generate similar refined meshes, the 'Defect' and the 'Interpolation Error' techniques appear to work slightly better.

Method	900		1600		2500		3600	
	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio
Uniform Grid	6.06×10^{-6}	1.00	1.75×10^{-6}	1.00	1.03×10^{-6}	1.00	4.89×10^{-7}	1.00
Surface Area - PQI	3.45×10^{-6}	1.75	9.46×10^{-7}	1.84	6.87×10^{-7}	1.49	3.37×10^{-7}	1.45
Defect - PCI	2.32×10^{-6}	2.61	7.98×10^{-7}	2.19	3.15×10^{-7}	3.26	1.54×10^{-7}	3.17
Stepwise - PCI	3.30×10^{-6}	1.83	9.70×10^{-7}	1.80	4.30×10^{-7}	2.39	2.32×10^{-8}	2.10
FQI - PCI	2.17×10^{-6}	2.79	6.84×10^{-7}	2.55	2.54×10^{-7}	4.05	1.14×10^{-7}	4.28
MGGHAT	3.96×10^{-6}	1.53	9.11×10^{-7}	1.92	4.15×10^{-7}	2.48	2.33×10^{-7}	2.09

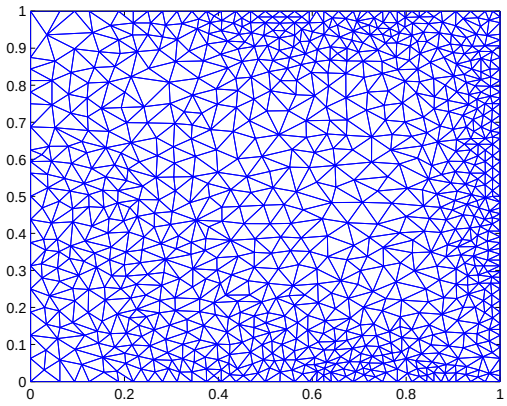
Table 5.12: First test problem: Average error and ratio of improvement of all mesh refinement techniques comparing to a uniform mesh.

Method	900		1600		2500		3600	
	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio
Uniform Grid	1.44×10^{-1}	1.00	9.05×10^{-2}	1.00	6.01×10^{-2}	1.00	3.99×10^{-2}	1.00
Surface Area - PQI	9.53×10^{-2}	1.51	5.02×10^{-2}	1.80	3.26×10^{-2}	1.84	2.72×10^{-2}	1.46
Defect - PCI	5.39×10^{-2}	2.67	3.46×10^{-2}	2.61	2.49×10^{-2}	2.41	1.50×10^{-2}	2.66
Stepwise - PCI	7.38×10^{-2}	1.95	3.96×10^{-2}	2.28	2.92×10^{-2}	2.05	1.72×10^{-2}	2.31
FQI - PCI	7.51×10^{-2}	1.91	3.18×10^{-2}	2.75	1.91×10^{-2}	3.14	1.31×10^{-2}	3.04
MGGHAT	6.53×10^{-2}	2.20	3.04×10^{-2}	2.97	1.99×10^{-2}	3.02	1.57×10^{-2}	2.54

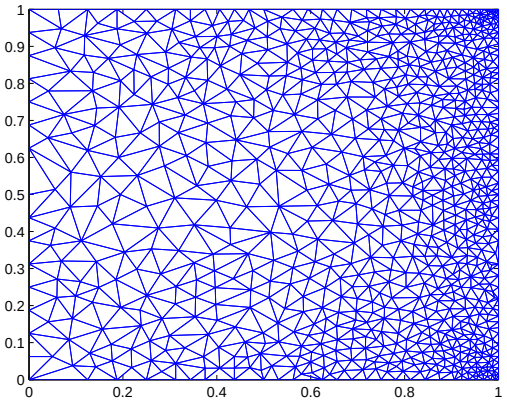
Table 5.13: First test problem: Maximum defect and ratio of improvement of all mesh refinement techniques comparing to a uniform mesh.



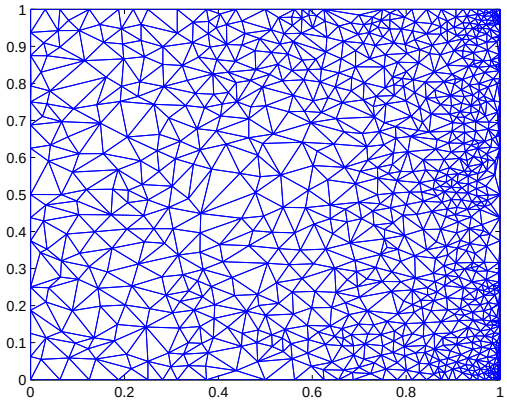
(a) Uniform Grid



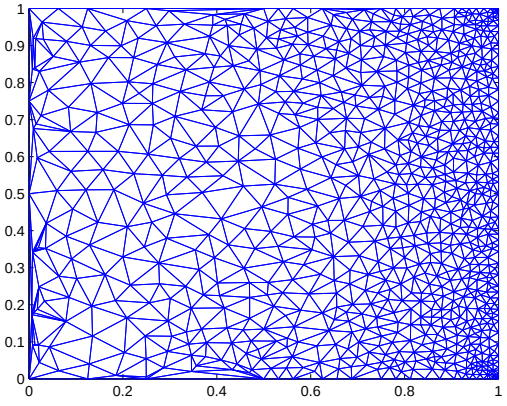
(b) Surface Area - PQI



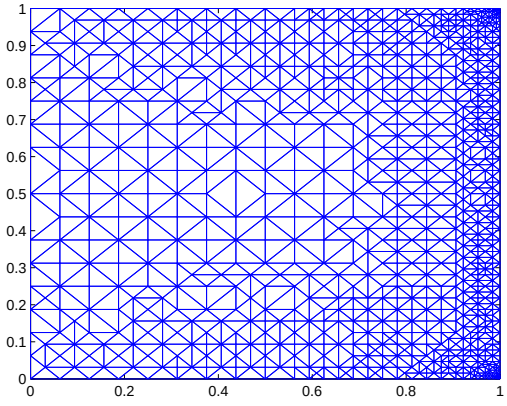
(c) Defect - PCI



(d) Stepwise - PCI

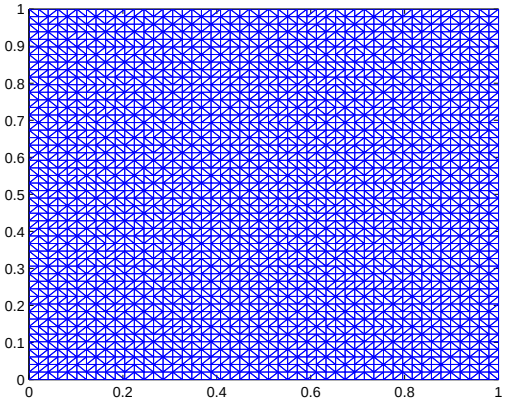


(e) PCI - PQI

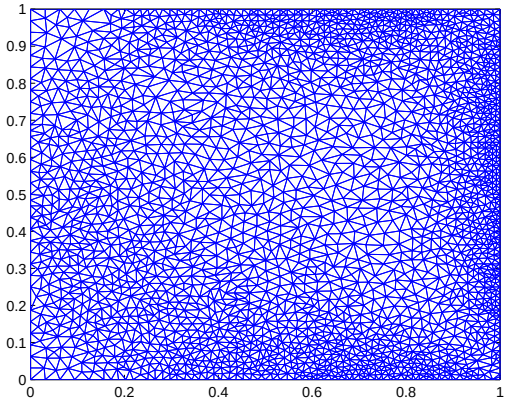


(f) MGGHAT

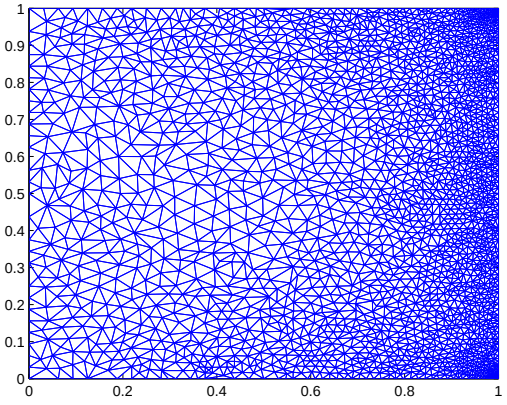
Figure 5.20: First test problem: Final meshes generated by discussed mesh refinement techniques for 900 points.



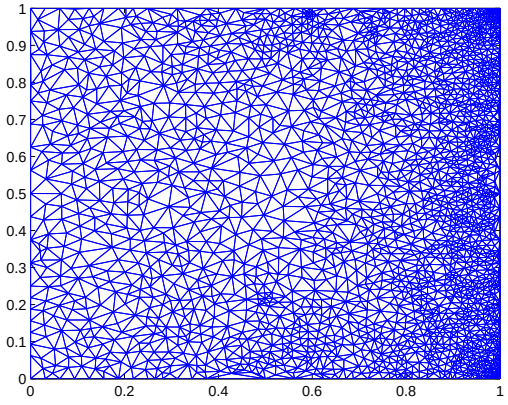
(a) Uniform Grid



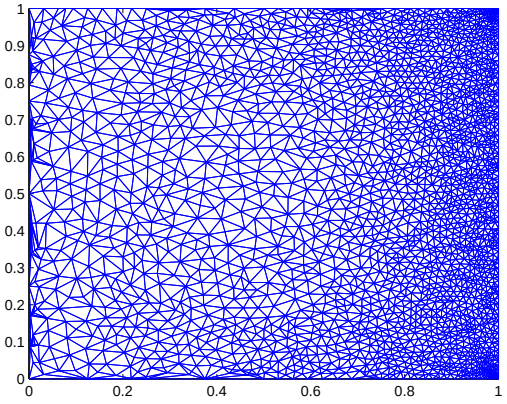
(b) Surface Area - PQI



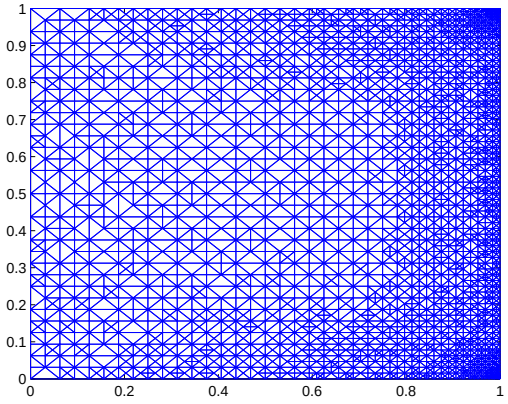
(c) Defect - PCI



(d) Stepwise - PCI



(e) PCI - PQI



(f) MGGHAT

Figure 5.21: First test problem: Final meshes generated by discussed mesh refinement techniques for 2500 points.

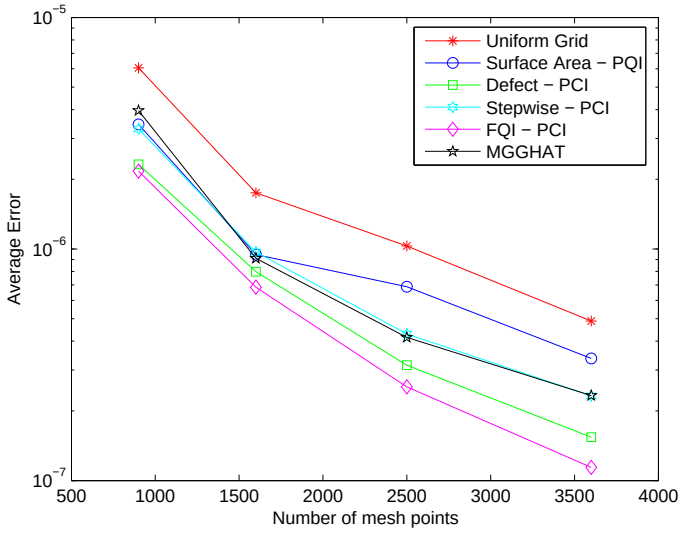


Figure 5.22: First test problem: Average error of all mesh refinement techniques using different number of mesh points.

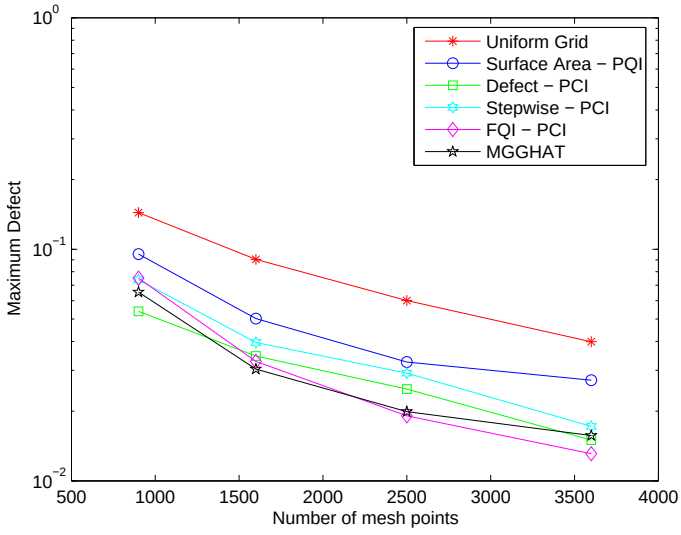


Figure 5.23: First test problem: Maximum defect of all mesh refinement techniques using different number of mesh points.

Figures 5.27 and 5.28 show the final triangulations using all the techniques discussed in section 3.3 for the second test problem, for 900 and 2500 mesh points, respectively. It can be seen that the 'Surface Area' technique creates a refined mesh with a pattern different from what other techniques do for this test problem. Tables 5.14, 5.15 and 5.16 report the average and maximum error in the solution as well as the maximum defect for the second test problem using several mesh selection techniques for 900, 1600, 2500 and 3600 points. In addition, Figures 5.24, 5.25 and 5.26 show the same information in semi-log plots. The errors have been computed over a 100×100 uniform grid inside the domain.

What we can see in the table and figures is:

- The mesh generated by the 'Surface area' technique (Figures 5.27(b) and 5.28(b)) for this problem is very close to the 'Uniform mesh' because the height of the peak is comparable to the size of the mesh elements for this particular problem, and consequently, there is a little difference in the surface areas.

- All other techniques including the one for MGGHAT generate finer mesh elements close to the peak (.5,.117).

- For the average error (Table 5.14 and Figure 5.24), all techniques other than 'Surface Area' generate results 5 to 10 times better than the 'Uniform mesh'.

- For the maximum error (Table 5.15 and Figure 5.25), the 'Defect' and the 'Interpolation Error' techniques appear to work up to 60 times better than the 'Uniform mesh'.

- For the maximum defect (Table 5.16 and Figure 5.26), the 'Defect', the 'Stepwise Error' and the 'Interpolation Error' techniques work 4 to 6 times better than the 'Uniform mesh'.

Method	900		1600		2500		3600	
	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio
Uniform Grid	3.17×10^{-6}	1.00	9.97×10^{-7}	1.00	4.07×10^{-7}	1.00	1.83×10^{-7}	1.00
Surface Area - PQI	1.66×10^{-6}	1.90	6.11×10^{-7}	1.63	2.34×10^{-7}	1.73	1.04×10^{-7}	1.75
Defect - PCI	3.06×10^{-7}	10.35	1.06×10^{-7}	9.40	4.62×10^{-8}	8.80	2.16×10^{-8}	8.47
Stepwise - PCI	6.16×10^{-7}	5.14	1.88×10^{-7}	5.30	7.43×10^{-8}	5.47	3.61×10^{-8}	5.06
FQI - PCI	3.59×10^{-7}	8.83	1.28×10^{-7}	7.78	5.26×10^{-8}	7.73	2.81×10^{-8}	6.51
MGGHAT	3.99×10^{-7}	7.94	1.28×10^{-7}	7.33	5.59×10^{-8}	7.28	2.82×10^{-8}	6.48

Table 5.14: Second test problem: Average error and ratio of improvement of all mesh refinement techniques comparing to a uniform mesh.

Method	900		1600		2500		3600	
	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio
Uniform Grid	2.73×10^{-4}	1.00	8.85×10^{-5}	1.00	3.93×10^{-5}	1.00	1.88×10^{-5}	1.00
Surface Area - PQI	2.01×10^{-4}	1.35	7.05×10^{-5}	1.25	2.39×10^{-5}	1.64	1.42×10^{-5}	1.75
Defect - PCI	6.42×10^{-6}	42.52	2.41×10^{-6}	36.72	7.58×10^{-7}	51.84	3.14×10^{-7}	59.87
Stepwise - PCI	2.01×10^{-5}	13.58	4.25×10^{-6}	20.82	3.93×10^{-6}	10.00	1.81×10^{-6}	10.38
FQI - PCI	4.50×10^{-6}	60.66	1.79×10^{-6}	49.44	7.29×10^{-7}	53.90	4.81×10^{-7}	39.08
MGGHAT	2.51×10^{-5}	10.87	4.82×10^{-6}	18.36	2.37×10^{-6}	16.58	6.20×10^{-7}	30.32

Table 5.15: Second test problem: Maximum error and ratio of improvement of all mesh refinement techniques comparing to a uniform mesh.

Method	900		1600		2500		3600	
	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio	Avg. Error	Ratio
Uniform Grid	6.74×10^{-2}	1.00	3.72×10^{-2}	1.00	2.73×10^{-2}	1.00	1.63×10^{-2}	1.00
Surface Area - PQI	4.87×10^{-2}	1.38	2.84×10^{-2}	1.31	1.80×10^{-2}	1.51	1.20×10^{-2}	1.35
Defect - PCI	1.36×10^{-2}	4.95	7.56×10^{-3}	4.93	4.63×10^{-3}	5.89	3.33×10^{-3}	4.89
Stepwise - PCI	1.65×10^{-2}	4.08	9.07×10^{-3}	4.11	5.73×10^{-3}	4.76	4.21×10^{-3}	3.87
FQI - PCI	1.39×10^{-2}	4.84	7.83×10^{-3}	4.76	5.01×10^{-3}	5.44	3.51×10^{-3}	4.64
MGGHAT	5.07×10^{-2}	1.32	2.78×10^{-2}	1.34	1.20×10^{-2}	2.27	6.00×10^{-3}	2.71

Table 5.16: Second test problem: Maximum defect and ratio of improvement of all mesh refinement techniques comparing to a uniform mesh.

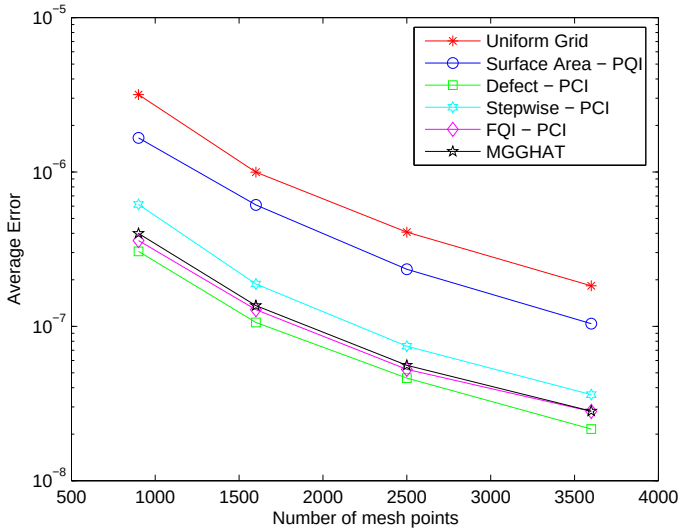


Figure 5.24: Second test problem: Average error of all mesh refinement techniques using different number of mesh points.

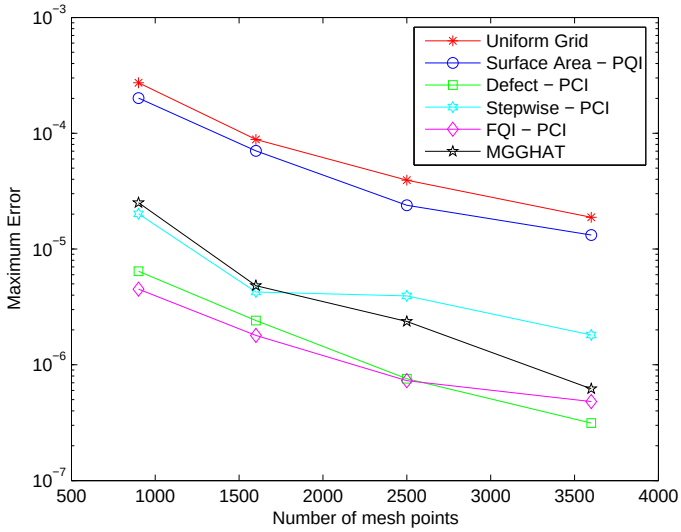


Figure 5.25: Second test problem: Maximum error of all mesh refinement techniques using different number of mesh points.

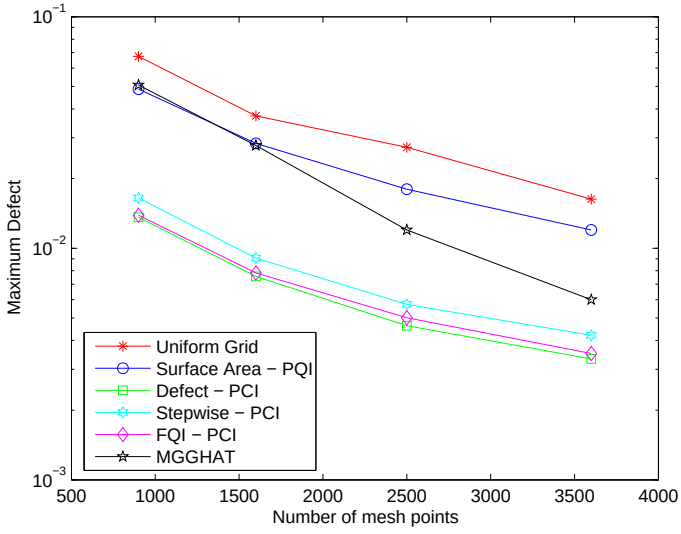
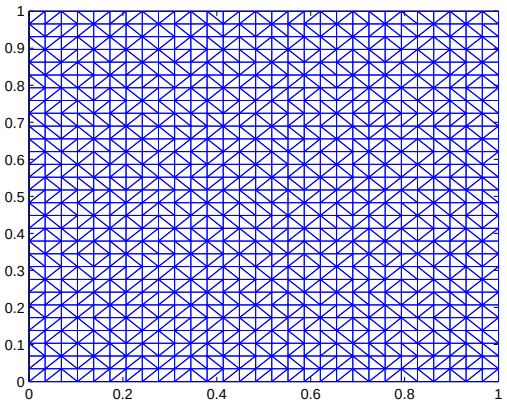
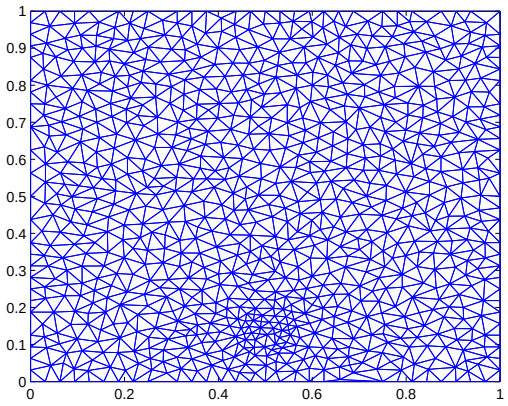


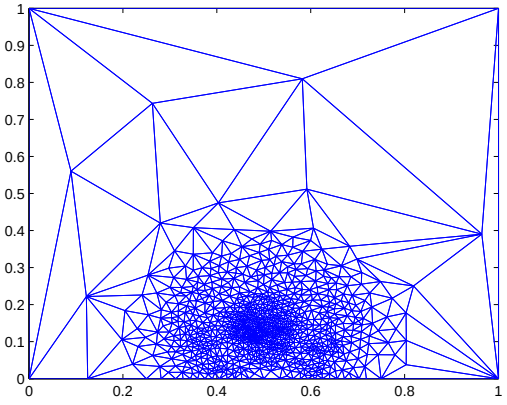
Figure 5.26: Second test problem: Maximum defect of all mesh refinement techniques using different number of mesh points.



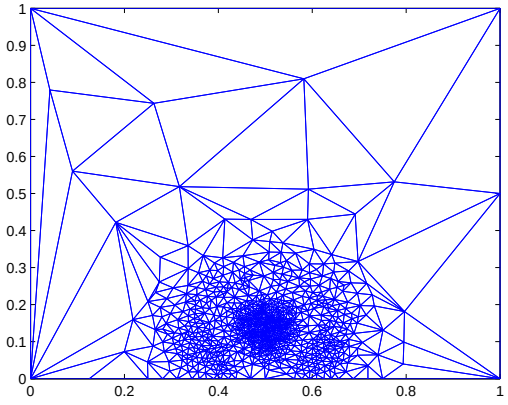
(a) Uniform Grid



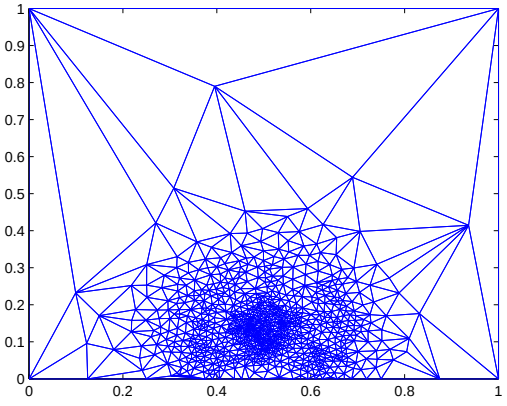
(b) Surface Area - PQI



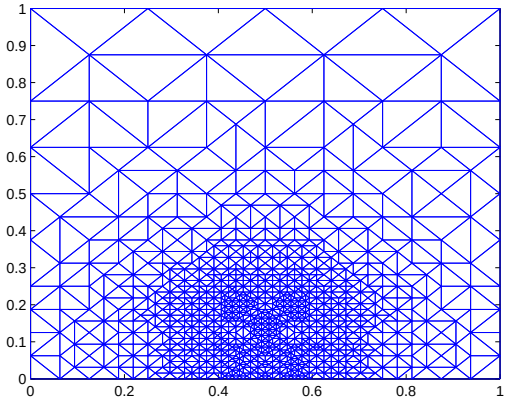
(c) Defect - PCI



(d) Stepwise - PCI

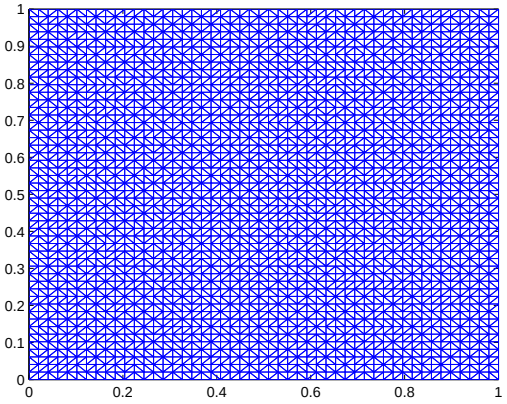


(e) PCI - PQI

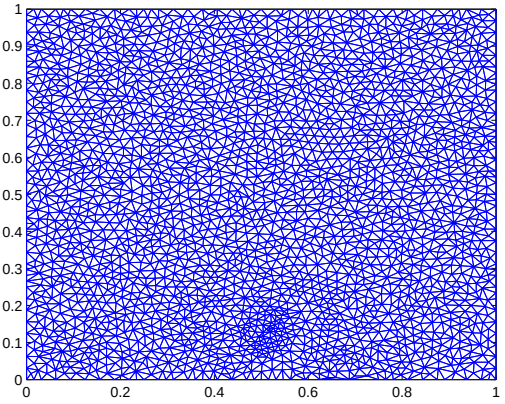


(f) MGGHAT

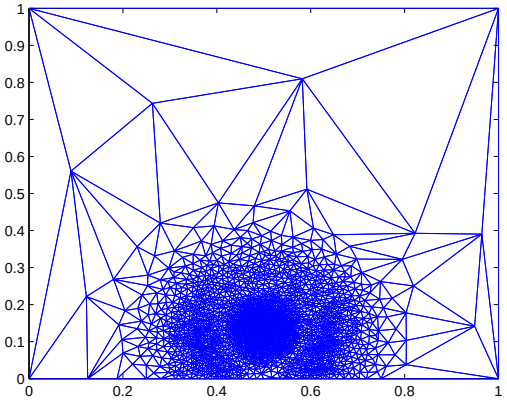
Figure 5.27: Second test problem: Final meshes generated by discussed mesh refinement techniques for 900 points.



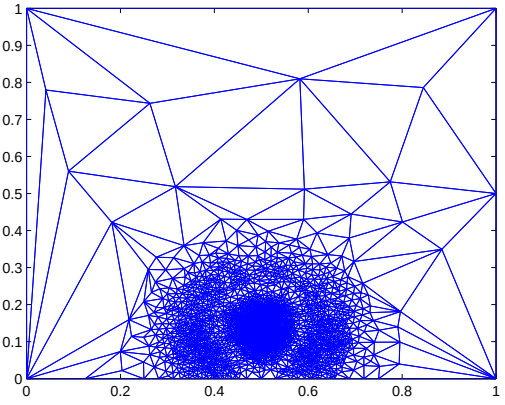
(a) Uniform Grid



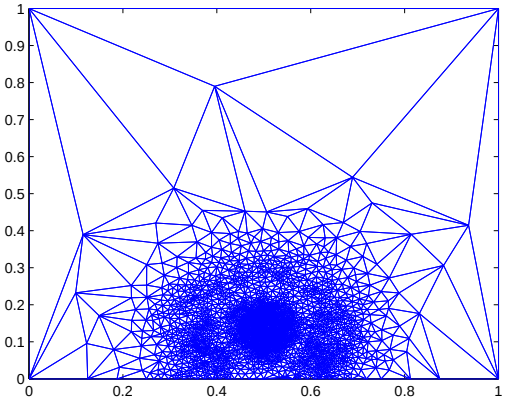
(b) Surface Area - PQI



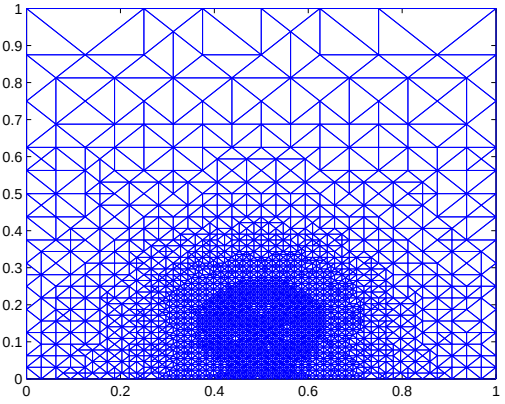
(c) Defect - PCI



(d) Stepwise - PCI



(e) PCI - PQI



(f) MGGHAT

Figure 5.28: Second test problem: Final meshes generated by discussed mesh refinement techniques for 2500 points.

Comparing the final meshes generated by DEI-based techniques with the one generated by the MGGHAT internal mesh refinement technique in Figures 5.27 and 5.28, one may wonder where the problem is symmetric with respect to $x = 0.5$, why our techniques do not generate a symmetric final mesh while MGGHAT does. There are three main reasons for this behavior. First, the initial mesh used for our techniques was not symmetric which could generally lead to a non-symmetric final mesh. Secondly, in order to approximate a measure of the error in the mesh selection step ('defect', 'surface area' and so on), we use a set of random points which could be different for mesh elements. And thirdly, as discussed earlier, unlike the PCI in which we use a single fixed point to generate the piecewise polynomial, we use some random collocation points in order to define the PQI and the FQI. Apparently, if we address these three issues, we will end up with a symmetric final mesh for our techniques as well. Figure 5.29 shows a final mesh with 200 points generated by 'Defect-PCI' approach, starting with a symmetric mesh and using a non-random approach to approximate a measure of the error in the mesh selection step.

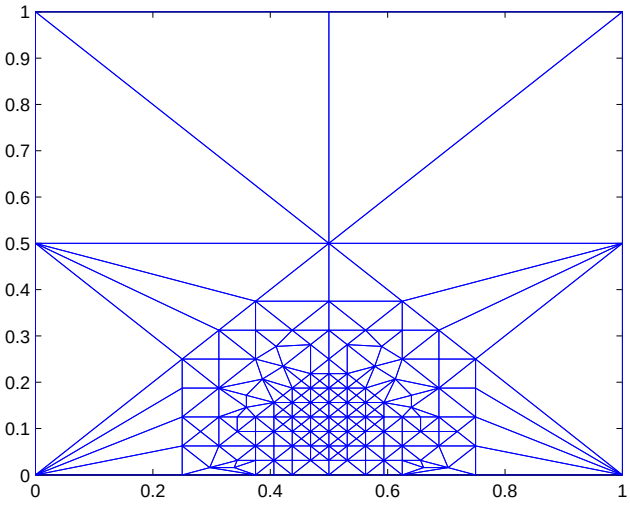


Figure 5.29: Second test problem: A symmetric mesh generated by 'Defect-PCI' approach, starting with a symmetric mesh and using a non-random approach.

5.5 Error Estimation

The implementation of the error estimation based on the companion equation has been done in Fortran. We have used INTCOL and HERMCOL as the underlying PDE method to solve both the original and the companion equations [25]. Since these subroutines are essentially written for a tensor product rectangular mesh and rectangular domain, in order to implement our error estimation approach, we have implemented the DEI-based bi-cubic polynomials for a rectangular mesh.

Figure 5.30 shows the results of applying the companion equation to estimate the error and to improve the solution for the third test problem. In addition, figure 5.31 illustrates the same results for U_x. In these figures, we report the true error, $tru - er$, on the left hand side, the estimated error, $est - er$, in the middle, and the error in the improved solution $U(x, y) + E(x, y)$, $imp - er$, on the right hand side. We can see that

- Comparing the left hand side figures with the middle ones, one can see how well we are able to estimate the error by solving the companion equation.

- Comparing the left hand side figures with the right hand side ones shows that we are able to improve the solution and decrease the maximum error by adding the solution of the companion equation to the original solution.

Table 5.17 reports the average and maximum error in U, U_x, and U_y for the built-in interpolant, for our PCI $U(x, y)$ denoted $tru - er$, and for our improved interpolant $U(x, y) + E(x, y)$, denoted $imp - er$. Each of these statistics were determined over a 100×100 mesh for the third test problem. The ratios of improvement have been also included in the table. The ratio on each line shows the rate of improvement on the next line comparing to the line before. One can observe that

- The DEI-based piecewise polynomial is more accurate than the built-in interpolant.

- The maximum error decreases reasonably after adding the solution of the companion equation.

- We are unable to improve the average accuracy of the solution for this test problem.

mesh size		U		U_x		U_y	
		Avg	Max	Avg	Max	Avg	Max
20×20	Built-In	7.65×10^{-7}	6.70×10^{-5}	1.93×10^{-5}	2.07×10^{-3}	1.93×10^{-5}	2.07×10^{-3}
	Ratio	2.88	2.23	1.78	1.46	1.78	1.46
	$tru - er$	2.65×10^{-7}	3.00×10^{-5}	1.07×10^{-5}	1.42×10^{-3}	1.07×10^{-5}	1.42×10^{-3}
	Ratio	0.92	3.20	1.79	3.03	1.79	3.03
	$imp - er$	2.87×10^{-7}	9.36×10^{-6}	6.02×10^{-6}	4.67×10^{-4}	6.02×10^{-6}	4.67×10^{-4}
40×40	Built-In	4.99×10^{-8}	1.28×10^{-5}	2.65×10^{-6}	7.91×10^{-4}	2.65×10^{-6}	7.91×10^{-4}
	Ratio	2.83	1.43	1.86	1.53	1.86	1.53
	$tru - er$	1.76×10^{-8}	8.91×10^{-6}	1.42×10^{-6}	5.15×10^{-4}	1.42×10^{-6}	5.15×10^{-4}
	Ratio	0.88	5.60	1.64	4.49	1.64	4.49
	$imp - er$	1.99×10^{-8}	1.58×10^{-6}	8.61×10^{-7}	1.14×10^{-4}	8.61×10^{-7}	1.14×10^{-4}
80×80	Built-In	3.99×10^{-9}	3.81×10^{-6}	2.97×10^{-7}	1.20×10^{-4}	2.96×10^{-7}	1.20×10^{-4}
	Ratio	4.11	4.87	1.48	1.18	1.48	1.18
	$tru - er$	9.70×10^{-10}	7.82×10^{-7}	1.99×10^{-7}	1.01×10^{-4}	1.99×10^{-7}	1.01×10^{-4}
	Ratio	0.71	4.26	1.60	3.87	1.60	3.87
	$imp - er$	1.35×10^{-9}	1.83×10^{-7}	1.24×10^{-7}	2.61×10^{-5}	1.24×10^{-7}	2.61×10^{-5}

Table 5.17: The average and maximum error in U, U_x, and U_y evaluated on a 100×100 mesh for the third test problem.

As discussed in section 4.3, solving the companion equation can be a time-consuming process because a large number of local PDEs are solved and the observation that there is a considerable amount of overhead involved in 'setting up' each local problem. Table 5.18

shows the required time for solving the original equation (part 1) and also for solving the companion equation (part 2) for the first test problem. Obviously, solving the companion equation needs much more CPU time which can be decreased by considering parallel implementation.

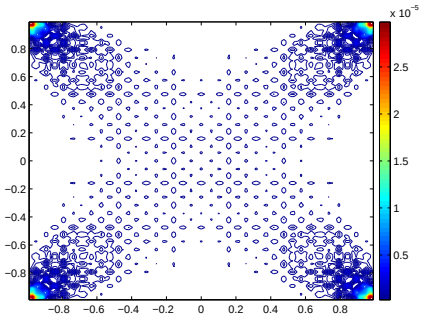
	20×20	40×40	80×80
Part 1	0.21	1.31	8.14
Part 2	3.45	20.11	125.61
Total	3.66	21.42	133.75

Table 5.18: CPU Time (in seconds) for different mesh sizes for the first test problem with $\beta = 20$.

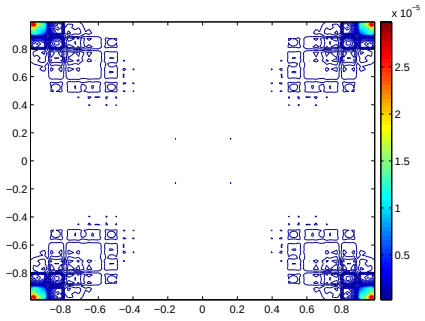
Table 5.19 reports the speed-up we obtain by using up to 16 processors in order to approximate the solution of the first test problem using the companion equation approach for a mesh of size 48×48. Furthermore, Figure 5.32 shows the required time for each part and total required time and Figure 5.33 shows a graphical representation of the speed-up we obtained using up to 16 processors. As expected, the parallel implementation is scalable and we can speed the execution of the second part by a factor close to the number of processors. It also can be seen, using 16 processors decreases the execution time of the second part to 2.25 seconds which is now comparable to the execution time of the first part (1.76 seconds).

Number of processors	1	2	3	4	6	8	12	16
Part 1	1.76	1.76	1.76	1.76	1.76	1.76	1.76	1.76
Part 2	25.36	12.69	8.49	6.50	4.58	3.49	2.41	2.25
Part 2 Speed-up	1.00	1.99	2.98	3.90	5.53	7.26	10.52	11.27
Total	27.12	14.45	10.25	8.26	6.34	5.25	4.17	4.01
Total Speed-up	1.00	1.87	2.64	3.28	4.27	5.16	6.50	6.76

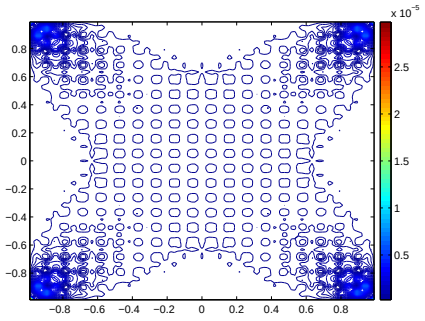
Table 5.19: CPU Time (in seconds) and speed-up obtained using 2 to 16 processors for a mesh of size 48×48 for the first test problem with $\beta = 20$.



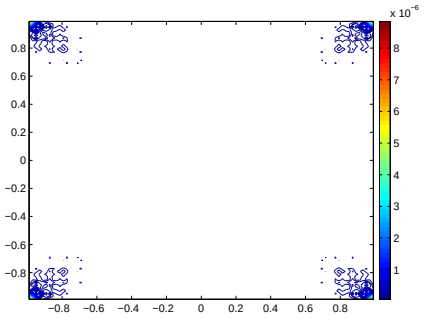
(a) $tru - er$ (20×20)



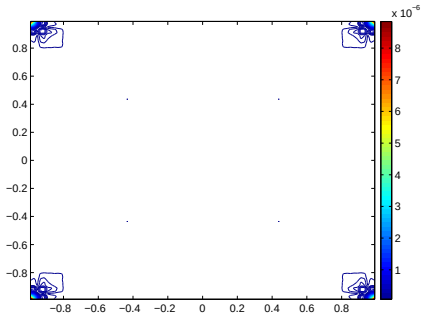
(b) $est - er$ (20×20)



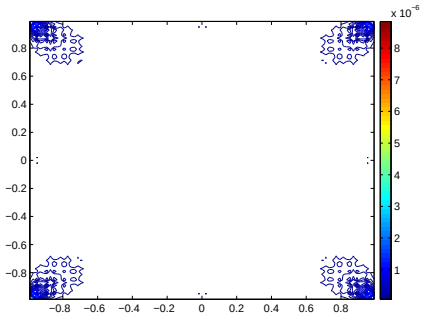
(c) $imp - er$ (20×20)



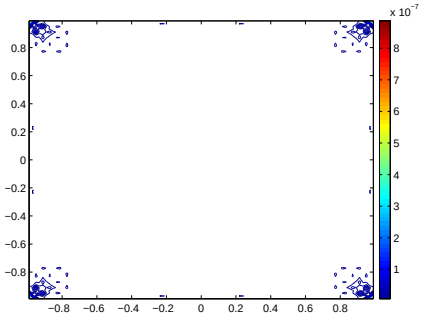
(d) $tru - er$ (40×40)



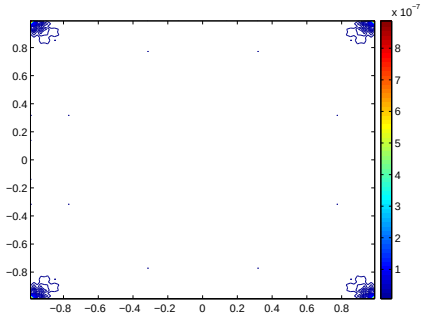
(e) $est - er$ (40×40)



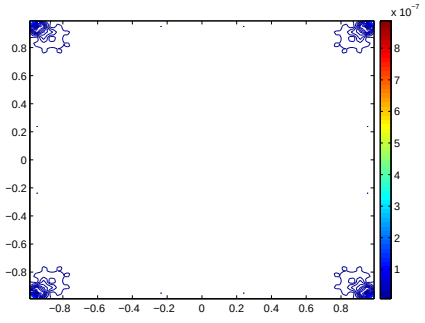
(f) $imp - er$ (40×40)



(g) $tru - er$ (80×80)

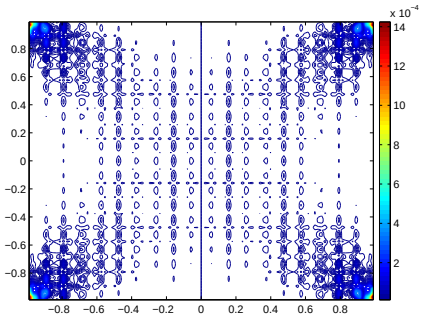


(h) $est - er$ (80×80)

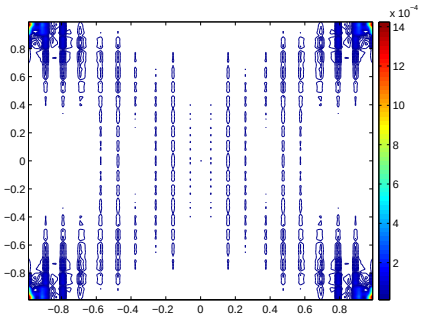


(i) $imp - er$ (80×80)

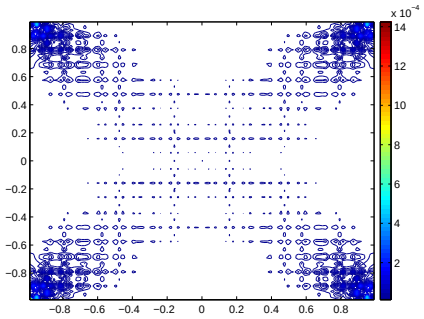
Figure 5.30: Contour plots of the errors, $tru - er$, $est - er$, and $imp - er$ in U for the third test problem.



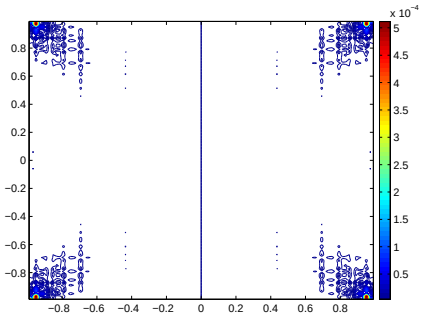
(a) $tru - er$ (20×20)



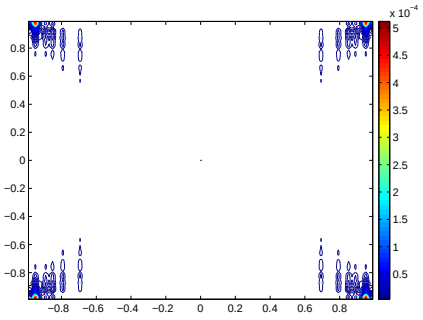
(b) $est - er$ (20×20)



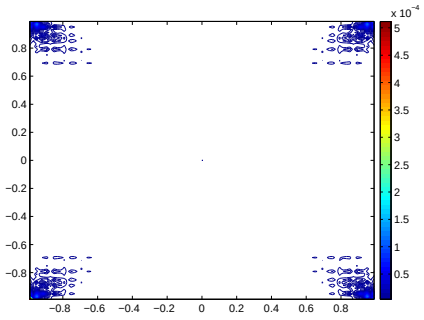
(c) $imp - er$ (20×20)



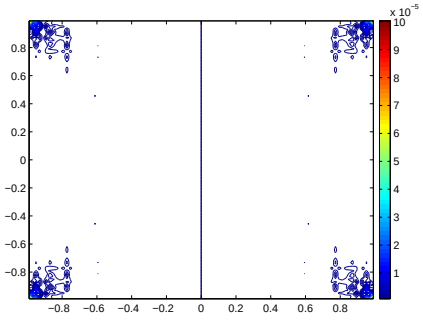
(d) $tru - er$ (40×40)



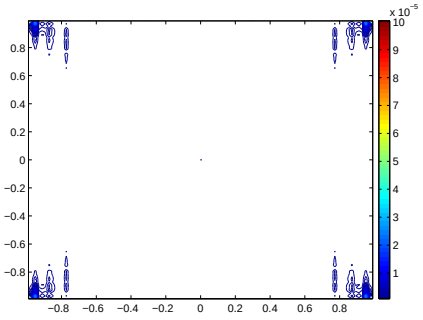
(e) $est - er$ (40×40)



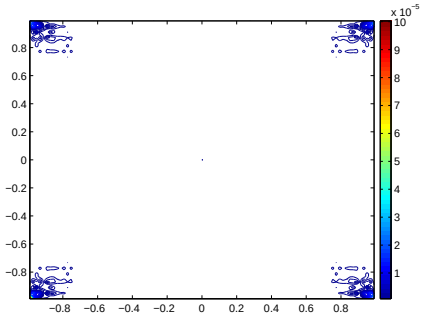
(f) $imp - er$ (40×40)



(g) $tru - er$ (80×80)



(h) $est - er$ (80×80)



(i) $imp - er$ (80×80)

Figure 5.31: Contour plots of the errors, $tru - er$, $est - er$, and $imp - er$ in U_x for the third test problem.

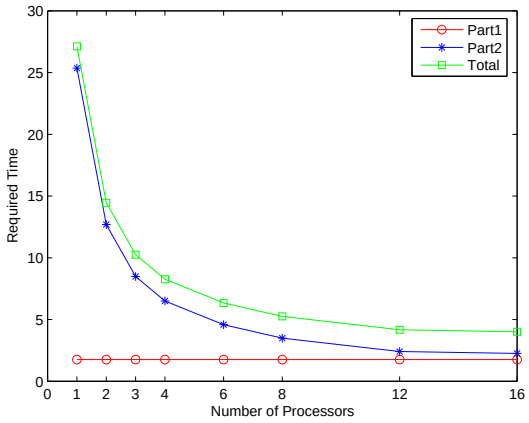


Figure 5.32: CPU Time (in seconds) for both parts and total applying 2 to 16 processors for a mesh of size 48×48 for the first test problem with $\beta = 20$.

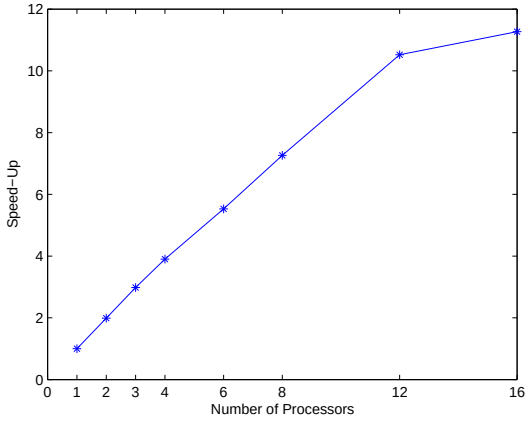


Figure 5.33: Speed-up obtained using 2 to 16 processors for a mesh of size 48×48 for the first test problem with $\beta = 20$.

Chapter 6

Conclusions

6.1 Summary

We have investigated using the DEI to accurately approximate the solution of PDEs at off-mesh points. We introduced the PCI, a two dimensional piecewise interpolant with scattered data that accurately approximates the solution of two-dimensional PDEs at off-mesh points. The main advantage of the PCI is that it generates a continuous representation. We have also investigated its extension to three dimensions and investigated three candidate interpolants defined for second-order elliptic PDEs. Our results indicates that a pure tri-cubic interpolant is particularly effective. We have also showed that, although it is the best in terms of realistic and non distracting visualizations, it can still suffer from discontinuities along neighbor elements' faces.

In addition, we have carried out an investigation of an important application of our interpolation techniques. The application is scientific visualization where standard renderers require that the function to be displayed be known on a very fine mesh. We have presented three fast algorithms for drawing contour plots of the approximate solution of two-dimensional second-order elliptic PDEs. We have also directly extended one of the approaches to three dimensions and described an algorithm for drawing three-dimensional

level sets for the solution of three-dimensional second-order elliptic PDEs. We then presented numerical results that demonstrate the scalable nature of this algorithm.

As another application, we focused on adaptive mesh refinement for two dimensional PDEs and introduced some generic algorithms based on the use of DEI-based interpolants. Each algorithm has two main steps; the mesh selection and the mesh refinement. For the mesh selection step, we presented four 'generic' strategies based on properties of the problem alone rather than a special-purpose strategy designed for a specific PDE method. For the mesh refinement step, we discussed advantages and disadvantages of well-known mesh refinement algorithms and presented a hybrid method using local refinement and local reconnection approaches.

We also introduced a global error estimator by introducing a companion equation based on the original PDE and a DEI. A single companion equation is then solved for each mesh element of the coarse mesh. In addition, we discussed how these companion equations can be solved in parallel. The results show that the parallel approach is able to decrease the required time for solving the set of local companion equations close to the time required to solve the original equation itself.

6.2 Future Work

Some interesting directions for future work might be

- We extended one of our two-dimensional contouring algorithms, the Intercept method, to three dimensions. The extension of other two methods to three dimensions is an area for future investigation.

- Our focus, in all areas we investigated, was restricted to a second order elliptic PDE. A generalization to other types of PDEs is another area for future work.

- In our global error estimator, we considered simple Dirichlet boundary conditions

$e(x, y) = 0$ for each local problem. Finding more realistic boundary conditions should result in a better estimation of the error.

- Direct extension of our AMR algorithm to three dimensions is another direction for future study.

Appendix A

Adaptive Mesh Refinement History

Up to now, several approaches have been used to improve the quality of unstructured meshes. These approaches can be classified into three basic categories:

- **Local Refinement:** Point insertion or deletion to refine or coarsen a mesh [5] [27] [28] [32] [36].
- **Local Reconnection:** Change mesh topology by face or edge swapping for a given set of vertices [11] [18] [26] [41].
- **Local Mesh Smoothing:** Relocate mesh points to improve mesh quality without changing mesh topology [6] [8] [15] [16] [18].

In the following, an overview of some specific strategies associated with each category will be presented.

A.1 Local Refinement

Rather than using a uniform mesh with grid points evenly spaced in a domain, adaptive mesh refinement techniques place more grid points in areas where the error in the solution is likely to be largest. The mesh is adaptively refined and/or unrefined during the computation according to some measure of 'local error' or 'local behavior' of the approximate solution on the domain. Typically, one begins with an initial mesh conforming

to a particular geometry. This mesh is selectively refined to construct a modified mesh that is used to obtain an 'improved' approximate solution which hopefully will satisfy a certain error tolerance. A 'monitor function' has to be defined as a measure of the 'error' or 'local behavior' of a function over a mesh element. A mesh refinement strategy then refers to an attempt to 'equidistribute' this measure over all elements of a mesh by introducing new mesh points, removing some and/or 'remeshing' the current mesh. A typical adaptive mesh refinement algorithm is shown in Figure A.1.

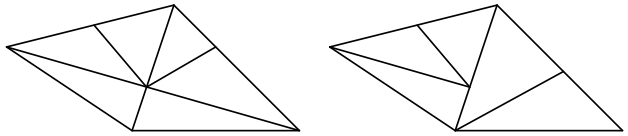
$k = 0$
Solve the PDE on the mesh, T_k
Estimate some 'measure' associated with each triangle, t_i, of T_k
while the measure associated with a triangle is larger than the given tolerance **do**
  Determine a set of triangles, $S_k \subset T_k$, to refine based on the measure estimate
  Refine the triangles in S_k, and any other triangles necessary to form T_{k+1}
  Solve the PDE on this new mesh T_{k+1}
  Estimate the measure on each triangle of T_{k+1}
  $k = k + 1$
end do

Figure A.1: A typical adaptive mesh refinement algorithm.

It begins by assuming that an initial coarse mesh is given by triangulation T_0 consistent with the geometry of the problem domain. The focus is on the step in the algorithm where the current mesh T_k is adaptively refined (the step in bold face in Figure A.1). Note that the refinement of the mesh must maintain several important properties, given that finite element approximations are to be determined. Three such properties are:

- **Conforming Mesh:** A mesh is *conforming* if the intersection of any two triangles in T_k is a single vertex, a line segment connecting two vertices, or the empty set.

An edge of a triangle is called $\frac{1}{s+1}$-*nonconforming* if it has $s > 0$ vertices between its two endpoints [28]. A triangle is called *conforming* if none of its edges are $\frac{1}{s+1}$-*nonconforming* for any $s > 0$. Figure A.2 shows examples of *conforming* and *nonconforming* meshes.



(a) A conforming mesh (b) A $\frac{1}{2}$-nonconforming mesh

Figure A.2: Examples of conforming and nonconforming meshes.

- **Graded Mesh:** A mesh is *graded* (or *smooth*) if adjacent triangles do not differ dramatically in area. A nonsmooth mesh could result in the finite element approximation being quite far from the real solution [27].

- **Bounded (Minimum/Maximum) Angle Mesh:** A mesh is *bounded-angle* if all angles of the triangles in the mesh are bounded away from 0 and π. The latter condition is necessary because the discretization error in a finite element approximation has been shown to grow as the maximum angle approaches π [3]. We would also like to avoid small angles because the condition number of the matrices arising from mesh elements has been shown to grow as $O(\frac{1}{\theta_{min}})$, where θ_{min} is the smallest angle in the mesh [19].

A number of mesh refinement algorithms have been shown to maintain some or all mesh properties given above. In the following, we briefly review the four most widely known of these refinement algorithms.

A.1.1 Mid-Point (Centroid) Insertion Algorithm

The most obvious means of dividing a triangle to maintain a conforming mesh is to place a nodal point at the center of the triangle and connect it to the three existing nodal points [27]. As shown in Figure A.3, this process creates three new triangles. Unfortunately, repeated refinement of triangles in this way clearly results in angles that tend toward 0 and π (violating the third property). It also can result in ungraded meshes as illustrated on the rightmost mesh of Figure A.3.

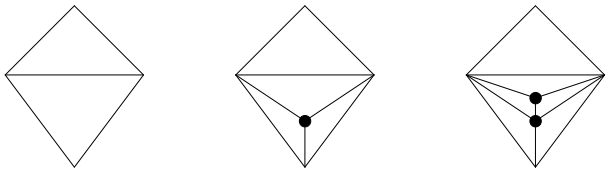


Figure A.3: The mid-point insertion algorithm preserves the conforming property but can violate graded and bounded-angle properties.

A.1.2 Bisection Algorithm

Bisection divides the triangle area exactly in half, and the bisected angle is also halved. This algorithm will result in a nonconforming mesh, as shown in Figure A.4. However this difficulty can be avoided by propagating the refinement to the 'newly introduced' nonconforming triangles. Rivara [36] uses bisection of triangles across the longest edge (dividing the largest angle) and selective divisions across smaller edges. Moreover, if triangles are bisected only across their longest edge, one can bound the maximum and minimum angles of resulting triangles independently of the number of times the resulting triangles are bisected. If a triangle and its descendants are repeatedly bisected across their longest edges, the smallest resulting angle is bounded by at worst one-half the smallest angle in the original triangle [37].

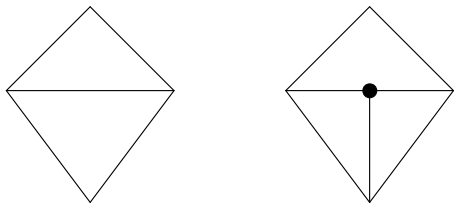


Figure A.4: The bisection algorithm violates conforming property.

Rivara has described an effective algorithm for mesh refinement based on bisection in [36]. The algorithm assumes that an initial set of triangles in T_k have been marked for refinement based on some associated error estimates. As triangles become nonconforming, they are also marked for refinement. The algorithm, given in Figure A.5, continues until a *conforming* mesh, T_{k+1}, has been constructed. Rivara shows that this algorithm will terminate; however no particular bound exists for L, the number of times the while loop is executed. Figure A.6 shows a worst-case example of refinement propagation for which L is $O(n)$, where n is the number of triangles in T_k. As mentioned in [27], in practice L is usually a small constant number independent of n.

Let S_0 be the set of marked triangles

$i = 0$

While (S_i is not empty) **do**

  Bisect triangles in S_i across their longest edge

  Let S_{i+1} be the set of all remaining nonconforming triangles

  $i = i + 1$

end do

Figure A.5: The longest edge bisection algorithm.

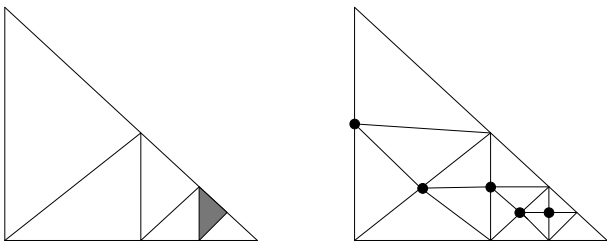


Figure A.6: A worst-case example of propagation of refinement based on Rivara's algorithm.

Rivara has described variants of this algorithm, including one in which *simple bisection* is combined with bisection across the longest edge to reduce L [36]. *Simple bisection* means bisection across an edge that may not be the longest. In this algorithm, a triangle is first bisected across its longest edge. If either of the two new triangles become nonconforming, as the result of bisection of a neighbor, they are bisected across the nonconforming edge. The algorithm yields the same bounds for angles as the longest edge algorithm.

Stamm et al. have developed a modified longest side bisection algorithm to increase the lower bound on the smallest angle without increasing the total number of final triangles [42]. Assuming α as the smallest angle in the original triangulation, they showed that with a small modification to the bisection algorithm, the lower bound on the smallest angle can be increased to $\frac{2}{3}\alpha$ ($\frac{\alpha}{2}$ for the original bisection algorithm).

A.1.3 Regular Refinement Algorithm

A third means of triangle division is regular refinement where the midpoints of the edges are connected forming four new triangles [27]. As illustrated in Figure A.7, this approach can result in a nonconforming mesh. However, a conforming mesh can be obtained by temporarily refining triangles with one nonconforming edge through bisection [5]. Triangles initially marked for refinement are refined by using regular refinement. As refinement propagates, any triangle with at least two nonconforming edges is also regularly refined.

When only triangles with one nonconforming edge remain, the remaining nonconforming edges are temporarily bisected. Prior to the next refinement, these bisected triangles are merged. This algorithm has been used in the software package PLTMG [4]. Note that no refined mesh angle can be less than half the smallest initial mesh angle since the four triangles resulting from regular refinement are all similar to original triangle.

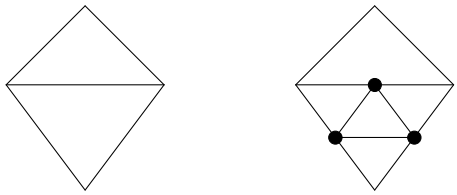


Figure A.7: The regular refinement algorithm violates conforming property.

A.1.4 Newest-Node Algorithm

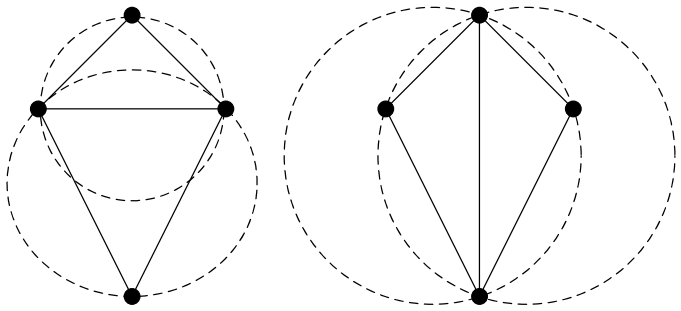
The newest-node algorithm of Sewell [39] is also based on bisection, but without the restriction on bisecting the longest edge. In this algorithm, a triangle is always bisected by using its newest node. Unlike bisection and regular refinement algorithms, it avoids propagations by refining triangles only in pairs. However, because of the pair restriction, it is possible that a triangle may never be selected to be refined. A modified algorithm proposed by Mitchell [30] eliminates this deficiency by ensuring that every triangle is one of a pair of triangles that can be refined. Unfortunately, this modification requires a recursive refinement of triangles adjacent to unrefined triangles. This refinement results a propagation similar to the bisection and regular refinement algorithms.

Mitchell compared these three methods in a series of numerical experiments and found that it was difficult to choose a consistently superior algorithm [30]. In addition, he found that all three algorithms were superior to using uniform refinement except on smooth problems.

It is worth noticing that in [28] a parallel algorithm for adaptive mesh refinement based on local refinement approach is presented that is suitable for implementation on distributed-memory parallel computers.

A.2 Local Reconnection

Local mesh reconnection (reconfiguration) techniques hold the mesh points fixed but change the connectivity of part of a mesh to improve mesh quality (That is the locations of the mesh points do not change but the 'triangulation' does). These techniques are widely used and one of the best known examples is edge flipping applied to a two-dimensional triangular mesh to construct a Delaunay triangulation. Among all triangulations of a vertex set, a Delaunay triangulation maximizes the minimum angle in the triangulation [41]. Consider four arbitrary points in a plane. As shown in Figure A.8, there are only two possible triangulations, one is Delaunay and the other is not (A special case is when the fourth point lies on the circumcircle of the first three points makes both triangulations Delaunay). As illustrated in Figure A.8, in a Delaunay triangulation, the circumcircle of each triangle contains no other vertices.



(a) A Delaunay triangulation (b) A non-Delaunay triangulation

Figure A.8: Examples of two possible triangulations of four points.

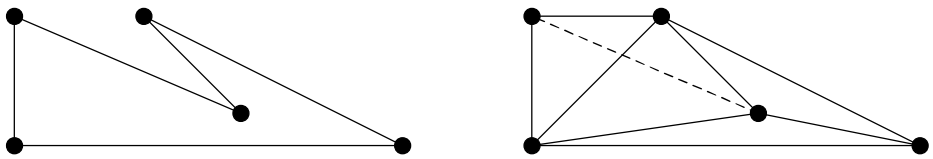


Figure A.9: The Delaunay triangulation of a set of vertices does not necessarily solve the mesh generation problem, because it may contain poor quality triangles and may omit some of the domain boundaries.

A.2.1 Edge Flipping in Two Dimensions

The edge flipping algorithm begins with an arbitrary triangulation and searches for an edge that is not Delaunay. The definition of a Delaunay edge is exactly the same as the definition of a Delaunay triangle. That is, its circumcircle does not enclose either of the vertices opposite the edge in the two triangles that contains the edge. Note that all edges on the boundary (convex hull) of the triangulation are considered to be Delaunay. An edge is not flippable when the containing quadrilateral is not convex.

There are a wide variety of measures for the quality of a triangular element, the most obvious being the smallest and largest angles associated with the element. In [29] Miller et al. have shown that the most natural and elegant measure for analyzing a Delaunay refinement algorithm is the *circumradius-to-shortest edge ratio* of an element. Fortunately, this ratio is the metric that is naturally optimized by Delaunay triangulation. In two dimensions, this ratio $(\frac{r}{d})$ is related to the smallest angle α by the formula $\frac{r}{d} = \frac{1}{2\sin(\alpha)}$. The smaller a triangle's ratio, the larger its smallest angle [41].

A.3 Local Mesh Smoothing

Mesh smoothing algorithms try to improve the mesh quality by adjusting the vertex positions without changing the mesh topology. Local mesh smoothing algorithms adjust the location of a single mesh point by using only the information at incident vertices rather than global information in the mesh. A typical vertex, v, and its adjacent set are shown in Figure A.10. The vertices in the adjacent set are shown as solid circles in the figure. As the vertex v is moved, only the quality of the elements incident on v, shown as shaded triangles in the figure, are changed. Vertices shown as unfilled circles are not adjacent to v and the quality of triangles that contains these vertices are not affected by a change in the location of v. Since more than one sweep through the mesh might be required to improve the overall mesh quality, it is important that each individual adjustment be inexpensive to apply.

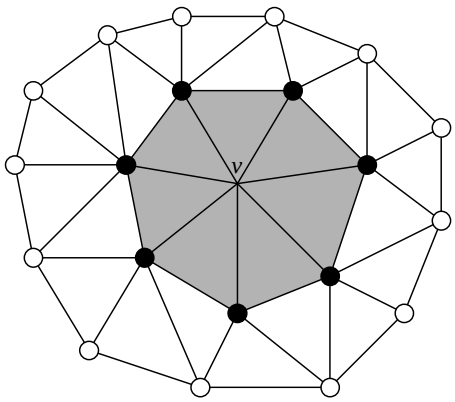


Figure A.10: A vertex v and the adjacent triangles whose quality is affected by a change in the position of v.

A significant amount of work has been done in the area of local mesh smoothing. In the following, an overview of some local mesh smoothing algorithms including Laplacian smoothing and optimization-based smoothing will be presented.

A.3.1 Laplacian Smoothing

Laplacian smoothing is by far the most common local smoothing technique. This approach, in its simplest form, replaces the position of a vertex v by the average of its neighbors's positions. The method is computationally inexpensive, but it does not guarantee improvement in element quality. In fact, the method can produce an invalid mesh containing elements that have negative volume. Figure A.11 shows how Laplacian smoothing can generate an invalid mesh. The unfilled square, v', marks the average of the positions of the vertices adjacent to v. As can be seen the area of the filled elements in the right of Figure A.11 is negative.

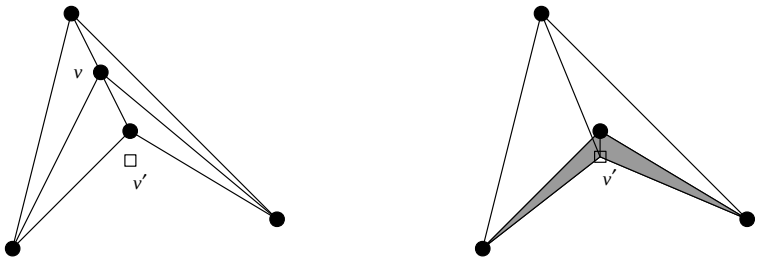


Figure A.11: A set of triangles for which Laplacian smoothing results in an invalid mesh.

Many researchers have used and extended the capabilities of Laplacian smoothing. Some variations of Laplacian smoothing include:

- Weighting the contribution of each neighboring node in the average function by edge length, element area, or other similar criteria.

- Constraining the node movement to avoid the creation of elements with negative areas.

A.3.2 Optimization-based Smoothing

A newer form of smoothing, that is receiving more attention lately, is optimization-based smoothing. Instead of moving nodes based on an heuristic algorithm, as is done in Laplacian smoothing, the nodes are moved in an attempt to minimize a given distortion metric. While optimization-based smoothing is more expensive than Laplacian smoothing, it gives better results especially for concave regions in the geometry. Several such techniques have been proposed and we briefly review some of them now. The methods differ primarily in the optimization procedure used or the quantity that is optimized.

In [4] Bank introduced a smoothing procedure for a two-dimensional triangular mesh that uses a measure of the 'quality of the shape' of each element defined by

$$q(t) = \frac{4\sqrt{3}A}{\sum_{i=1}^{3} l_i^2},$$

where A is the area of the triangular element and l_i is the length of edge i. The maximum value for $q(t)$ corresponds to an equilateral triangle (where $q(t) = 1$). Each local submesh is improved by using a line search procedure. The search direction is determined by the line connecting the current position of v to the position that results in the worst element becoming equilateral. The line search terminates when at least one other affected element's shape quality value equals that of the improving element. One variant of this technique attempts to directly compute the new location by using the two worst elements in the local submesh. In this case the line search procedure is used only in the cases for which the new position results in a third element, different from the original two worst elements, with the smallest shape measure. Bank also presented an optimization-based smoothing algorithm, specifically designed for adaptively improving finite element triangulations by making use of a posteriori estimates [6].

A similar approach for tetrahedral meshes in three dimensions is described by Shephard and Georges [40]. The measure of quality for each element incident on v is computed

by using the formula

$$q(t) = \kappa \frac{V^4}{(\sum_{i=1}^{4} A_i^2)^3},$$

where V is the volume of the element and A_i is the area of face i. The parameter κ is chosen so that $q(t)$ has a maximum of one (corresponding to an equilateral tetrahedron). A line search similar to that done by Bank is performed, where the search direction is determined by the line connecting the current position of v to the position that improves the worst element in the local submesh to equilateral.

Freitag et al. proposed a method for two and three-dimensional meshes based on the steepest descent optimization technique for nonsmooth functions [17] [15] [18]. The goal of the optimization approach is to determine the position that maximizes the composite function

$$\phi(\mathbf{x}) = \min_{1 \le i \le l} f_i(\mathbf{x}),$$

where the functions f_i are based on various measures of mesh quality such as max-min angles or element aspect ratios and l is the number of functions defined on the local submesh. For example, in two-dimensional triangular meshes, maximizing the minimum angle of a local submesh containing m elements requires $l = 3m - 2$ function evaluations. The search direction at each step is computed by solving a quadratic programming problem that gives the direction of steepest descent from all possible convex linear combinations of the gradients in the active set. The line search subproblem is solved by predicting the points at which the set of active functions will change based on the first order Taylor series approximations of the $f_i(\mathbf{x})$.

Amenta et al. show that the optimization techniques used in [17] [15] are equivalent to the generalized linear programming technique and has an expected linear solution time [2]. The convex level set criterion for solution uniqueness of generalized linear programs can be applied to these smoothing techniques, and they determine the convexity of the level sets for a number of standard mesh quality measures in both two and three dimensions.

A.3.3 Combined Laplacian and Optimization-based Smoothing

Both Shephard and Georges [40] and Freitag and Ollivier-Gooch [17] [18] presented experimental results that demonstrate the effectiveness of combining a variant of Laplacian smoothing with their respective optimization-based procedures. The version of Laplacian smoothing used by Shephard and Georges allows the vertex to move to the centroid of the incident vertices only if the worst element maintains a shape measure $q(t)$ above a fixed limit. Otherwise, the line connecting the centroid and the initial position is bisected, and the bisection point is used as the target position. Freitag and Ollivier-Gooch accept the Laplacian step whenever the local submesh is improved. In both cases, the Laplacian smoothing step is followed by optimization-based smoothing for only the worst elements. Experiments in [17] showed that using optimization-based smoothing when the minimum angle was less than 30° in two dimensions and 15° in three dimensions significantly improves the meshes at a small computational cost.

Bibliography

[1] G. S. Almasi and A. Gottlieb. *Highly parallel computing.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[2] Nina Amenta, Marshall Bern, and David Eppstein. Optimal point placement for mesh smoothing. In *SODA '97: Proceedings of the 8th annual ACM-SIAM symposium on Discrete algorithms*, pages 528–537, Philadelphia, PA, USA, 1997.

[3] I. Babuška and A. K. Aziz. On the angle condition in the finite element method. *SIAM Journal Numerical Analysis*, 13:214–226, 1976.

[4] Randolph E. Bank. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations. Users' Guide 8.0.* SIAM, Philadelphia, PA, 1998.

[5] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser. Refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman et al., editor, *Scientific Computing*, pages 3–17. IMACS/North-Holland, 1983.

[6] Randolph E. Bank and R. Kent Smith. Mesh smoothing using A posteriori error estimates. *SIAM Journal on Numerical Analysis*, 34(3):979–997, 1997.

[7] Emma L. Bradbury and Wayne H. Enright. Fast contouring of solutions to partial differential equations. *ACM Transaction on Mathematical Software*, 29(4):418–439, December 2003.

[8] Scott A. Canann, Joseph R. Tristano, and Matthew L. Staten. An approach to combined laplacian and optimization-based smoothing for triangular, quadrilateral, and quad-dominant meshes. In *Proceedings of the 7th International Meshing Roundtable*, pages 479–494, 1998.

[9] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report TR-89-983, Department of Computer Science, Cornell University, 1989.

[10] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, August 1998.

[11] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the 8th ACM Symposium on Computational Geometry*, pages 43–52, 1992.

[12] Wayne H. Enright. Accurate approximate solution of partial differential equations at off-mesh points. *ACM Transaction on Mathematical Software*, 26(2):274–292, June 2000.

[13] Wayne H. Enright. On the use of 'arc length' and 'defect' for mesh selection for differential equations. *Computing Letters (CoLe)*, 1(2):47–52, 2005.

[14] Wayne H. Enright. Verifying approximate solutions to differential equations. *Journal of Computational and Applied Mathematics*, 185(2):203–211, 2006.

[15] Lori Freitag, Mark Jones, and Paul Plassman. An efficient parallel algorithm for mesh smoothing. In *Proceedings of the 4th International Meshing Roundtable*, pages 47–58. Sandia National Laboratories, 1995.

[16] Lori Freitag, Mark Jones, and Paul Plassman. A parallel algorithm for mesh smoothing. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.

[17] Lori Freitag and Carl Ollivier-Gooch. A comparison of tetrahedral mesh improvement techniques. In *Proceedings of the 5th International Meshing Roundtable*, pages 87–100, 1996.

[18] Lori Freitag and Carl Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40:3979–4002, 1997.

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

[20] Hassan Goldani Moghaddam. Efficient contouring on unstructured meshes. Master's thesis, University of Toronto, 2004.

[21] Hassan Goldani-Moghaddam and Wayne H. Enright. Efficient contouring on unstructured meshes for partial differential equations. *ACM Transactions on Mathematical Software*, 34(4), July 2008. Article 19, 25 pages.

[22] Hassan Goldani Moghaddam and Wayne H. Enright. The PCI: A scattered data interpolant for the solution of partial differential equations. In *Proceedings of International Conference on Adaptive Modeling and Simulation*, ADMOS 2005, Barcelona, Spain, September 2005.

[23] Hassan Goldani Moghaddam and Wayne H. Enright. A scattered data interpolant for the solution of three dimensional PDEs. In *Proceedings of European Conference on Computational Fluid Dynamics*, ECCOMAS CFD 2006, Netherland, September 2006.

[24] Øyvind Hjelle and Morten Dæhlen. *Triangulations and Applications (Mathematics and Visualization)*. Springer-Verlag New York Inc., Secaucus, NJ, USA, 2006.

[25] E. N. Houstis, W. F. Mitchell, and J. R. Rice. Algorithm 638: INTCOL and HERMCOL: Collocation on rectangular domains with bicubic Hermite polynomials. *ACM Transactions on Mathematical Software*, 11(4):416–418, December 1985.

[26] Barry Joe. Construction of three-dimensional improved-quality triangulations using local transformations. *SIAM Journal on Scientific Computing*, 16(6):1292–1307, 1995.

[27] Mark T. Jones and Paul E. Plassmann. Adaptive refinement of unstructured finite-element meshes. *Finite Elem. Anal. Des.*, 25(1-2):41–60, 1997.

[28] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for adaptive mesh refinement. *SIAM Journal on Scientific Computing*, 18(3):686–708, 1997.

[29] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. A Delaunay based numerical method for three dimensions: generation, formulation, and partition. In *Proceedings of the 27th Annual ACM Aymposium on the Theory of Computing*, pages 683–692, May 1995.

[30] William F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Transactions on Mathematical Software*, 15(4):326–347, December 1989.

[31] William F. Mitchell. MGGHAT user's guide version 1.1, 1997.

[32] Carl F. Ollivier-Gooch. Multigrid acceleration of an upwind Euler solver on unstructured meshes. *AIAA Journal*, 33(10):1822–1827, 1995.

[33] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.

[34] Gonzalo A. Ramos and Wayne H. Enright. Interpolation of surfaces over scattered data. *Proceedings of the IASTED International Conference VISUALIZATION, IMAGING, AND IMAGE PROCESSING*, pages 219–224, September 2001.

[35] John R. Rice and Ronald F. Boisvert. *Solving elliptic problems using ELLPACK*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[36] Maria-Cecilia Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613, 1984.

[37] I. G. Rosenberg and F. Stenger. A lower bound on the angles of triangles constructed by bisecting the longest side. *Mathematics of Computation*, 29:390–395, 1975.

[38] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, May 1995.

[39] E. Sewell. A finite element program with automatic user-controlled mesh grading. In *Advances in Computer Methods for Partial Differential Equations III*, pages 8–10, IMACS, New Brunswick, 1979.

[40] M. Shephard and M. Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering*, 32(4):709–749, 1991.

[41] Jonathan Richard Shewchuk. Lecture notes on Delaunay mesh generation, September 1999.

[42] Christoph Stamm, Stephan Eidenbenz, and Renato Pajarola. A modified longest side bisection triangulation. In *Proceedings of the 10th Canadian Conference on Computational Geometry (CCCG'98)*, Montreal, Canada, 1998.