Collocation Methods for Linear Parabolic Partial Differential Equations

Qiang Zheng

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

December 2, 2005

Abstract

Collocation Methods for Linear Parabolic Partial Differential Equations

Qiang Zheng

This thesis presents a new class of collocation methods for the approximate numerical solution of linear parabolic partial differential equations. In the time dimension, the partial derivative with respect to time is replaced by finite differences, to form the implicit Euler method. At each time step, a polynomial approximating the exact solution is calculated for each triangular finite element created by the Rivara algorithm. Polynomials of adjacent finite elements have matching values and matching normal derivatives at a set of discrete points, called "matching points". The method of nested dissection is used to eliminate all variables at the interior matching points of the domain. The maximum error of the solution is of the order of the time step size, which is $O(dt)$, except when dt is sufficiently small. In that case, the maximum error can be very small, depending on the density of the space mesh.

An application based on OpenGL and Motif to visualize the solutions is also described in this thesis. Extensive numerical results, pictures of refined meshes, and $3D$ representations of the solutions are given.

Acknowledgments

I would like to express my sincere appreciation to my supervisor professor, Eusebius J. Doedel, for his guidance, support, papers, and patience, for completing my thesis. Also I would like to thank Chenghai Zhang, for his valuable advice for graphics and for writing this thesis, and Rui Chen, for assistance with the final preparation of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we introduce partial differential equations (PDEs), some classical methods for PDEs, and the more recent collocation methods. The objectives of this thesis are also discussed.

1.1 Partial differential equations

Differential equations come from various subjects in science and technology, such as physics, astronomy, geology, biochemistry, and nuclear technology. Equations which contain derivatives of a function $u(t)$ are called ordinary differential equations (ODEs). Sometimes, the function u can be a vector, in which case the equations form a system. Equations which contain partial derivatives of the function $u(x_1, x_2, \cdots, x_n, t)$ are called partial differential equations (PDEs). Here, u is a function of x_1, x_2, $\cdots$, x_n, t, which are $n+1$ independent variables. In many applications, t denotes the time variable and $X = (x_1, x_2, \cdots, x_n)^* \in R^N$ denotes the space variables. For the $2D$ and $3D$ cases, the notation $X = (x, y)^*$ and $X = (x, y, z)^*$, will be used, respectively. Sometimes, the equations do not involve the variable t, so t is omitted.

Generally, it is very difficult or even impossible to obtain exact solutions of differential equations, since they are very complicated. Thus people usually try to find methods to obtain approximate numerical solutions. With the help of modern computer technology, numerical solutions can often be calculated quickly.

1.2 Parabolic PDEs

PDEs containing partial derivatives higher than the second order are usually very complicated and not as often studied. PDEs are frequently of the form

$$\sum_{i,j=1}^{n+1} a_{ij}\frac{\partial^2 u}{\partial x_i \partial x_j} - q(x_1, x_2, \cdots, x_n, x_{n+1}, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \cdots, \frac{\partial u}{\partial x_n}, \frac{\partial u}{\partial x_{n+1}}) = 0, \tag{1.1}$$

where, $q(\cdot) \in R$. We assume that $t = x_{n+1}$ if the equations involve the variable t. a_{ij} may depend on $x_1, x_2, \cdots, x_n, x_{n+1}, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \cdots, \frac{\partial u}{\partial x_n}, \frac{\partial u}{\partial x_{n+1}}$. We can often assume that $a_{ij} = a_{ji}$, so that the matrix $A = [a_{ij}]$ is a symmetric matrix. If all eigenvalues of A have the same sign, then the equations are called elliptic PDEs. If at least one eigenvalue is zero, then the equations are called parabolic PDEs. If n of the eigenvalues have the same sign, and the remaining one has opposite sign, then the equations are called hyperbolic PDEs.

Equations in the form of (1.1) can be very complicated. It is difficult to deal with equations which have many variables. Also, if the coefficients a_{ij} are complicated functions, then the equations are usually difficult to solve. Many PDEs in real applications contain fewer variables, or even have constant coefficients, such as Laplace's equation, Poisson's equation, and the heat equation. Typical second order PDEs are:

$$a_1\frac{\partial^2 u}{\partial x_1^2} + a_2\frac{\partial^2 u}{\partial x_2^2} + \cdots + a_n\frac{\partial^2 u}{\partial x_n^2} - q = 0, \tag{1.2}$$

$$a_1\frac{\partial^2 u}{\partial x_1^2} + a_2\frac{\partial^2 u}{\partial x_2^2} + \cdots + a_n\frac{\partial^2 u}{\partial x_n^2} - q - \frac{\partial u}{\partial t} = 0, \tag{1.3}$$

$$a_1\frac{\partial^2 u}{\partial x_1^2} + a_2\frac{\partial^2 u}{\partial x_2^2} + \cdots + a_n\frac{\partial^2 u}{\partial x_n^2} - q - \frac{\partial^2 u}{\partial t^2} = 0, \tag{1.4}$$

where, in (1.2), $q = q(x_1, x_2, \cdots, x_n, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \cdots, \frac{\partial u}{\partial x_n})$, and in (1.3) and (1.4), $q = q(x_1, x_2, \cdots, x_n, u, t, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, \cdots, \frac{\partial u}{\partial x_n})$. The equations (1.2) are elliptic PDEs, the equations (1.3) are parabolic PDEs, and the equations (1.4) are hyperbolic PDEs. $a_1, a_2, \cdots, a_n$ are nonnegative constants. For elliptic PDEs of the form (1.2), at least two of $a_i, i = 1, 2, \cdots, n$, cannot be zero. For the other two, (1.3) and (1.4), at least one cannot be zero. The equations discussed in the present thesis are parabolic PDEs, which are used to describe phenomena that are time-dependent.

For introducing a new class of collocation methods for parabolic PDEs in this thesis, we only consider equations which have constant coefficients of value 1. For other constant coefficients, or for variable coefficients, the algorithm is very similar. Also, the equations considered in this thesis only contain three variables, (x, y, t), and they are linear. The form of the PDEs we consider will be given in Chapter 2. A brief discussion of the nonlinear case, and the case of equations containing four variables, (x, y, z, t), will be given in Chapter 8.

1.3 Classical numerical methods

The simplest numerical method for PDEs is the finite-difference method, which most books about numerical methods [1, 2] mention. Finite-difference methods use finite-difference equations to replace the partial differential equations at certain local grid points, to form discrete systems to be solved. These methods are especially suitable for equations depending on time (parabolic PDEs and hyperbolic PDEs). Different finite-differences lead to different methods. For parabolic PDEs, there are, for example, explicit methods, and implicit methods, including the Crank-Nicholson method. These methods are often easy to implement, but the accuracy is typically not very high. Also, it is not easy for these methods to deal with problems on irregular domains.

The Galerkin method is a classical numerical method which is based on concepts from functional analysis. It determines piecewise polynomials to approximately solve the differential equation on the full domain. Using different bases, the method will generate different polynomials. It is a suitable method for elliptic boundary value PDEs. However, the basis may be difficult to choose, if the domain is complicated.

The finite-element method is probably the most widely used method at present to solve engineering problems. Many computation software tools use this method for dealing with PDEs. For example, MATLAB uses this method in its PDE toolbox [3]. The finite-element method is normally derived from the Galerkin method. This method allows to deal with complicated domains. It refines the original domain into a certain mesh, so that it is easier to choose the basis and to form the piecewise polynomial in each small local region.

Collocation methods are widely used to solve ODE boundary value problems. For PDEs, they are at present not used very often. The class of collocation methods discussed in this thesis is new for parabolic PDEs.

1.4 Collocation methods

The new class of collocation methods for parabolic PDEs is derived from the idea used for elliptic PDEs by Doedel [4, 5].

For the time dimension, we use finite differences to replace the partial differential derivatives with respect to time. For the space dimensions, we refine the original domain, to generate small regions which are called local elements, and the full domain becomes a mesh. In each local element, at a given time step, we select matching points and collocation points to construct a piecewise polynomial, just like the new collocation methods for elliptic PDEs. Step by step in time, we compute the approximate numerical solutions, given certain initial conditions and boundary conditions. The main characteristics of the methods are:

1. High order of accuracy can be attained for the space dimensions. This is demonstrated for solutions of time-independent problems in Chapter 6.

2. The piecewise polynomial solutions need not be globally continuous. We only require a certain degree of continuity at so-called "matching points".

3. The linear systems generated during the solution process can be efficiently solved by the method of nested dissection. Nested dissection is usually the part needing the most calculations. Therefore, its efficient implementation is a very important objective.

As mentioned above, we only discuss relatively simple linear equations. In principle, the methods generalize to other cases, but they need some modifications in the construction of the polynomials. We will, however, treat some relatively complicated spatial domains.

1.5 Organization of the thesis

In order to explain the algorithm clearly, we focus on the new collocation methods for the case of linear parabolic PDEs. We also give examples to test the effectiveness of the methods. For some of these examples, the exact solution is known, which allows us to observe the accuracy. For a better understanding of the numerical solutions, a visualization tool based on OpenGL and X11/Motif has been developed to show $3D$ animations of the solutions. For the benefit of further study, nonlinear PDEs and PDEs in $3D$ space are also discussed briefly.

The thesis is organized as follows. Chapter 1 serves as an introduction. Chapter 2 describes the linear parabolic PDEs to be studied, as well as the idea of the collocation methods. Chapter 3 describes the Rivara algorithm that generates the meshes. Chapter 4 introduces the nested dissection method that eliminates the unknown variables at the interior matching points. The boundary conditions are then used to determine the solution at the matching points on the boundary, and back substitution is used to calculate the solution at the interior matching points. Chapter 5 describes the details of the implementation and the structure of the numerical software. Chapter 6 presents some schemes for selecting the matching points and the collocation points. Five parabolic PDE examples are used to illustrate solutions obtained by the collocation methods. Extensive numerical results and pictures are offered. Chapter 7 describes the details of visualizing the solutions, and discusses ways to do $3D$ plotting using OpenGL, and to save images as animation files. Chapter 8 gives some ideas on how to solve nonlinear parabolic PDEs, and PDEs in $3D$ space. An algorithm for bisecting tetrahedral regions and generating $3D$ meshes is also presented. Chapter 9 gives conclusions and discusses some topics that need further study. Appendices show the details of some important data structures and functions.

Chapter 2

Collocation Methods

2.1 Linear parabolic PDE problems

Consider the simple cases of (1.3), where all coefficients a_i, $i = (1, 2, \cdots, n)$ are equal to 1. The general second order parabolic PDEs can then be written as

$$\frac{\partial u}{\partial t} = \Delta u - q(X, t, u, \nabla u), \quad X \in \Omega \subset R^N,\ t \in [t_0, t_1] \subset R, \tag{2.1}$$

where $u(X, t) \in R, \Delta$ is Laplace's operator of u with respect to X, ∇u is the gradient of u with respect to X, $q(X, t, u, \nabla u) \in R$; *i.e.*,

$$\Delta = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2}, \quad \nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \cdots, \frac{\partial}{\partial x_n} \right)^*,$$

where $*$ denotes transpose, ∇u is a vector and $\Delta = \nabla \cdot \nabla$.

If $q(X, t, u, \nabla u)$ is of the form

$$q(X, t, u, \nabla u) = b(X, t)^* \nabla u + c(X, t) u + f(X, t), \quad X \in \Omega \subset R^N,\ t \in [t_0, t_1] \subset R, \tag{2.2}$$

where $b(X, t) \in R^N$, $c(X, t), f(X, t) \in R$, then Equation (2.1) is called a linear parabolic PDE. We can rewrite a linear parabolic PDE as

$$\frac{\partial u}{\partial t} = \Delta u - b(X, t)^* \nabla u - c(X, t) u - f(X, t), \quad x \in \Omega \subset R^N,\ t \in [t_0, t_1] \subset R. \tag{2.3}$$

In the $2D$ case, this becomes

$$\frac{\partial u}{\partial t} = \Delta u - b_1(x,y,t)\frac{\partial u}{\partial x} - b_2(x,y,t)\frac{\partial u}{\partial y} - c(x,y,t)u - f(x,y,t), \\ (x,y) \in \Omega \subset R^2,\ t \in [t_0, t_1] \subset R, \tag{2.4}$$

where $b_1, b_2 \in R$. Even simpler cases, in which q does not depend on ∇u and u, are of the form

$$\frac{\partial u}{\partial t} = \Delta u - f(x,y,t), \quad (x,y) \in \Omega \subset R^2,\ t \in [t_0, t_1] \subset R. \tag{2.5}$$

Most examples given in this thesis are of this type. As for the initial conditions, they take the form

$$u(X, t_0) = u_0(X), \quad X \in \Omega \subset R^N, t_0 \in R. \tag{2.6}$$

If the boundary conditions are of the form

$$u(X,t) = g(X,t), \quad X \in \partial\Omega, t \in [t_0, t_1] \subset R, \tag{2.7}$$

where $g \in R$, then they are called Dirichlet boundary conditions. If they are of the form

$$\frac{\partial u}{\partial \eta} = g(X,t), \quad X \in \partial\Omega, t \in [t_0, t_1] \subset R, \tag{2.8}$$

or

$$\frac{\partial u}{\partial \eta} + u = g(X,t), \quad X \in \partial\Omega, t \in [t_0, t_1] \subset R, \tag{2.9}$$

where η is the unit exterior normal vector, and $\frac{\partial u}{\partial \eta}$ is the derivative of u with respect to the direction of η, then they are called Neumann boundary conditions.

2.2 The finite difference approach

First, for a certain given small time step, we use a finite difference to replace the partial derivative with respect to t in (2.3). It approximates the original equation with error of

$O(dt)$ at a certain X.

$$\frac{u_k - u_{k-1}}{dt} = \Delta u_k - b(X, t_k)^* \nabla u_k - c(X, t_k) u_k - f(X, t_k), \quad k = 1, 2, \cdots, n, \tag{2.10}$$

where $dt = t_k - t_{k-1}$. Here dt is the step size in time, which can be fixed or variable. Inductively, having already computed u_{k-1}, we wish to solve for u_k. We can rewrite (2.10) in the form of a linear operator:

$$Lu_k \equiv \Delta u_k - b(X, t_k)^* \nabla u_k - c(X, t_k) u_k - \frac{1}{dt} u_k = f(X, t_k) - \frac{1}{dt} u_{k-1}. \tag{2.11}$$

This is a linear elliptic PDE problem at time t_k,

$$Lu_k = f_k(X), \quad \text{where } f_k(X) = f(X, t_k) - \frac{1}{dt} u_{k-1}. \tag{2.12}$$

Now consider the space domain Ω. The domain can in principle be quite general. For clarity, we will start with some simple cases. More complex domains can be represented by a combination of several simple ones. Examples will be shown later. Let us assume that we have a simple N-cube. For the $2D$ case, this is a square. Since we want to apply refinement in order to generate smaller regions, we subdivide Ω into two subdomains. This subdivision is continued recursively until a desired level of refinement is reached. We call the smallest regions "finite elements". A binary tree can be used to represent this data structure. Although the binary tree is not necessary for the discretization, it is useful for the nested dissection algorithm described in Chapter 4. An example is provided in Figure 2.1.

Although square elements work well if the domain Ω is itself a square region, they cannot easily deal with more general domains. For example, they cannot easily deal with a triangular domain, since the domain contains an angle that is not a right angle. In this case, it is better to divide Ω into smaller triangular elements. First we subdivide the original square domain Ω into two triangles, and then we apply recursive subdivision, as shown in Figure 2.2.

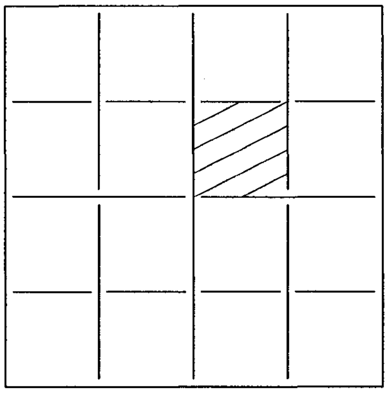


Figure 2.1: A domain Ω with a square recursive subdivision.

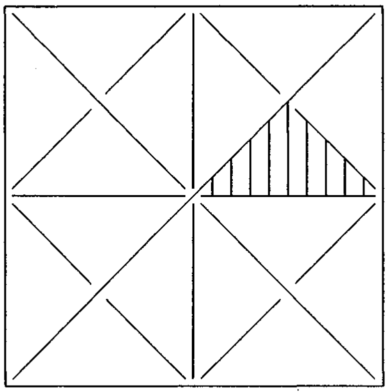


Figure 2.2: A domain Ω with a triangular recursive subdivision.

In this thesis, we use the Rivara algorithm to generate triangular meshes, as will be discussed in Section 3.1.

Consider any finite element Ω_l in Ω, for example, the shaded element in Figure 2.1 or in Figure 2.2. If $u(X, t_k)$ is known on $\partial\Omega_l$ and if Ω_l is small enough, then the solution $u(X, t_k)$ of the local elliptic problem is defined in Ω_l. Also, $\frac{\partial u}{\partial \eta}$, the normal derivative of u on the boundary $\partial\Omega_l$, will be defined along any smooth part of the boundary. For any finite element Ω_l, an "ansatz" discretization formula for (2.12) at points $x_i, i = 1, 2, \cdots, n$, selected on the boundary of the finite element Ω_l, can be written as

$$v_i^k = \sum_{j=1}^{n} \alpha_{ij} u_j^k + \sum_{j=1}^{m} \beta_{ij} f_k(z_j), \quad i = 1, 2, \cdots, n. \tag{2.13}$$

The points x_i will be called "matching points". Above, u_i^k denotes the approximate solution at point x_i and at time step t_k, $u_i^k \approx u(x_i, t_k)$. Similarly, v_i^k denotes the normal derivative,

$v_i^k \approx \nabla u(x_i, t_k)^* \eta_i$, where η_i is the unit exterior normal to the finite element's boundary at x_i. The points $z_j, j = 1, 2, \cdots, m$, lying inside Ω_l or near it, are called "collocation points". There are many strategies to select the matching points and the collocation points, and some strategies may give a much higher accuracy than others. Indices denoting the element have been suppressed for simplicity of notation. An example is shown in Figure 2.3.

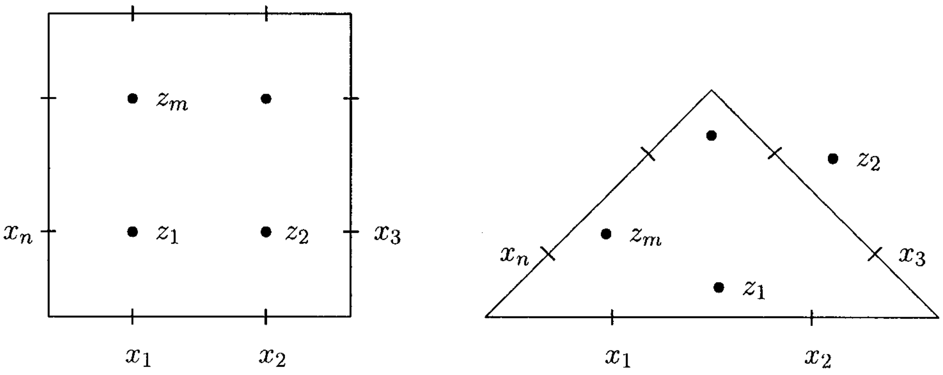


Figure 2.3: A finite domain of Ω.

For each finite element of Ω, there is a discrete equation corresponding to each x_i on the boundary of the finite element. The coefficients α_{ij} and β_{ij} are determined by requiring (2.13) to be satisfied exactly for a basic set of equations with known solution. Let ϕ_k be a polynomial, and $f_k = L\phi_k$. Then the equation $Lu_k = f_k$ has the obvious solution $u_k = \phi_k$. More precisely, let P_{n+m} be a polynomial space of dimension $n + m$, and require that

$$\nabla\phi_k(x_i)^*\eta_i = \sum_{j=1}^{n} \alpha_{ij}\phi_k(x_j) + \sum_{j=1}^{m} \beta_{ij}L\phi_k(z_j), \; i = 1, 2, \cdots, n, \; \forall\phi_k \in P_{n+m}. \tag{2.14}$$

Let $u_k = (u_1^k, u_2^k, \cdots, u_n^k)^*$, $v_k = (v_1^k, v_2^k, \cdots, u_n^k)^*$, $f_k = (f_k(z_1), f_k(z_2), \cdots, f_k(z_m))^*$, and

$$A = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2n} \\ \vdots & \vdots & & \vdots \\ \alpha_{n1} & \alpha_{n2} & \cdots & \alpha_{nn} \end{pmatrix}, \quad B = \begin{pmatrix} \beta_{11} & \beta_{12} & \cdots & \beta_{1m} \\ \beta_{21} & \beta_{22} & \cdots & \beta_{2m} \\ \vdots & \vdots & & \vdots \\ \beta_{n1} & \beta_{n2} & \cdots & \beta_{nm} \end{pmatrix}.$$

Then (2.13) can be written as

$$v_k = Au_k + Bf_k. \tag{2.15}$$

Suppose $\phi_i, i = 1, 2, \cdots, n+m$ form a basis of P_{n+m}: $Span\{\phi_1, \phi_2, \cdots, \phi_{n+m}\} = P_{n+m}$. If furthermore we define

$$\Phi = \begin{pmatrix} \phi_1(x_1) & \phi_1(x_2) & \cdots & \phi_1(x_n) \\ \phi_2(x_1) & \phi_2(x_2) & \cdots & \phi_2(x_n) \\ \vdots & \vdots & & \vdots \\ \phi_{n+m}(x_1) & \phi_{n+m}(x_2) & \cdots & \phi_{n+m}(x_n) \end{pmatrix}, \tag{2.16}$$

$$L_\Phi = \begin{pmatrix} L\phi_1(z_1) & L\phi_1(z_2) & \cdots & L\phi_1(z_m) \\ L\phi_2(z_1) & L\phi_2(z_2) & \cdots & L\phi_2(z_m) \\ \vdots & \vdots & & \vdots \\ L\phi_{n+m}(z_1) & L\phi_{n+m}(z_2) & \cdots & L\phi_{n+m}(z_m) \end{pmatrix}, \tag{2.17}$$

and

$$R_\Phi = \begin{pmatrix} \nabla\phi_1(x_1)^*\eta_1 & \nabla\phi_1(x_2)^*\eta_2 & \cdots & \nabla\phi_1(x_n)^*\eta_n \\ \nabla\phi_2(x_1)^*\eta_1 & \nabla\phi_2(x_2)^*\eta_2 & \cdots & \nabla\phi_2(x_n)^*\eta_n \\ \vdots & \vdots & & \vdots \\ \nabla\phi_{n+m}(x_1)^*\eta_1 & \nabla\phi_{n+m}(x_2)^*\eta_2 & \cdots & \nabla\phi_{n+m}(x_n)^*\eta_n \end{pmatrix}, \tag{2.18}$$

then Equation (2.14) can be written as

$$(\Phi | L_\Phi) \begin{pmatrix} A^* \\ B^* \end{pmatrix} = R_\Phi. \tag{2.19}$$

For the finite difference approximation to be well defined, the matrix $(\Phi|L_\Phi)$ must be nonsingular. There are many possible schemes for choosing the matching points x_i, the collocation points z_j, and the space P_{n+m} so that nonsingularity is satisfied.

In addition to (2.15), we also require the boundary conditions to be satisfied. For example, for Dirichlet boundary conditions, we require that

$$u_i^k = g(x_i, t_k), \quad x_i^k \in \partial\Omega \quad \text{at} \quad t = t_k. \tag{2.20}$$

2.3 The collocation approach

To find the approximate solutions, we associate a polynomial $p(X, t_k) \in P_{n+m}$ to each element at time step t_k. Here P_{n+m} is an appropriate $(n+m)$-dimensional polynomial space, spanned by basis functions $\phi_i, i = 1, 2, \cdots, n+m$,

$$p(X, t_k) = \sum_{i=1}^{n+m} c_i(t_k)\phi_i(X), \quad X \in \Omega_l. \tag{2.21}$$

For any two adjacent finite elements, the following conditions are required to be satisfied at the matching points x_i on the common boundary:

- The values of the neighboring polynomials match,
- The normal derivatives of the neighboring polynomials match with values of opposite sign. (2.22)

Further, each polynomial must satisfy the collocation equations at the collocation points z_j of the corresponding finite element.

$$Lp(z_j, t_k) = f_k(z_j), \quad j = 1, 2, \cdots, m. \tag{2.23}$$

Also, the boundary conditions must be satisfied. For Dirichlet conditions,

$$p(x_i, t_k) = g(x_i, t_k), \quad x_i^k \in \partial\Omega \text{ at } t = t_k. \tag{2.24}$$

The finite difference scheme from Section 2.2 and the collocation scheme are equivalent in the following sense:

Theorem 1 *For each finite element Ω_l, let the matrix $(\Phi|L_\Phi)$ be nonsingular and let the matrices A and B be defined by (2.19). Suppose there is a solution of the collocation scheme (2.22), (2.23) and (2.24). Then the values of the local polynomial and its normal derivative for all finite elements, evaluated at the points x_i, will satisfy the finite difference equations (2.15).*

Conversely, suppose there is a solution of the system of finite difference equations (2.15) *of all elements, also satisfying the boundary conditions (2.20). Then for each element, one can uniquely interpolate the finite difference solution at the matching points by a local polynomial $p(X, t_k) \in P_{n+m}$ that also satisfies the differential equation at the collocation points z_j. The local polynomials so constructed satisfy the continuity properties in (2.22).*

Proof. By (2.21), we have $p(X, t_k) = \sum_{i=1}^{n+m} c_i(t_k)\phi_i(X)$. The collocation equation (2.23) can now be written as $\sum_{i=1}^{n+m} c_i(t_k)L\phi_i(z_j) = f_k(z_j)$, or $L_\Phi^* c_k = f_k$ with L_Φ defined in (2.17), and where $c_k = (c_1(t_k), c_2(t_k), \cdots, c_{n+m}(t_k))^*$, and f_k is as in (2.15). Let $u_k = \Phi^* c_k$ and $v_k = R_\Phi^* c_k$, and use (2.19). Then

$$\begin{aligned} v_k - Au_k - Bf_k &= R_\Phi^* c_k - A\Phi^* c_k - Bf_k \\ &= [c_k^*(R_\Phi - \Phi A^*)]^* - Bf_k \\ &= [c_k^*(L_\Phi B^*)]^* - Bf_k \\ &= BL_\Phi^* c_k - Bf_k = B(L_\Phi^* c_k - f_k) = 0. \end{aligned}$$

Conversely, let (u_k, v_k) be a solution of the finite difference equations (2.15). Define $c_k = (c_1(t_k), c_2(t_k), \cdots, c_{n+m}(t_k))$ by

$$\begin{pmatrix} \Phi^* \\ L_\Phi^* \end{pmatrix} c_k = \begin{pmatrix} u_k \\ f_k \end{pmatrix}, \tag{2.25}$$

and let $p(X, t_k) = \sum_{i=1}^{n+m} c_i(t_k)\phi_i(X)$. Thus, c_k satisfies $\Phi^* c_k = u_k$ and $L_\Phi^* c_k = f_k$. Hence $p(X, t_k)$ satisfies the collocation equations (2.23). Since $(p(x_1, t_k), p(x_2, t_k), \cdots, p(x_n, t_k))^* = \Phi^* c_k = u_k$, neighboring polynomials are continuous at the points x_i. As for $\nabla p(X, t_k)^* \eta$, where $\eta = (\eta_1, \eta_2, \cdots, \eta_n)$, we have

$$\begin{aligned} \begin{pmatrix} \nabla p(x_1, t_k)^* \eta_1 \\ \nabla p(x_2, t_k)^* \eta_2 \\ \vdots \\ \nabla p(x_n, t_k)^* \eta_n \end{pmatrix} &= R_\Phi^* c_k = (\Phi A^* + L_\Phi B^*)^* c_k \\ &= A\Phi^* c_k + BL_\Phi^* c_k \end{aligned}$$

$$= \quad Au_k + Bf_k = v_k.$$

It follows that the value of $\nabla p(X, t_k)^* \eta$ for neighboring polynomials is also continuous at x_i.

2.4 Local basis construction

In the system (2.19), the coefficients α_{ij} and β_{ij} must be solved for each finite element. For this purpose, a basis $\phi_i, i = 1, 2, \cdots, n+m$ must be chosen. Generally, the basis of power functions is a good choice. For example, this basis is $\{1, x, y, x^2, xy, y^2, \cdots\}$ for the $2D$ case.

When calculating Φ, L_Φ and R_Φ, the values of x_i and z_j should be chosen as "local values", since this will avoid loss of significant digits. For example, in a local triangular element, we can use the distance between each x_i and the center of the element. Also, we can use LU-decomposition for the matrix $(\Phi|L_\Phi)$, in order to avoid calculating the inverse matrix, and reduce the amount of calculation, as will be shown in Section 5.6. Furthermore, from Equation (2.10) we can see that, if the coefficients b, c are time-independent, dt is constant, and the mesh does not change with time, we only need to calculate $(A|B)$ once and store it in memory. All examples given in this thesis belong to this case.

Chapter 3

The Rivara Algorithm and Triangular Meshes

As mentioned in Section 2.2, we need to refine the domain Ω. Since square elements are too restrictive, triangular elements will be used. There are many algorithms which use refinement to generate triangular meshes. The main reason to use triangular meshes is that it allows adaptive methods, which need less computation. Bank [15, 16, 17] developed a software system PLTMG for this purpose. For triangular refinement, there are two issues that we need to consider:

1. The number of triangles generated should be small, whenever possible.

2. The minimal angle in the generated triangles should not be too small, otherwise it will influence the accuracy of final solutions.

The Rivara algorithm [20, 21, 22, 61], which is used in this thesis, is a method that satisfies the above requirements.

3.1 The Rivara algorithm

A basic algorithm for applying refinement to a triangular domain is the Rivara algorithm. If the domain is a polygon, then we can first divide it into several triangles, and then use the Rivara algorithm to do further refinement. In the Rivara algorithm, a triangular region is always refined by inserting an additional edge from the midpoint of the longest edge to the opposite corner, as shown in Figure 3.1. The following steps define the Rivara algorithm:

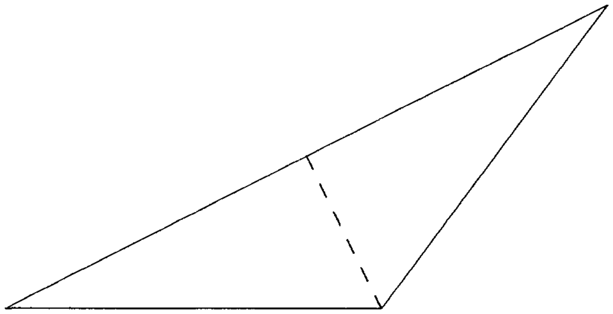


Figure 3.1: The refinement of a triangle.

1. First refine all triangles that need to be refined, based on certain refinement criteria. For example, refine any triangle whose longest edge is larger than a certain value, or whose center is near a refinement point. The refinement continues recursively until the refinement criterion has been met. The division of any triangle should be done by inserting an additional edge from the midpoint of the longest edge to the opposite corner.

2. Going back to the full domain, recursively find any triangle which has not been divided, but has edges that have already been divided. This means that the edges were divided while dividing neighboring triangles. The midpoints of these edges are called "hanging points". In another words, one must find the triangles with hanging points. Then we refine these triangles based on the longest edge criterion.

3. Repeatedly check for triangles with hanging points, and refine these triangles based on the longest edge criterion, until no such triangles remain.

The Rivara algorithm [22] has been proven to terminate with a regular mesh in a finite number of steps. Also, it will not generate bad angles, such as 0 and π. In fact, if α is the minimal angle in the original domain, then the minimal angle in the mesh, obtained after the algorithm terminates, is not less than $\alpha/2$. This was proved by Rosenberg and Stenger [24]. Figure 3.2 gives an example of how to deal with a hanging point P. The minimal angle of the mesh on the right will not be less than one half of the minimal angle on the left.

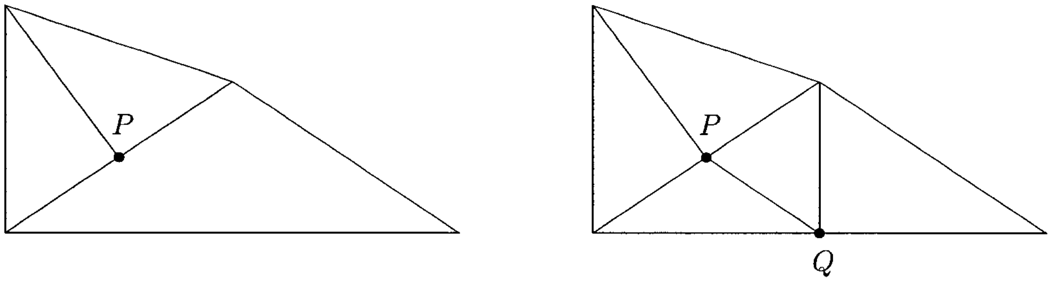


Figure 3.2: Elimination of a hanging point P according to the Rivara algorithm.

3.2 A variant of the Rivara algorithm

A variant of the Rivara algorithm to obtain a stronger and more uniform refinement, is to divide first all triangles that need to be refined into four subtriangles [20], as shown in Figure 3.3, and then to use the Rivara algorithm to apply further refinement. Again, this algorithm will terminate, and the minimal angle generated will not be less than one half of the original minimal angle. Considering the binary tree representing the mesh structure, the first step of this algorithm will make building the tree more complex, since more intermediate nodes need to be added into the tree. Also, these triangles cannot be treated as simple local triangular regions. This is why in this thesis we use the original Rivara algorithm.

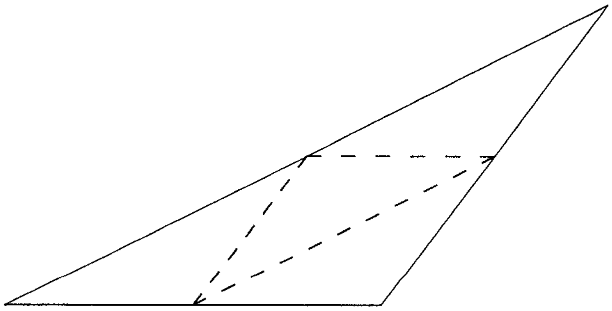


Figure 3.3: The refinement of a triangle into four equal parts.

3.3 Triangular meshes

The Rivara algorithm can generate different triangular meshes if the original domain or the refinement criteria are varied. A very simple criterion is that the longest edge of any triangular elements should not be larger than a certain value. If there is any local triangular region that has the longest edge larger than this value, then it needs to be divided. Otherwise, no division is necessary. Once there is no such local region left, we need to do further refinement to deal with the hanging points. Such a process will result in a uniform mesh if the original domain has regular shape, such as a square. This criterion is straightforward and easy to implement. However, if the solution of a PDE exhibits a peak, or even a singularity, then it is necessary to generate a mesh of high overall density, which will require much computation. Therefore, local refinements are needed to deal with sharp peaks and other rapid local variations of the solution of the PDE.

Figure 3.4 shows a uniform mesh for a unit square, generated with the criterion that the longest edge of any element must to be shorter than 0.2. Here we use d_m to represent the upper limit of the longest edge of the element. Figure 3.7 shows local refinement near

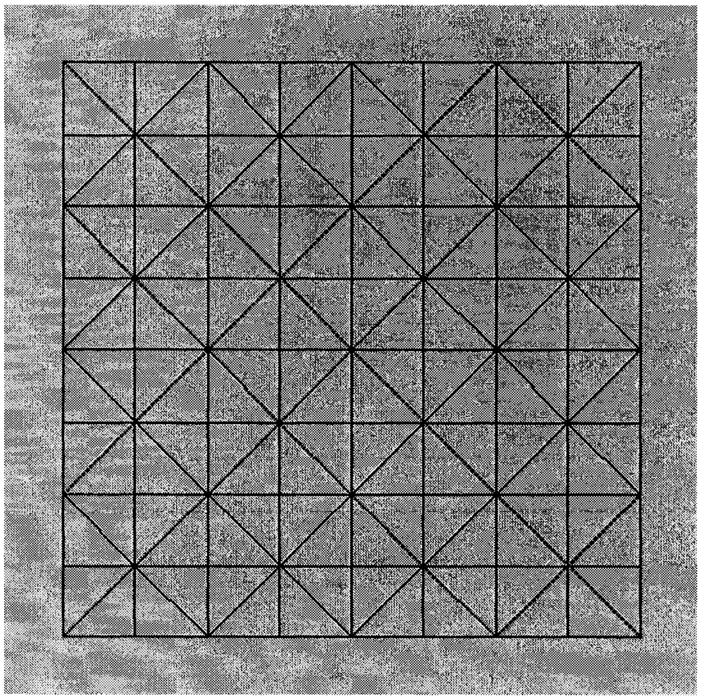


Figure 3.4: A uniform mesh generated by the Rivara algorithm of a unit square Ω, $d_m = 0.2$.

the points (0.5, 0.5) and (0.25, 0.625), with $d_m = 0.015$. Also, any local region, for which the distance from its center to one of the two refinement points is less than $d_r = 0.05$, may need refining. Figure 3.6 shows local refinement near the points (0.5, 0.5) and (0.25, 0.625) with $d_m = 0.0015$ and $d_r = 0.05$. Some other examples are shown in Figure 3.7 and Figure 3.8. Here, d_r denotes the upper limit of the distance from the center of the local region that needs to be refined to a reference line.

3.4 Higher dimensional cases

The Rivara algorithm is suitable for $2D$ space domains. As mentioned above, the minimal angle obtained will be no less than one half of the original minimal angle. This fact has only been proved for the $2D$ case. Spaces of dimension higher than two will pose a problem for the Rivara algorithm. For example, for $3D$ spaces, we can divide a tetrahedron similarly by adding a new surface passing through the midpoint of its longest edge and passing through the edge opposite to its longest edge, but we do not know whether recusively refining in this way will generate strange local regions or not. There are many recent papers that discuss the refinement of $3D$ space domains. Some discussion of this topic will also be given in Section 8.2.

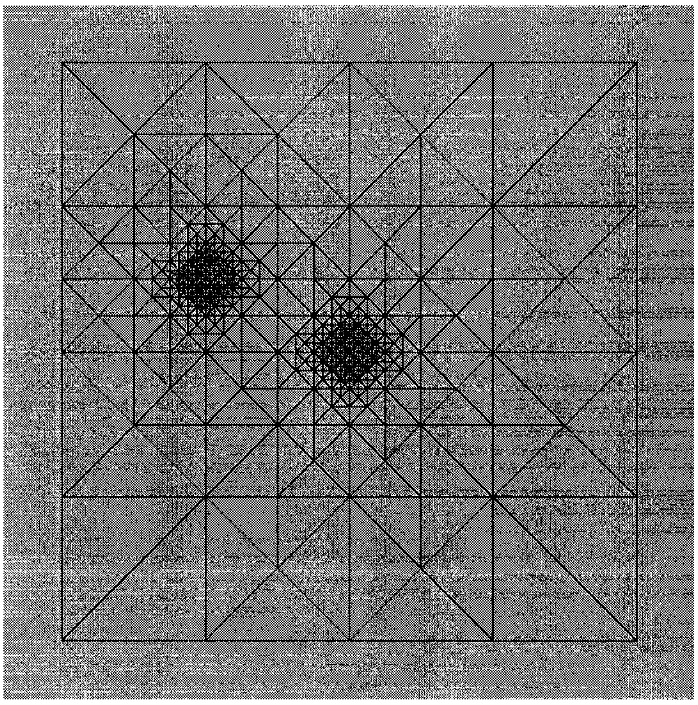


Figure 3.5: Local refinement near two points of a unit square Ω, $d_m = 0.015$, $d_r = 0.05$.

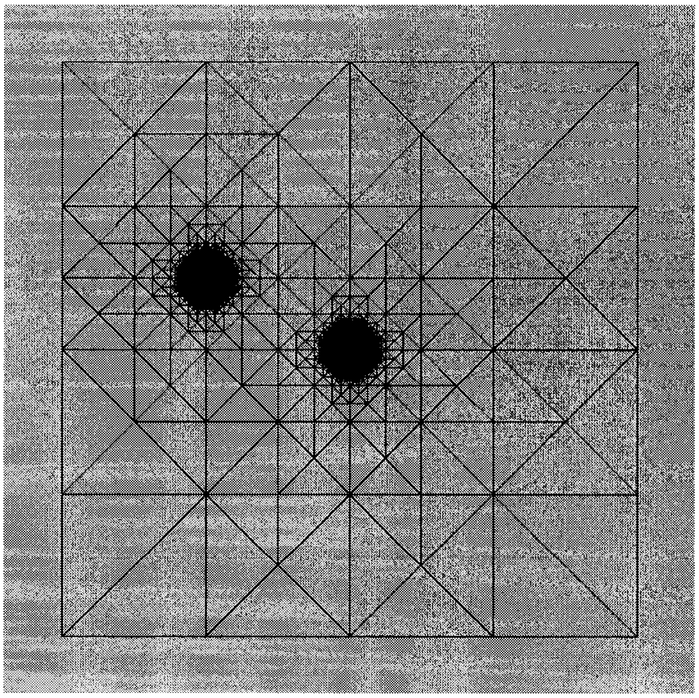


Figure 3.6: Local refinement near two points of a unit square Ω, $d_m = 0.0015$, $d_r = 0.05$.

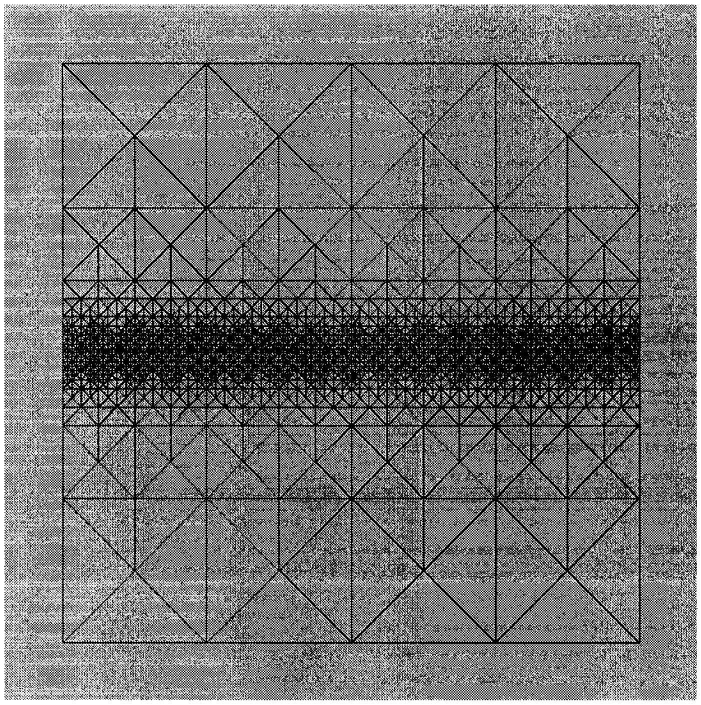


Figure 3.7: Local refinement near the line $x = 0.5$ of a unit square Ω, $d_m = 0.015$, $d_r = 0.05$.

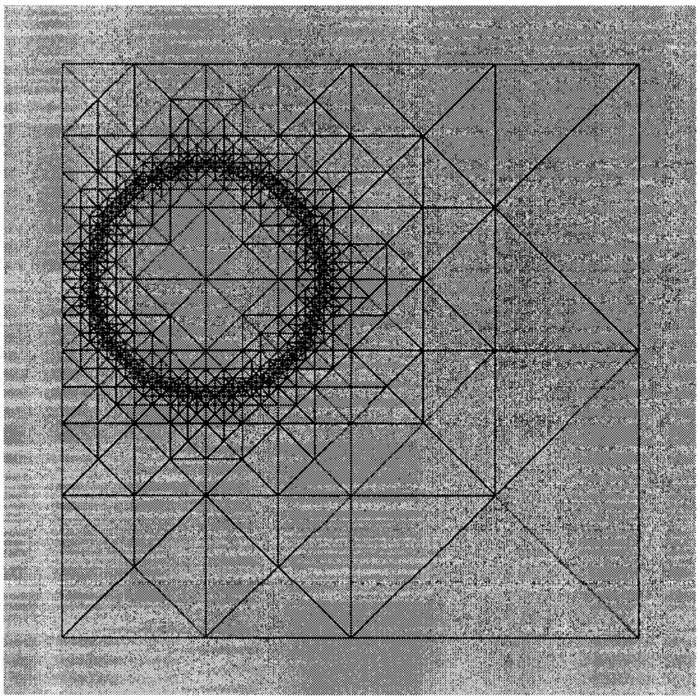


Figure 3.8: Local refinement near the circle $(x-0.25)^2+(y-0.625)^2 = 0.04$ of a unit square Ω, $d_m = 0.015$, $d_r = 0.005$.

Chapter 4

Nested Dissection

4.1 The procedure

The nested dissection algorithm does backward recursive elimination of the unknown u_k and v_k at the matching points on common boundaries of adjacent local regions. "Adjacent regions" means that the two regions result from subdivision of their union region. These two regions can be a polygon and a triangle generated by subdividing the original polygonal domain, or two triangles generated by subdividing a triangular region. Thus they can be represented as the descendant nodes of a common parent node in a recursive binary tree. "Backward recursion" means that the elimination starts at the leaves and terminates at the root of the binary tree.

This recursion procedure will result in the elimination of all unknown u_k and v_k at the interior matching points. What is left is a system in the unknown u_k and v_k only at the points x_i on $\partial\Omega$. The boundary conditions are then used to determine the values of u_k and v_k at x_i on $\partial\Omega$. Thereafter, a recursive backsubstitution will give the values of u_k and v_k at the matching points on each interior boundary.

For the Dirichlet boundary conditions (2.20), the values of u_k and v_k at the matching points on $\partial\Omega$ can be calculated directly. For Neumann boundary conditions, the situation is a little more complex. We need to combine the final system of equations, after elimination of the interior unknown u_k and v_k, with the boundary condition at each x_i on $\partial\Omega$, to form a new system and solve it.

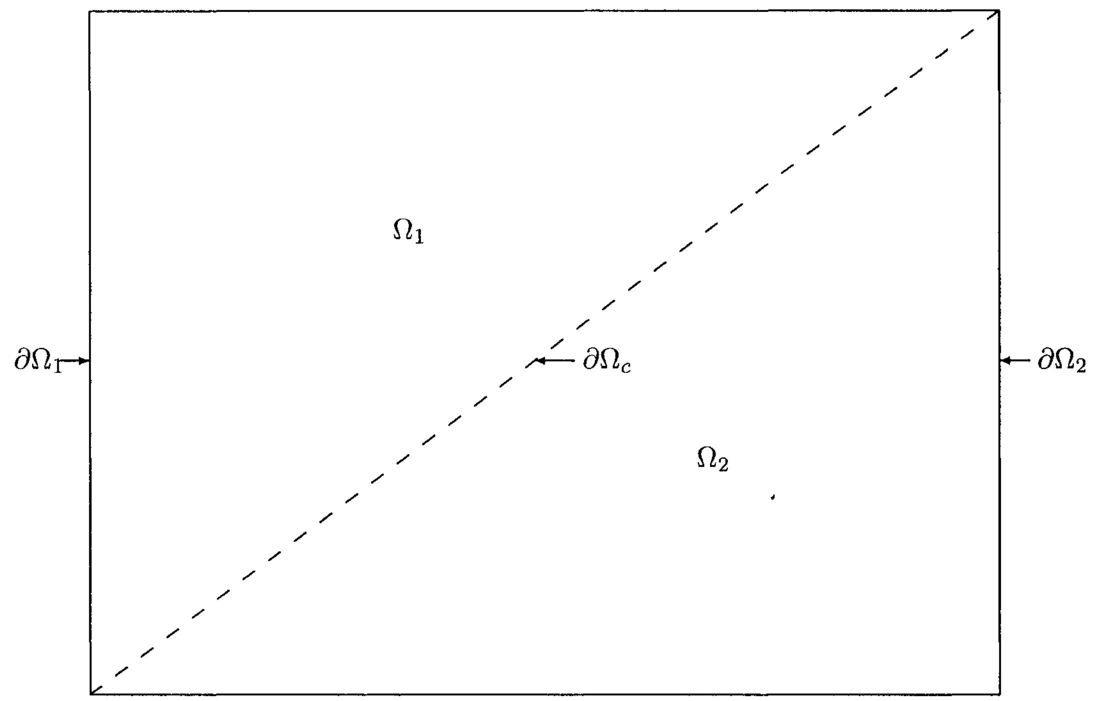


Figure 4.1: Two adjacent subregions and their union.

4.2 Local elimination

Consider two adjacent regions, Ω_1 and Ω_2, as shown in Figure 4.1. They can be finite elements or local regions, represented in the binary tree. In Ω_1, the common boundary will be called $\partial\Omega_c$, and the remaining part will be called $\partial\Omega_1$. In Ω_2, the remaining part will be called $\partial\Omega_2$. The variable u_k for region Ω_1 in the finite difference equation (2.15) is split into u_k^{12} with the corresponding x_i on $\partial\Omega_c$ and u_k^1 with the corresponding x_i on $\partial\Omega_1$. Similarly, u_k for region Ω_2 is split into u_k^{21} and u_k^2. The variable v_k for each region is split similarly.

Let

$$f_{ck} = Bf_k. \tag{4.1}$$

For region Ω_1, Equation (2.15) can be written in the form

$$v_k^1 = A_1u_k^1 + A_{12}u_k^{12} + f_{ck}^1, \tag{4.2}$$

$$v_k^{12} = B_1u_k^1 + B_{12}u_k^{12} + f_{ck}^{12}, \tag{4.3}$$

while for Ω_2, their form is

$$v_k^2 = A_2 u_k^2 + A_{21} u_k^{21} + f_{ck}^2, \tag{4.4}$$

$$v_k^{21} = B_2 u_k^2 + B_{21} u_k^{21} + f_{ck}^{21}, \tag{4.5}$$

where $(f_{ck}^1, f_{ck}^{12})^* = f_{ck}$ for Ω_1, $(f_{ck}^2, f_{ck}^{21})^*$ is similar for Ω_2. The continuity relations (2.22) give the equations

$$u_k^{12} = u_k^{21}, \quad v_k^{12} = -v_k^{21}. \tag{4.6}$$

Define

$$B_m \equiv B_{12} + B_{21}. \tag{4.7}$$

Then from (4.3), (4.5) and (4.6), it follows that

$$\begin{aligned} B_1 u_k^1 + B_{12} u_k^{12} + f_{ck}^{12} &= -(B_2 u_k^2 + B_{21} u_k^{21} + f_{ck}^{21}) \\ (B_{12} + B_{21}) u_k^{12} &= -(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^{12} + f_{ck}^{21}) \\ B_m u_k^{12} &= -(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^{12} + f_{ck}^{21}) \\ u_k^{12} &= -B_m^{-1}(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^{12} + f_{ck}^{21}). \end{aligned} \tag{4.8}$$

We substitute this equation into (4.2) and (4.4). Again using (4.6), we obtain

$$v_k^1 = A_1 u_k^1 - A_{12} B_m^{-1}(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^{12} + f_{ck}^{21}) + f_{ck}^1, \tag{4.9}$$

$$v_k^2 = A_2 u_k^2 - A_{21} B_m^{-1}(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^{12} + f_{ck}^{21}) + f_{ck}^2. \tag{4.10}$$

We can rewrite the above as

$$\begin{aligned} v_k^1 &= (A_1 - A_{12} B_m^{-1} B_1) u_k^1 + (-A_{12} B_m^{-1} B_2) u_k^2 + [f_{ck}^1 - A_{12} B_m^{-1}(f_{ck}^{12} + f_{ck}^{21})], \\ v_k^2 &= (-A_{21} B_m^{-1} B_1) u_k^1 + (A_2 - A_{21} B_m^{-1} B_2) u_k^2 + [f_{ck}^2 - A_{21} B_m^{-1}(f_{ck}^{12} + f_{ck}^{21})]. \end{aligned}$$

Define

$$C_{11} = A_1 - A_{12}B_m^{-1}B_1, \quad C_{12} = -A_{12}B_m^{-1}B_2,$$
$$C_{21} = -A_{21}B_m^{-1}B_1, \quad C_{22} = A_2 - A_{21}B_m^{-1}B_2,$$

and let $f_{ck}^m = f_{ck}^{12} + f_{ck}^{21}$, and

$$\begin{aligned} f_{ck}^{s1} &= f_{ck}^1 - A_{12}B_m^{-1}f_{ck}^m, \\ f_{ck}^{s2} &= f_{ck}^2 - A_{21}B_m^{-1}f_{ck}^m. \end{aligned}$$

Then the equations (4.9) and (4.10) can be written in the form

$$v_k^1 = C_{11}u_k^1 + C_{12}u_k^2 + f_{ck}^{s1}, \quad v_k^2 = C_{21}u_k^1 + C_{22}u_k^2 + f_{ck}^{s2}. \tag{4.11}$$

Equations (4.11) are the discrete equations for the composite region. Thus, for the union of Ω_1 and Ω_2, after eliminating u_k^{12} and v_k^{12} on the common boundary, the new equations are again of the form (2.15).

4.3 Complexity estimation

For nested dissection, for Equations (4.2)-(4.5) and (4.11) we need the following vector and matrix calculations:

$$B_m = B_{12} + B_{21}, \quad f_{ck}^m = f_{ck}^{12} + f_{ck}^{21},$$
$$B_mD_1 = B_1, \quad B_mD_2 = B_2, \quad B_mg_k = f_{ck}^m,$$
$$C_{11} = A_1 - A_{12}D_1, \quad C_{12} = -A_{12}D_2, \quad f_{ck}^{s1} = f_{ck}^1 - A_{12}g_k,$$
$$C_{21} = -A_{21}D_1, \quad C_{22} = A_2 - A_{21}D_2, \quad f_{ck}^{s2} = f_{ck}^2 - A_{21}g_k.$$

The matrices in these equations have the following dimensions:

$$\begin{array}{llll} A_1 : n_1 \times n_1, & A_{12} : n_1 \times n_{12}, & A_2 : n_2 \times n_2, & A_{21} : n_2 \times n_{12}, \\ B_1 : n_{12} \times n_1, & B_{12} : n_{12} \times n_{12}, & B_2 : n_{12} \times n_2, & B_{21} : n_{12} \times n_{12}, \\ D_1 : n_{12} \times n_1, & & D_2 : n_{12} \times n_2. & \end{array}$$

Now we count the number of arithmetic operations, restricting the count to multiplications and divisions. To perform a LU-decomposition of B, we need $\frac{1}{3}(n_{12}^2-1)n_{12}$ operations. To compute D_1, we need $n_{12}^2 n_1$ operations. Similarly, D_2 needs $n_{12}^2 n_2$ and g_k needs n_{12}^2 operations, while $A_{12}D_1$ needs $n_{12}n_1^2$ operations. $A_{12}D_2$, $A_{12}g_k$, $A_{21}D_1$, $A_{21}D_2$ and $A_{21}g_k$ are similar to $A_{12}D_2$. Given this, the total number of arithmetic operations W of the nested dissection of a local region is as follows:

$$\begin{aligned} W &= \frac{1}{3}(n_{12}^2 - 1)n_{12} + n_{12}^2 n_1 + n_{12}^2 n_2 + n_{12}^2 + n_{12} n_1^2 \\ &\quad + n_{12} n_2^2 + 2 n_{12} n_1 n_2 + n_{12} n_1 + n_{12} n_2 \\ &= \frac{1}{3} n_{12}(n_{12}^2 + 3n_{12}n_1 + 3n_{12}n_2 + 3n_{12} + 3n_1^2 + 3n_2^2 \\ &\quad + 6n_1 n_2 + 3n_1 + 3n_2) - \frac{1}{3} n_{12} \\ &= \frac{1}{3}[(n_{12} + n_1 + n_2)^3 - (n_1 + n_2)^3] - \frac{1}{3} n_{12}, \end{aligned}$$

or

$$W \approx \frac{1}{3}[(n_{12} + n_1 + n_2)^3 - (n_1 + n_2)^3]. \tag{4.12}$$

Note that when the generated mesh does not change with time, the matrix A in (2.15) does not change either. In fact, all matrices remain unchanged. Only the vector f_{ck}^m changes with time. Thus, we need to calculate the matrices only once and save them in memory. At each time step, we only need to update f_{ck}^m, which costs $(n_{12} + n_1 + n_2)n_{12}$ arithmetic operations. For example, if the unit square Ω is divided into a uniform mesh, and the level of recursive division is 5, as shown in Figure 4.2, then there will be 32 elements. For example, suppose each edge of each element contains three matching points, and the number of time steps is 100. Based on (4.12), taking account of the costs of updating f_{ck}^m at each time step, the total number of arithmetic operations will approximately be as follows:

From $a \rightarrow b$, we need

$$W_{local} \approx \frac{1}{3}[2 \times (3^3 - 2^3) + (7^3 - 6^3)] \times 3^3 = 55 \times 3^3.$$

From $a \rightarrow b \rightarrow c$, we globally need

$$\begin{aligned} W_{global} &\approx \sum_{k=1}^{2} 2^{2k-1} \times 55 \times (2^{3-k} \times 3)^3 + \frac{1}{3}(5^3 - 4^3)(2^2 \times 3)^3 \\ &= 55 \times 5184 + 61 \times 576 \\ &= 320256. \end{aligned}$$

Therefore, taking into account the updating of f_{ck}^m at each time step, the total is

$$\begin{aligned} W_{total} &\approx W_{global} + [\sum_{k=1}^{2} 2^{2k-1} \times 11 \times (2^{3-k} \times 3)^2 \\ &\quad + \frac{1}{3} \times 5 \times (2^2 \times 3)^2] \times (100 - 1) \\ &= 320256 + 651024 \\ &= 971280. \end{aligned}$$

Here it takes much more calculations to get the solution at the first time step than at other time steps. For other cases, where the mesh changes or where the basis changes with time, the matrices need to be recalculated at each time step.

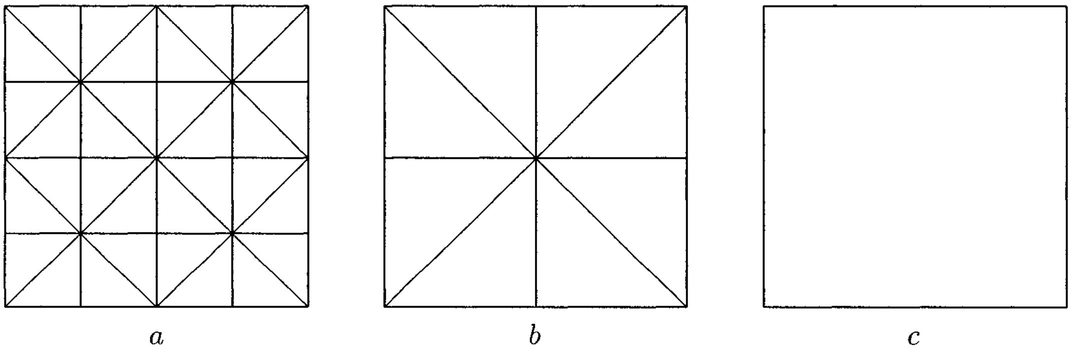


Figure 4.2: Nested dissection of a unit square Ω.

Chapter 5

Implementation

The numerical solution of linear parabolic PDEs, using collocation methods, is a complex procedure. This is why the data structures of the domain Ω and the generated meshes should be designed appropriately, and the Rivara algorithm and the nested dissection method should be implemented efficiently. Also, various kinds of boundary conditions must be dealt with. The implementation may also be combined with a visualization tool to give instant animation of the numerical solutions.

5.1 Objectives

In order to make the implementation flexible, in order to adapt to various PDEs, and to be easily expanded in further development, there are certain objectives which we need to consider:

- A suitable way to generate the basis of the power functions in any dimension.
- Appropriate data structures to describe general domains and generated meshes in any dimension, and to keep intermediate data for later use, for example, matrices and coefficients of local polynomials.
- An efficient way to implement the Rivara algorithm.
- An efficient way to do matrix calculations.

- A uniform way to deal with Dirichlet boundary conditions and with Neumann boundary conditions.
- A good implementation of the nested dissection method to deal with different meshes, for example, the mesh that has neighboring local regions sharing two common faces, as shown in Section 5.7.
- A runtime structure suitable for both calculation and instant visualization.

We have chosen the Language C/C++ to implement the calculation code, since this language is very flexible, efficient, and widely used on almost any platform. Also, it is very suitable to describe variable data structures, and it is easily linked to code written in other languages, such as FORTRAN.

5.2 Data structure

The data structure is the most important part of the implementation code. It should be flexible for calculation, and easy for further expansion.

5.2.1 Region description

We start from the description of a vertex in any dimension. The details of a vertex structure in C/C++ are given in Appendix A.1. Figure 5.1 shows the runtime memory image of a vertex structure: Here is the way to create a vertex structure.

```
vertex = (VERTEX *)malloc(sizeof(VERTEX) + nDim * sizeof(double));
......
vertex->coord = (double *)((char *)vertex + sizeof(VERTEX));
```

The entire structure can be created at one time. This is more efficient than to create the main structure and the coordinate data area separately. In fact, in the implementation, all variable structures are created this way, except for some members whose size can not be determined at their creation time.

Second, we look at the structure of a face. The details of a face structure in C/C++ are

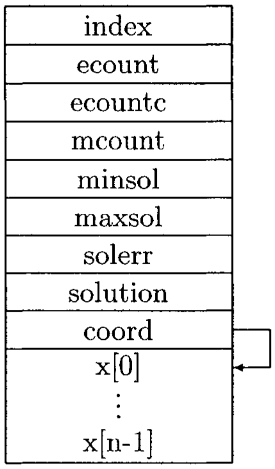


Figure 5.1: The runtime memory image of a vertex structure.

given in Appendix A.2. Figure 5.2 shows the runtime memory image of a face structure.

In this figure, $p = nmatp$, which denotes the number of matching points on this face. The member "region" has two pointers, which point to the two regions sharing this face. When the face is a part of the boundary, one pointer of "region" pointing to the outside region will be NULL. The member "next" has two pointers, which point to the two subdivisions of the face. If there is no further subdivision, these two pointers will be NULL.

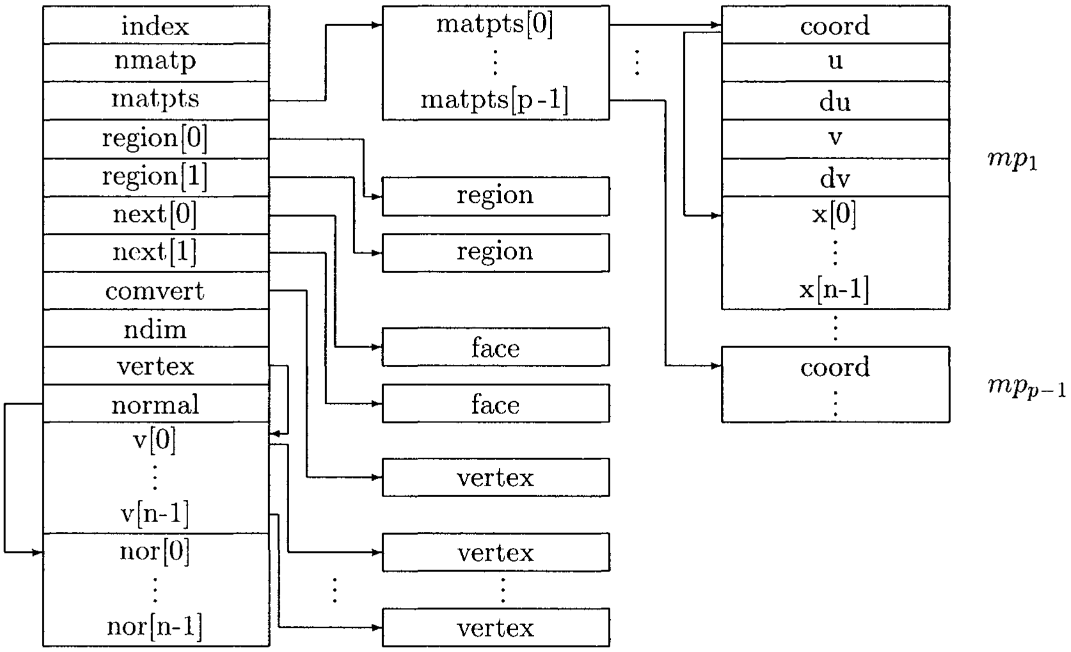


Figure 5.2: The runtime memory image of a face structure.

Third, we look at the structure of a region. The details of a region structure in C/C++ are given in Appendix A.3. Figure 5.3 shows the runtime memory image of a region structure. In this figure, the sizes of the arrays are determined by the input parameters. Here, we only use some symbols, such as nm, nc and ns, for clear explanation. The member "bgdata" points to the structure storing the information for local triangular regions. The member "egdata" points to the structure storing information for elements. The member "ngdata" points to the structure storing the information that is necessary for nested dissection. A region data structure will not have all valid pointers to each of these three structures at the same time. Therefore, we use dashed arrows to denote them.

Generally, matrices can not be directly allocated in C/C++. They can only be defined as variables, and the number of columns also need to be known. Since many matrices are needed during the calculation process, dynamic allocation is very useful in the implementation. For example, the member "colpts" of "egdata" points to a matrix storing the coordinates of the collocation points of an element. The number of collocation points will only be known when the application gets it from a configuration file. Therefore, the dimension of this matrix will be variable for different input values. For convenience of dynamic allocation, the function Alloc2D in Appendix B.1.1 will allocate a $n1 \times n2$ matrix. When accessing an item of the matrix, it is not necessary to calculate its position, since we can directly find it through the pointers. Some other matrices, such as $(A|B)$, are allocated together through the function AllocMulCol2D in Appendix B.1.2, for conveniently calculating (2.19).

5.2.2 Binary tree

The recursive binary tree mentioned in Section 2.2 is generated based on a region structure as one node, shown in Figure 5.3. In fact, the binary tree is adaptable. It can represent a very general domain, as long as the domain can be subdivided into two parts. For example, in the $2D$ case, the domain can be any polygon. When the binary tree is created, it looks like the one shown in Figure 5.4.

Here, dashed lines mean that there are several levels of nodes. Each intermediate node must have two children. It can be a locally divided triangular region, or even a combined region

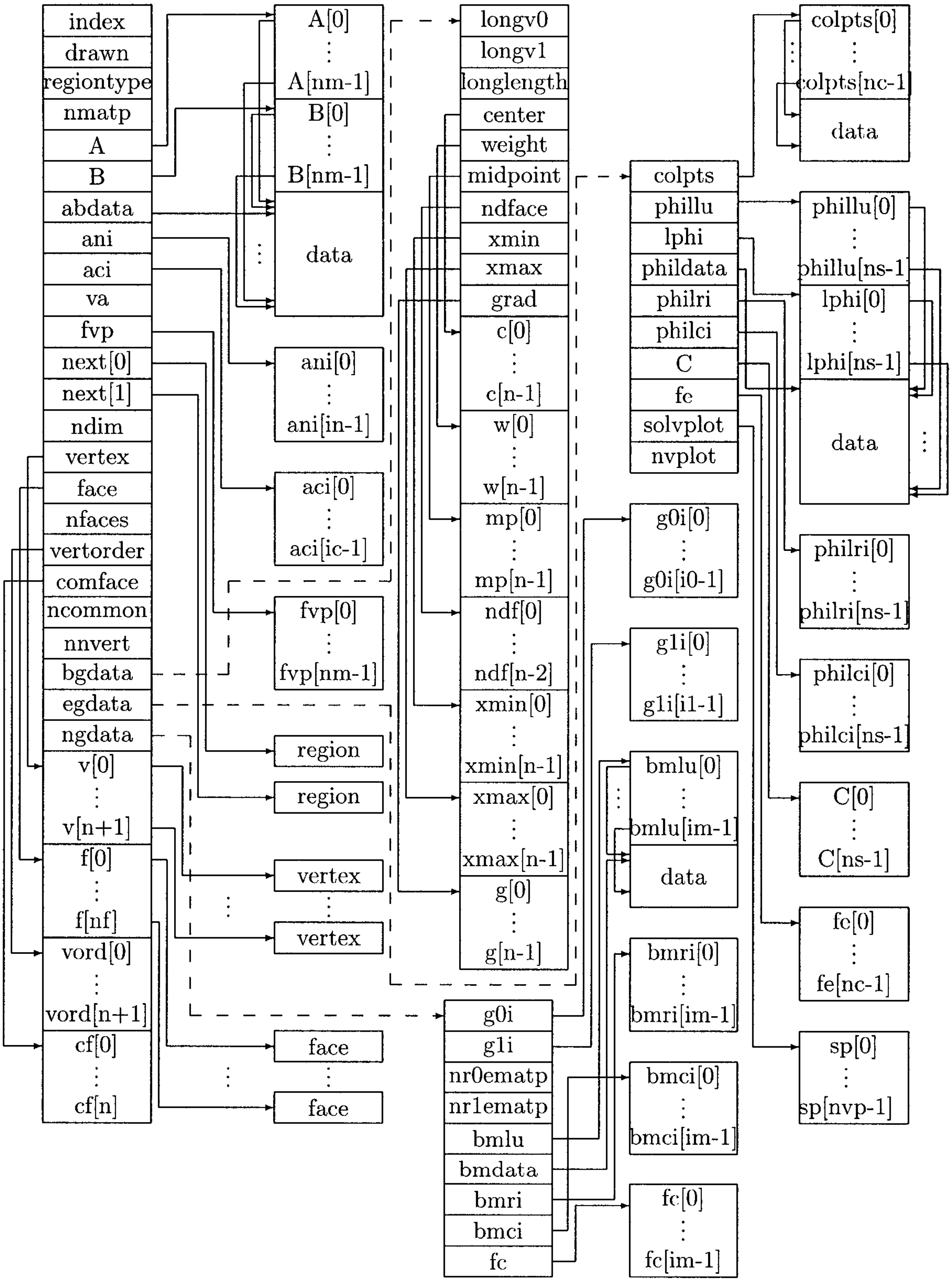


Figure 5.3: The runtime memory image of a region structure

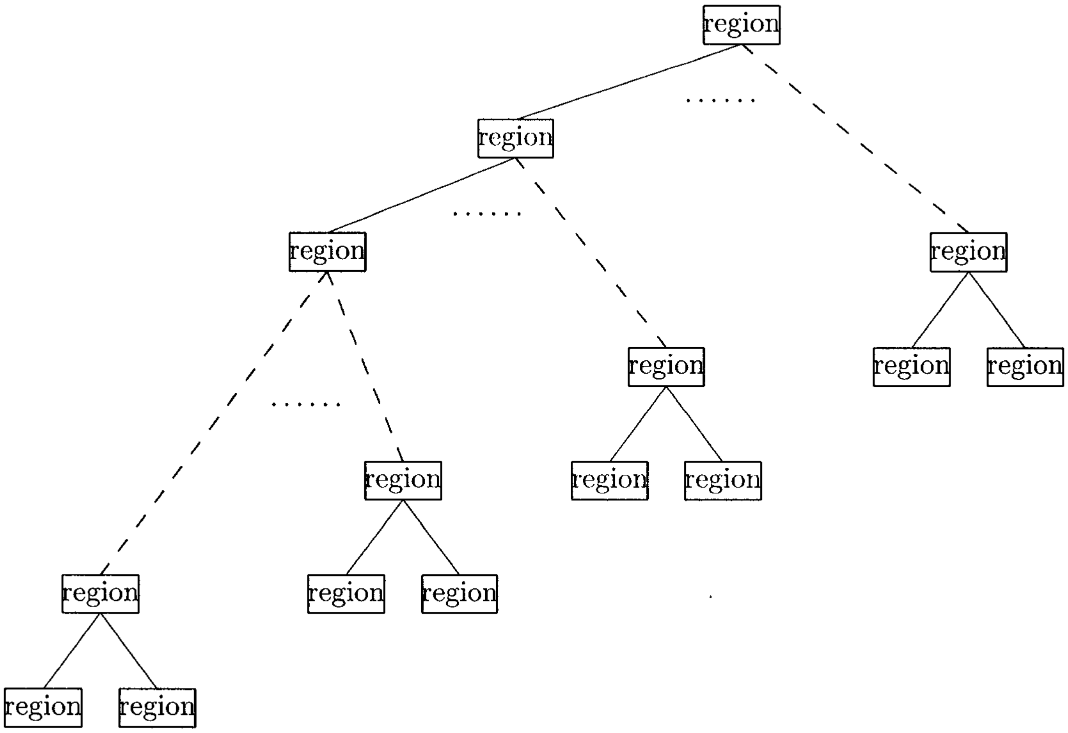


Figure 5.4: The structure of a binary tree.

which is formed by a polygon and a triangle. The first few levels are created manually, the lower levels are generated through the Rivara algorithm. This will be discussed in Section 5.3.

5.2.3 Input parameters

Parameters that set the number of matching points on each face, the number of collocation points in each element, time, original domain description, PDE problem module, *etc.*, are given in a configuration file. The calculation application will first read them into a parameter structure. The details of this structure in C/C++ are given in Appendix A.4.

5.3 Region definition

Since the domain of PDEs can be very general, it is not easy to find a suitable way to describe a domain of any shape. The present implementation in this thesis can deal with any polygon in $2D$ space.

5.3.1 A local triangular region

In this thesis, each local region in $2D$ space must be a triangle. This way, the Rivara algorithm can be used for further refinement. Therefore, there must be a definition of a local triangular region that gives the information on vertices, faces and their relations. Figure 5.5 shows a local region and its labels. First, we label the three vertices in anticlockwise direction, and we label the opposite face with the same number. Then we subdivide, if necessary, and we label the new regions as in Figure 5.6. The vertices and faces of a child region are labeled in the same way as above. The original vertex 2 is still vertex 2 in each child region. The newly generated midpoint of the longest edge, which is face 2, is vertex 1 in the left child region and vertex 0 in the right child region. The original face 2 now has two children, face 2 of the left child region, and face 2 of the right child region. The common face is face 0 in the left child region and face 1 in the right child region. There are two other cases, namely, where face 0 or face 1 is subdivided. The labeling for these two cases is the same.

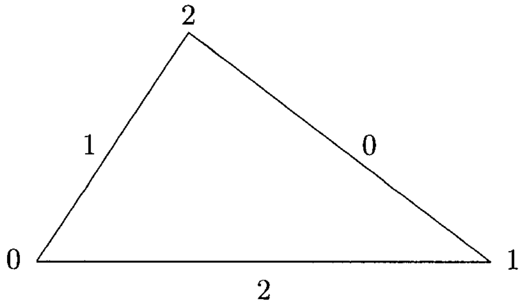


Figure 5.5: A local region and its labels.

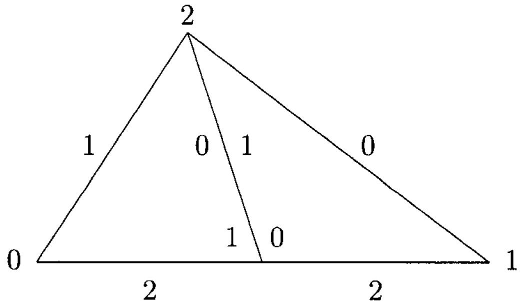


Figure 5.6: A subdivided local region and its labels.

5.3.2 The original domain of a simple polygon

The original domain will not always be a triangle. Often, the domain is a polygon. In such a case, we divide the polygon into several triangles, and we label the vertices and the faces. For the sake of simplicity, assume that each local region shares only one common face with its sibling. Figure 5.7 shows the relation of the first two triangles and their labels .

There is always a way to label the vertices and the faces to let the common face be face

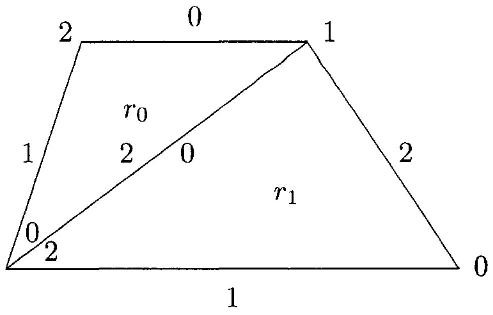


Figure 5.7: A combined region r_2 containing two triangles.

2 in r_0 and face 0 in r_1. Together r_0 and r_1 form a quadrilateral r_2. We continue to relabel the faces of r_2. Then r_2 can be combined with the next triangle r_3, as shown in Figure 5.8. We assume r_2 shares face 2, which is face 1 in r_1, with r_3. If the common face is any face

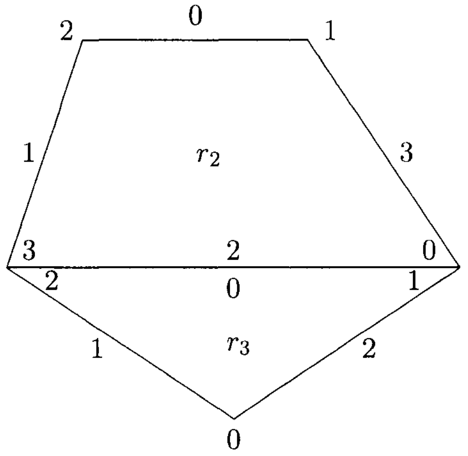


Figure 5.8: A combined region r_4 containing one quadrilateral and one triangle.

other than 2, then the processing way will be the same. Now r_2 and r_3 form a pentagon r_4. For r_2, the labels of the vertices are still given in Figure 5.8. But in fact, they will not influence the calculation process. On the other hand, the labels of the faces are very important, since they will determine the order of the matching points later in the nested dissection process. Note that we always want to label the newly added triangle so that face 0 is the common face, in order to make the process simple. Now, if the pentagon r_4 is the full original domain, then the mesh has been completed. If there are still other triangles, we continue the process until all the triangles have been added and the original domain, a polygon, has been formed. The corresponding original binary tree will be like that shown in Figure 5.9.

5.3.3 The original domain of a more complex polygon

If we suppose that the original domain is a hexagon, then it is possible to divide it into four triangles, where each triangle shares one common face with its sibling, as shown in Figure 5.10. However, it is better to divide the domain into six triangles, since this produces better angles, as Figure 5.11 shows.

For the case of Figure 5.11, we can still create a binary tree, in which each of the first five triangles shares only one common face with its siblings. However, the last triangle has two common faces, as Figure 5.12 shows.

The present implementation takes this into account. Figure 5.13 shows an example that applies local refinement, as seen in Figure 3.7, but this time the domain is a right hexagon with the length of each edge being 0.5.

In this example, the left border of the hexagon is at $x = 0$. Each line drawn in the Figure 5.13 is an approximation, since the width of the domain, $0.5 \times \sqrt{3}$, is an irrational number.

For this problem, we can still use the strategy of one common face. For some other cases, we have to use two common faces. For example, if the original domain contains a hole, then there is no way to use only one common face. Section 6.6 of the thesis provides an example that will give information pertaining to this.

5.4 Basis generation

There are ways to generate the basis of power functions in any dimension. In the collocation method, we need to know the exponents and coefficients of the basis, the first order derivatives and the second order derivatives. A suitable strategy for this is to use a recursive function to calculate the exponent of each coordinate and the coefficient of each basis. The implementation functions are given in Appendix B.2.

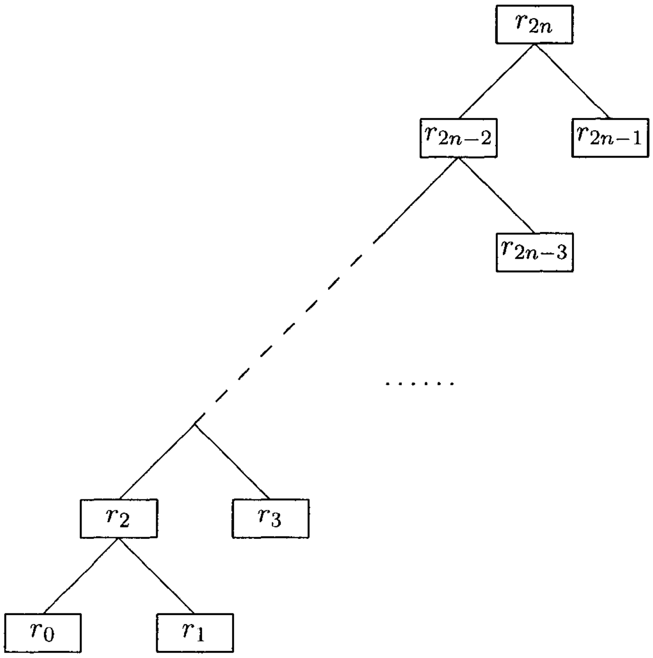


Figure 5.9: The original binary tree of a polygon domain.

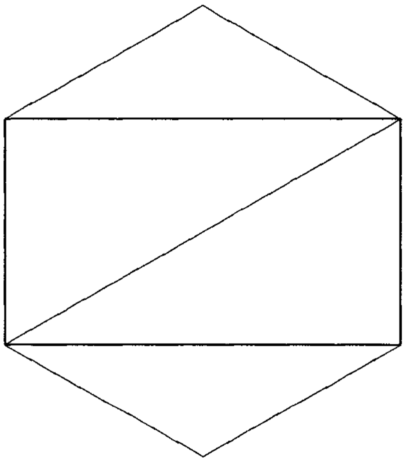


Figure 5.10: A hexagon divided into four triangles.

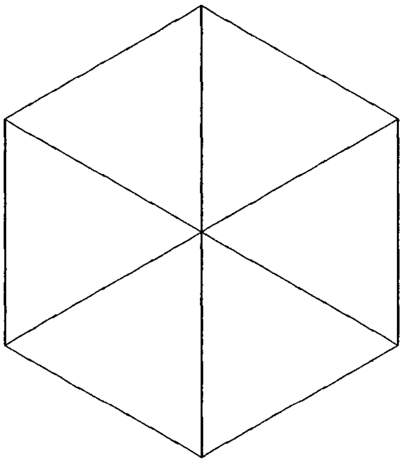


Figure 5.11: A hexagon divided into six triangles.

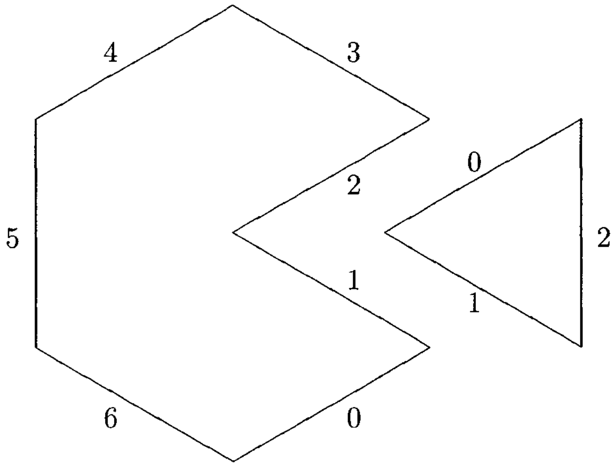


Figure 5.12: The labels of a hexagon divided into six triangles.

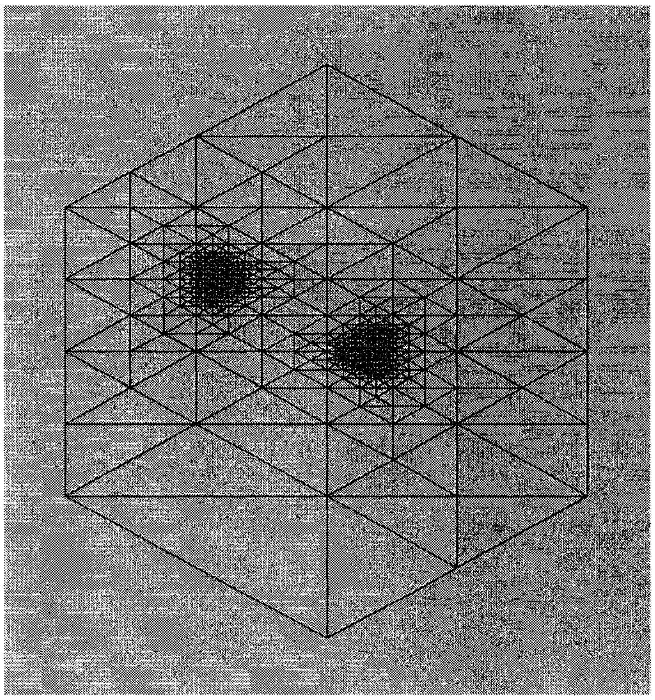


Figure 5.13: Local refinement near two points of hexagon Ω, $d_m = 0.015$, $d_r = 0.05$.

5.5 Selection of the matching points and the collocation points

There are many ways to select the matching points and the collocation points. One way to select the matching points is to use equal intervals, which gives uniformly spaced points. Another way is to select Gauss points. Numerical results show that Gauss points for square element regions can produce higher accuracy than other selection strategies. In the implementation code, we allow uniform points or Gauss matching points. The selection of the collocation points in a triangular element presents a problem, since there is no easy known strategy like that for square elements, which uses the intersections of lines connecting the pairs of opposite matching points to be the collocation points. We have used various strategies, including random collocation points, to see if the solutions are accurate. In some cases, inappropriate selection can make the matrix $(\Phi|L_\Phi)$ singular. Therefore, we may have to select many possible choices of collocation points in order to find stable ones. The procedures used in the implementation are presented in Section 6.1.

5.6 Matrix calculations

Since matrix addition, multiplication and transposition is fairly easy, it will not be mentioned here. The matrix calculations mentioned are Gauss elimination and matrix inversion. Matrix inversion can be done through Gauss elimination. Thus we will focus on Gauss elimination.

Gauss elimination is often used in solving systems of linear equations, a topic mentioned in many books on linear algebra [27]. The common way to perform Gauss elimination is to apply LU-decomposition first, then solve two systems of equations of triangular form. A variant of Gauss elimination is Gauss elimination with pivoting, which interchanges rows, or both, rows and columns, in order to let the pivot in the subsystem at each step be as large as possible. This avoids loss of accuracy due to large multipliers. For example, consider

$$\begin{pmatrix} 1 & 1 \\ 0.0000001 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

If we assume that the accuracy of representation is six significant decimal digits, then the solution is

$$x_0 = 1.00000E + 00, \quad x_1 = 0.00000E + 00.$$

However, this solution is not correct. If we first interchange row 1 and row 2, then the pivot at the top left corner becomes the largest. Solving this system, we get

$$x_0 = 1.00000E + 00, \quad x_1 = 1.00000E + 00,$$

which is accurate.

As the above example shows, selecting the largest pivot necessitates the interchange of rows or columns. Since the intermediate matrices are needed for later use, these interchanges will make the processing more complicated. A better way is to remember the order of the changed rows and columns, and not actually to interchange them. In the implementation, we use a pointer array for rows and a pointer array for columns to keep track of the order of the interchanged rows and columns. For example, consider solving a linear system of equations of the form:

$$\begin{pmatrix} 2 & 1 & 3 & 1 \\ 4 & 1 & 1 & 1 \\ 1 & 1 & 3 & 2 \\ 2 & 2 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 8 \\ 8 \\ 9 \end{pmatrix}.$$

We only treat one column, for simplicity. Using regular LU-decomposition, and storing the matrix L in the bottom left corner of the original matrix, omitting 1s on the diagonal of L, it produces

$$\begin{pmatrix} 2 & 1 & 3 & 1 \\ 2 & -1 & -5 & -1 \\ \frac{1}{2} & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & -7 & -7 \end{pmatrix}.$$

The solution is

$$(x_0, x_1, x_2, x_3)^* = (1, 2, 1, 1)^*.$$

Now, using Gauss elimination with pivoting, we apply LU-decomposition again. First, we set the original order in the arrays. This produces

$$\begin{array}{cc} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{cccc} 2 & 1 & 3 & 1 \\ 4 & 1 & 1 & 1 \\ 1 & 1 & 3 & 2 \\ 2 & 2 & 1 & 2 \end{array}\right) \end{array}.$$

When interchanges are needed, we only change the order in the corresponding pointer array, but the actual rows or the columns are not interchanged. After division, we write L and U separately for the sake of clarity, This produces

$$\begin{array}{ccc} & \begin{array}{cccc} 0 & 2 & 1 & 3 \end{array} & \begin{array}{cccc} 0 & 2 & 1 & 3 \end{array} \\ \begin{array}{c} 1 \\ 0 \\ 3 \\ 2 \end{array} & \left(\begin{array}{cccc} \frac{1}{2} & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{7} & \frac{10}{11} & 1 \\ \frac{1}{2} & 1 & \frac{1}{5} & 0 \end{array}\right) & \left(\begin{array}{cccc} 0 & \frac{1}{2} & \frac{5}{2} & \frac{1}{2} \\ 4 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{7}{5} & 0 & \frac{7}{5} \end{array}\right) \end{array}.$$

Now, we put L and U together to save memory:

$$\begin{array}{cc} & \begin{array}{cccc} 0 & 2 & 1 & 3 \end{array} \\ \begin{array}{c} 1 \\ 0 \\ 3 \\ 2 \end{array} & \left(\begin{array}{cccc} \frac{1}{2} & \frac{1}{2} & \frac{5}{2} & \frac{1}{2} \\ 4 & 1 & 1 & 1 \\ \frac{1}{4} & \frac{1}{7} & \frac{10}{11} & 1 \\ \frac{1}{2} & \frac{7}{5} & \frac{1}{5} & \frac{7}{5} \end{array}\right) \end{array}.$$

This time, the order of the calculated $x_i, i = (1, \cdots, 4)$ should be in the order of the columns' pointer array and the order of the equations used should be in the order of the rows' pointer array. First, we solve

$$Lg = f$$

and we get

$$(g_0, g_1, g_2, g_3)^* = (4, 8, 1, \frac{21}{5})^*.$$

Then we solve

$$Ux = g.$$

Note that the order is the order of the columns' pointer array, $(0, 2, 1, 3)$. The solution is then the same as above.

5.7 Nested dissection labeling and calculation

The nested dissection algorithm necessitates many matrix calculations. Also, it necessitates the repeating merging of two local regions into a larger region. Since the order of the matching points is very important, the process should be arranged carefully. Starting from the two elements shown in Figure 5.14, we assume that there are two matching points on each face, and we allow the number of collocation points to be variable. We merge the two elements as follows:

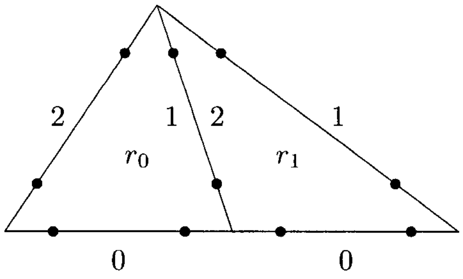


Figure 5.14: Two sibling elements and their matching points.

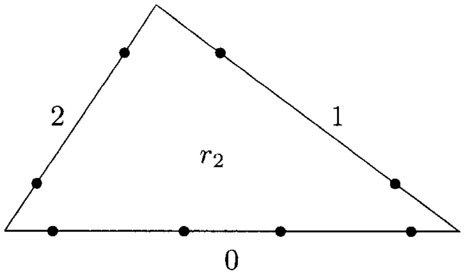


Figure 5.15: A combined local region and its matching points.

Here, the collocation points are not indicated, since their number is not important in the process. First, we label the matching points of each element according to the local order of its faces. From (4.1), (4.2) and (4.3), we can see that the order of f_{ck} is the same as the order of v_k, which is also the order of the matching points. The order of the matching points on each face starts from the vertex with the smallest global index and ends with the vertex with the largest global index. This will cause the labels to have a unique order. The labels of the matching points of r_0 are $(0,1,2,3,4,5)$. Separate the ones of the common face as l_0^c and the rest as l_0^n. Thus

$$l_0^c = (2,3), \quad l_0^n = (0,1,4,5).$$

Those of r_1 are

$$l_1^c = (4,5), \quad l_1^n = (0,1,2,3).$$

Also we label the matching points of the combined local region r_2 in Figure 5.15, *i.e.*, $(0,1,2,3,4,5,6,7)$. We separate the matching points of r_0 from the ones of r_1. We give the order of r_0 as g_0 and the order of r_1 as g_1. We assume that the order of face 0 is from left to right. Then g_0 and g_1 are

$$g_0 = (2,3,4,5), \quad g_1 = (0,1,6,7).$$

Now, according to (4.2), (4.3), (4.4) and (4.5), we divide the matrix A of r_0 and the one of r_1 into $A1$, A_{12}, B_1,B_{12}, A_2, A_{21}, B_2, B_{21}. They are located in matrix A, as shown in Figure 5.16.

Next, refering to (4.11), we first allocate memory for the matrix A of r_2, and determine the locations of C_{11}, C_{12}, C_{21} and C_{22}. Their locations are shown in Figure 5.17. Then we calculate B_m according to (4.7). We define

$$T_1 = A_{12}B_m^{-1}, \quad T_2 = A_{21}B_m^{-1}.$$

For these equations, we need to obtain the matrix inverse of B_m. However, matrix inversion will take much calculation. We can rewrite the above equations in the form of

$$T_1 B_m = A_{12}, \quad T_2 B_m = A_{21}. \tag{5.1}$$

Equations (5.1) are systems of linear equations, but the unknown variables T_1 and T_2 are on the left side. We can still use LU-decomposition to solve these two equations. We apply LU-decomposition to the matrix B_m. Then we allocate T_1 and T_2 and solve the triangular

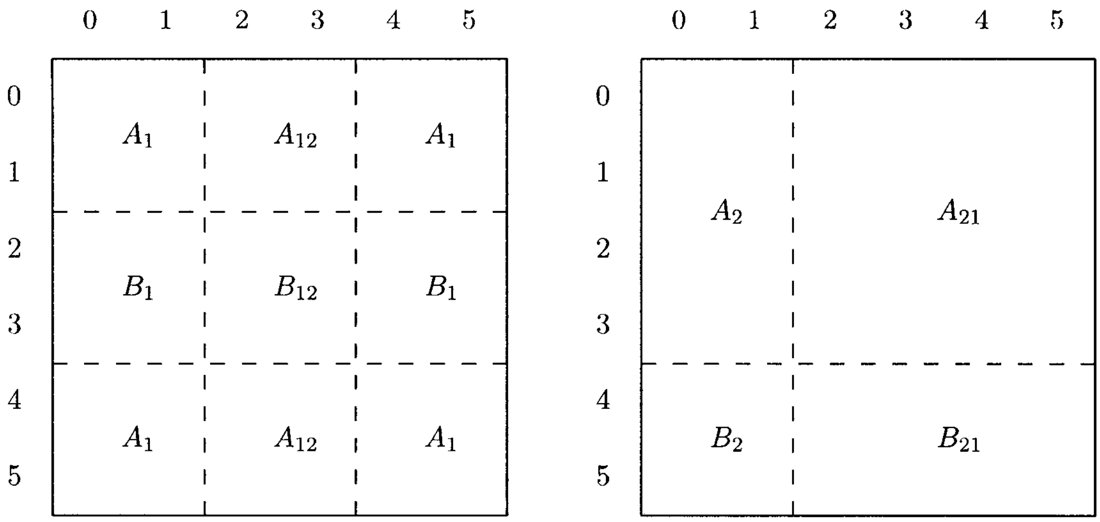


Figure 5.16: Matrix A division of r_0 and the division of r_1.

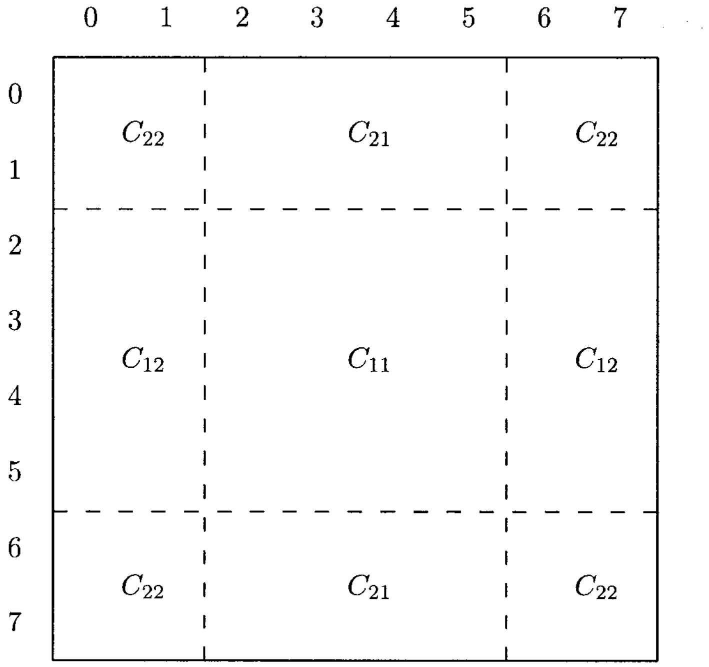


Figure 5.17: Matrix A division of r_2.

form equations, although they are different from Section 5.6. We assume

$$B_m = L_m U_m, \tag{5.2}$$

then (5.1) becomes

$$T_1 L_m U_m = A_{12}, \quad T_2 L_m U_m = A_{21}.$$

For example, T_1 can be solved from

$$\begin{aligned} GU_m &= A_{12}, \\ T_1 L_m &= G. \end{aligned}$$

Thus,

$$C_{11} = A_1 - T_1 B_1, \quad C_{12} = -T_1 B_2,$$
$$C_{21} = -T_2 B_1, \quad C_{22} = A_2 - T_2 B_2.$$

Also, we calculate f_{ck}^m, to get

$$\begin{aligned} f_{ck}^{s1} &= f_{ck}^1 - T_1 f_{ck}^m, \\ f_{ck}^{s2} &= f_{ck}^2 - T_2 f_{ck}^m. \end{aligned}$$

To calculate C_{11}, first calculate $T_1 B_1$ and save the result in the location of C_{11}. Then we subtract it from A_1 to get C_{11}. The matrices C_{12}, C_{21}, C_{22}, and the vectors f_{ck}^{s1} and f_{ck}^{s2} are calculated in a similar way.

The LU-decomposition of the matrix B_m must be kept in memory for the later back substitution. Most other matrices also need to be kept for the next time step, if they do not change. This will save much calculation time.

We continue with the recursive nested dissection process. Although the number of rows and columns of the matrix A becomes larger, the process will be the same.

5.8 Applying the boundary conditions

After recursive nested dissection, we get a linear system of equations of the form:

$$v_k = Au_k + f_{ck}.$$

We assume that there are n_b matching points on the boundary. Then u_k and v_k are both vectors of n_b items. There are $2n_b$ variables in total. But there are only n_b equations, that cannot determine all the variables. For this, we now need to use the boundary conditions.

In this thesis, we keep things simple. Since there are various kinds of boundary conditions, including nonlinear boundary conditions, solving such equations will be more difficult. We only consider the linear cases mentioned in Section 2.1. They are the Dirichlet boundary conditions and the Neumann boundary conditions, as in (2.7), (2.8) and (2.9). The boundary conditions at n_b matching points give n_b equations. If we put these n_b equations and the previous n_b equations together, we will get a system of the form,

$$M_1 u_k + M_2 v_k + h = 0.$$

Here M_1 and M_2 are $2n_b \times n_b$ matrices, and h is a $2n_b$ vector.

Now we can use LU-decomposition to solve this system for all u_k and v_k on the boundary. If the boundary conditions are only Dirichlet type, then we only need to solve for u_k, and it is not necessary to solve for v_k.

After solving for u_k, we can use (4.8) to calculate u_k of the matching points on the common face. We rewrite (4.8) as

$$B_m u_k^{12} = -(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^{12} + f_{ck}^{21}).$$

Note that the order of u_k^1 is g_0, the order of u_k^2 is g_1, the order of f_{ck}^{12} is l_0^c and the order of f_{ck}^{21} is l_1^c. Here g_0, g_1, l_0^c, l_1^c refer to the full domain and its two children. Let

$$f_{ck}^m = f_{ck}^{12} + f_{ck}^{21},$$

so that

$$B_m u_k^{12} = -(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^m).$$

Since B_m has already been decomposed, as given in (5.2), we can use it directly. Then

$$L_m U_m u_k^{12} = -(B_1 u_k^1 + B_2 u_k^2 + f_{ck}^m).$$

Given this, we only need to solve the triangular forms. This will also save much calculation time. Recursively, using (4.8) to do back substitution, we can solve for all u_k at the matching points on each common face. All the values of u_k at the interior matching points have then been obtained.

5.9 Calculation of local polynomials

If we only want to know the solution value at each matching point, then they have already been found. However, if we want to know the value at other points, then it is necessary to calculate the coefficients of the local polynomial of each element. In this thesis, since we solve parabolic PDEs, we need to know the value at each collocation point also. We can get these values from the local polynomials. Also, we want to know the values at the vertices of each element for graphical display of the solutions.

In each element, using (2.25), we can form a linear system, and use LU-decomposition to solve it. For the coefficients c_k, since LU-decomposition of $(\Phi | L_\Phi)$ has already been done when we compute the matrix $(A|B)$, we can reuse it. Let $c_i^k = c_i(t_k), i = 1, 2, \cdots, n+m$. We can get the value u_k^s at a certain point x_k^s of the element through

$$u_k^s = \sum_{i=1}^{n+m} c_i^k \phi(x_k^s).$$

For each vertex of an element, there can be several values, obtained from the different local polynomials of the elements sharing this vertex. They are kept and used for drawing pictures. These values may be also useful for future study of mesh adaption strategies.

5.10 Initial conditions and time step

We can use the initial conditions (2.6) to get initial u values at the collocation points of the first time step, and then compute the u values at the matching points of the second time step based on (2.12). Then we need to know the u values at the collocation points of the second time step, which must be done by calculating the local polynomial of each element. Using these u values and (2.12) to update f_k, we can then solve for the u values at the matching points of the third time step. We continue to do this until the last time step has been reached.

If the mesh does not change, then the intermediate matrices need not be recalculated. But if the mesh changes, each matrix should be recalculated at each time step. At present, we only consider a static mesh. Adaptive meshes that change in time are a topic for future study.

5.11 Deployment

5.11.1 Flexible design

Code deployment is important when designing programs. Good deployment can make programs very flexible. In Section 5.2, we have already solved the problem of how to design suitable data structures without recompiling the code, when some parameters change, such as the number of matching points on each element face. A flexible implementation can do calculations directly without recompiling and linking the source code, when the PDE changes. Also, the core part of the calculation code can be reused in different environments, without recompiling and linking. For example, when doing calculations, we only need to output some solutions to the console. When plotting, we may let the graphics tool directly access the core part of the calculation and plot pictures concurrently.

5.11.2 Code deployment

In order to accomplish the above, we need to divide the code into several parts and put these into modules. Figure 5.18 shows the structure of the deployment.

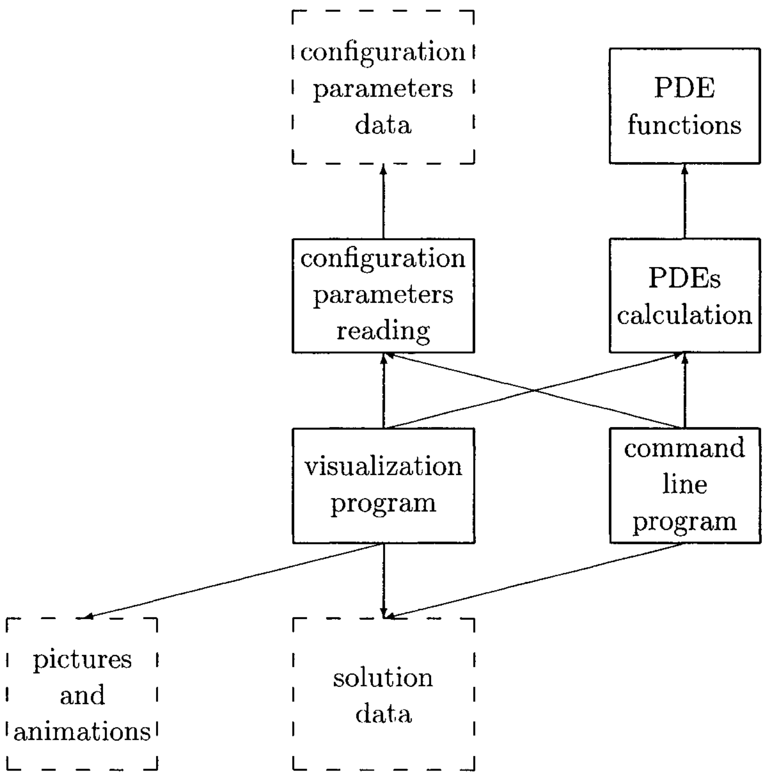


Figure 5.18: Code deployment structure.

Here, dashed boxes denote input or output files. Other boxes denote applications or modules. We can realize “configuration parameters reading”, “PDEs calculation” and “PDE functions” as modules.

Different operating systems have different kinds of module files. Module files are always dynamically loaded after the program starts running. For most Unix systems, a module file is a file with the extension “.so”, called a “shared library”. Linux also uses this style. On the other hand, for Windows systems, a module file is a file with the extension “.dll” called a “dynamic-load library”. The modules “configuration parameters reading” and “PDEs calculation” will always be used, so they can be linked directly. But the module “PDE functions”, which gives the initial conditions, the boundary conditions, some PDE-specific functions, *etc.*, will change when the PDE changes. It is better to dynamically load this module at runtime. The configuration file for a certain PDE will indicate the proper module to load.

Creating a shared library file on a Linux system is done as follows, assuming that the GNU gcc compiler is used [29, 30, 31]. First, we compile the source files:

```
gcc -shared -fPIC <compile options>
-o <.o file> <.c file>
```

Then, we link the object files to create a module file:

```
gcc -shared -fPIC -o <.so file> <link options>
<list of .o files> -l<list of libraries>
```

Loading a shared library file on a Linux system at runtime is done as follows [32]:

```
void *mod;
double (*userf)(...);
......
mod = dlopen(<.so file name>, RTLD_LAZY);
......
userf = dlsym(mod, <function name>);
......
userf(...);
......
dlclose(mod);
```

The ".so" files should be put into a library directory that can be found by the Linux systems. For other Unix systems, Desitter has written an article [34] containing detailed information. Shah and Xiao supplied another article [35].

Creating a shared library file on the Windows systems is done as follows, assuming that the Microsoft Visual C/C++ compiler is used [36].

First, we need to give a ".def" file to offer the information of the functions that need to be exported, or to give the declarations in the ".cpp" source file.

```
__declspec(dllexport) double userf(...);
```

Then, we compile the source files:

```
cl <.obj file> <compile options> /c <.c file>
```

Thereafter, we link the object files to create a module file:

```
link <.obj files> <link options> /DEF <.def file>
<list of libraries> <.dll file>
```

Loading a dynamic-link library file on the Windows systems at runtime is done as follows [37]:

```
HMODULE mod;
double (*userf)(...);
......
mod = LoadLibrary(<.dll file name>);
......
userf = GetProcAddress(mod, <function name>);
......
FreeLibrary(mod);
```

Also, the ".dll" files should be put into a library directory that can be found by the Windows systems.

Chapter 6

Numerical Results

We will first consider a simple equation whose solution is time-independent, to observe the error caused by the mesh, the number of matching points, and the number of collocation points. Thereafter we will treat some time-dependent cases.

6.1 Strategy for choosing the matching points and the collocation points

In Section 5.5, we have mentioned that inappropriate selection of the matching points and the collocation points may cause the matrix $(\Phi|L_\Phi)$ to be singular, or some pivot of the matrix to be very small. For the $2D$ case, we can select random points in or near an element as the collocation points. For the matching points we can use uniform points or Gauss points.

A point in or near a certain triangle can be determined as

$$x_p = c_0x_0 + c_1x_1 + c_2x_2, \quad c_0 + c_1 + c_2 = 1$$
$$c_0, c_1, c_2 \in R, \quad x_p, x_0, x_1, x_2 \in R^2,$$

where x_p is the coordinate of the point and x_1, x_2, x_3 are the coordinates of the three vertices of the triangle. We can also use polar coordinates to describe random points. For simplicity, some constraints are introduced.

1. Each element in a mesh uses the same number of points and the same pattern of selection.

2. The number of collocation points for each element is $3n$ or $3n+1$, where n is an integer which is greater than or equal to zero. Each group of three points is selected as given below in equation (6.1). If the number of collocation points is $3n+1$, then the last point is the center of the element.

$$\begin{aligned} x_p^{3i} &= c_0^i x_0 + c_1^i x_1 + c_2^i x_2, \\ x_p^{3i+1} &= c_2^i x_0 + c_0^i x_1 + c_1^i x_2, \quad i = 1, 2, \cdots, n \\ x_p^{3i+2} &= c_1^i x_0 + c_2^i x_1 + c_0^i x_2. \end{aligned} \tag{6.1}$$

For example, for four collocation points, a possible group of positions is shown in Figure 6.1.

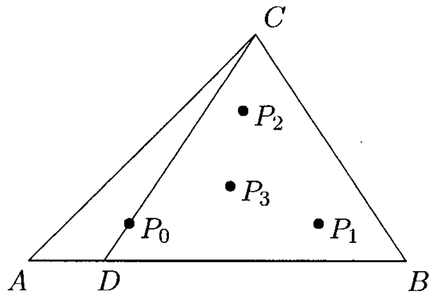


Figure 6.1: Four collocation points in an triangular element.

Thus, if $AD : DB = c_0 : c_1$, then $DP_0 : P_0C = c_2 : (c_0 + c_1)$. P_1 and P_2 are also determined in this way, only the order of A, B, C is changed.

Although each element can use a different selection pattern, tests show that this can make the calculation process non-convergent. Because of this, we select random collocation points, keeping n triples of the coefficients c_0, c_1, c_2. Then we use these coefficients to determine the collocation points for meshes of different sizes, and we observe the accuracy. Table 6.1 gives the coefficients of the collocation points selected for the test examples in Section 6.2:

Table 6.1: The coefficients of the selected collocation points

m	c_0	c_1	c_2
3	0.5739868425643008	0.2130065787178496	0.2130065787178496
4	0.1882744115722712	0.4058627942138644	0.4058627942138644
6	0.6230625503803894	0.1884687248098053	0.1884687248098053
	0.2129734373711858	0.3935132813144071	0.3935132813144071
7	0.1433732137751642	0.4283133931124179	0.4283133931124179
	0.8207992328427728	0.0896003835786136	0.0896003835786136
9	0.9322935905923572	0.0338532047038214	0.0338532047038214
	0.3034683416148034	0.3482658291925983	0.3482658291925983
	0.2721280014478266	0.3639359992760867	0.3639359992760867
10	0.0475797644106576	0.4762101177946712	0.4762101177946712
	0.7319216349776470	0.1340391825111765	0.1340391825111765
	0.8472868394326824	0.0763565802836588	0.0763565802836588

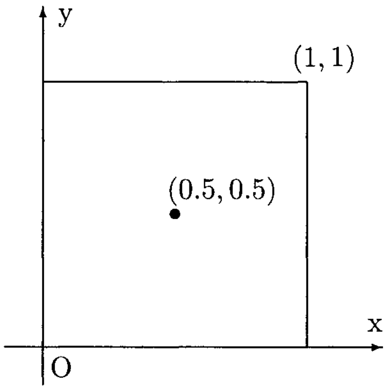


Figure 6.2: A unit square Ω.

6.2 A time-independent PDE

Consider the PDE

$$\begin{aligned} \frac{\partial u}{\partial t} &= \Delta u - 2e^{x+y}, \quad (x,y) \in \Omega \subset R^2 \\ u(x,y,0) &= e^{x+y} + 1, \quad (x,y) \in \Omega \\ u(x,y,t) &= e^{x+y}, \quad (x,y) \in \partial\Omega, \end{aligned}$$

where Ω is the unit square, with center at $(0.5, 0.5)$, as shown in Figure 6.2. The solution

Table 6.2: The total number of elements in Ω.

d_m	1/2	1/4	1/8	1/16	1/32	1/64
n_e	16	64	256	1024	4096	16384

of this PDE is time-independent, namely,

$$u(x, y, t) = e^{x+y} + 1, \quad (x, y) \in \Omega.$$

In this example we use uniform meshes of different sizes. We let d_m denote the upper limit of the longest edge of the elements, n is the number of matching point on each face, and m is the number of collocation points in each element. Table 6.2 gives the total number of elements n_e, for different d_m.

The maximum absolute errors at the matching points are given in Table 6.3 and Table 6.4, for different selection strategies for the matching points.

We also use the local polynomial of each element to calculate the solutions at the vertices. These values are used by the graphics. We can also use these values as a measure of the accuracy of the local polynomials. Since each vertex may be shared by several elements, there may be several different values. The error of the solutions at each vertex is taken to be the maximum of the errors in these values, and the maximum absolute error given in Table 6.5 and Table 6.6 is the maximum of the errors of the solutions at all vertices.

We can see that if d_m becomes smaller, then the solutions become more accurate. If the number of matching points and the number of collocation points increase, *i.e.*, n and m become larger, then the solution also becomes more accurate. Gauss points appear to give higher accuracy than uniform points. Most cases follow this pattern, but there are some exceptions. For example, in Table 6.3, when $d_m = 1/64$, the error for the case $n = 3$, $m = 6$ is smaller than the error for the case $n = 4$, $m = 7$. Since the collocation points selected here are random points, the solution may not be the most accurate, so the above phenomenon may at times occur.

Figure 6.3 gives a $3D$ plot for this PDE. Here we used $d_m = 1/16$, $n = 4$, $m = 7$.

Table 6.3: Maximum absolute error at the matching points (uniform points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=5	n=4
d_m	m=1	m=3	m=4	m=6	m=7	m=9	m=9	m=10
1/2	8.27e-02	4.08e-03	4.45e-04	1.50e-05	1.08e-05	1.70e-06	2.19e-06	2.39e-06
1/4	2.44e-02	6.25e-04	3.59e-05	7.57e-07	3.73e-07	5.25e-08	4.23e-08	1.21e-07
1/8	6.64e-03	1.46e-04	2.72e-06	4.37e-08	1.21e-08	2.55e-09	7.62e-10	7.27e-09
1/16	1.73e-03	3.63e-05	1.97e-07	2.66e-09	3.83e-10	1.48e-10	2.69e-11	4.50e-10
1/32	4.42e-04	9.02e-06	1.35e-08	1.62e-10	1.10e-10	9.31e-12	3.20e-12	2.82e-11
1/64	1.12e-04	2.26e-06	8.97e-10	9.97e-12	4.28e-10			

Table 6.4: Maximum absolute error at the matching points (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=5	n=4
d_m	m=1	m=3	m=4	m=6	m=7	m=9	m=9	m=10
1/2	8.27e-02	3.88e-03	4.04e-04	2.62e-05	2.58e-05	1.19e-06	6.58e-06	1.48e-07
1/4	2.44e-02	6.85e-04	3.18e-05	1.19e-06	1.14e-06	2.18e-08	1.64e-07	3.30e-09
1/8	6.63e-03	1.43e-04	2.23e-06	6.53e-08	4.32e-08	1.01e-09	3.53e-09	6.09e-11
1/16	1.73e-03	3.39e-05	1.55e-07	3.74e-09	1.48e-09	6.03e-11	6.42e-11	1.09e-12
1/32	4.41e-04	8.33e-06	1.07e-08	2.24e-10	5.07e-11	4.03e-12	2.22e-12	7.46e-14
1/64	1.11e-04	2.07e-06	7.14e-10	1.39e-11	5.30e-12			

Table 6.5: Maximum absolute error at the vertices (uniform points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=5	n=4
d_m	m=1	m=3	m=4	m=6	m=7	m=9	m=9	m=10
1/2	5.15e-01	6.53e-03	1.09e-03	6.25e-05	3.27e-05	1.85e-06	3.98e-06	4.74e-06
1/4	1.49e-01	7.64e-04	7.62e-05	2.20e-06	1.15e-06	5.32e-08	6.80e-08	1.58e-07
1/8	4.01e-02	1.50e-04	5.05e-06	7.32e-08	3.84e-08	2.55e-09	1.10e-09	7.89e-09
1/16	1.04e-02	3.65e-05	3.25e-07	3.44e-09	1.24e-09	1.48e-10	2.62e-11	4.59e-10
1/32	2.66e-03	9.03e-06	2.06e-08	1.86e-10	1.70e-10	9.32e-12	3.46e-12	2.83e-11
1/64	6.70e-04	2.26e-06	1.30e-09	1.07e-11	4.57e-10			

Table 6.6: Maximum absolute error at the vertices (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=5	n=4
d_m	m=1	m=3	m=4	m=6	m=7	m=9	m=9	m=10
1/2	5.15e-01	5.30e-03	6.06e-04	4.16e-05	2.36e-05	8.26e-07	7.01e-06	8.58e-07
1/4	1.49e-01	8.75e-04	4.23e-05	1.47e-06	9.34e-07	1.53e-08	1.61e-07	1.53e-08
1/8	4.01e-02	1.45e-04	2.79e-06	6.75e-08	3.29e-08	9.45e-10	3.49e-09	2.56e-10
1/16	1.04e-02	3.40e-05	1.79e-07	3.80e-09	1.09e-09	5.92e-11	6.38e-11	4.25e-12
1/32	2.65e-03	8.33e-06	1.22e-08	2.27e-10	3.53e-11	4.01e-12	2.21e-12	1.13e-13
1/64	6.70e-04	2.07e-06	8.17e-10	1.39e-11	4.81e-12			

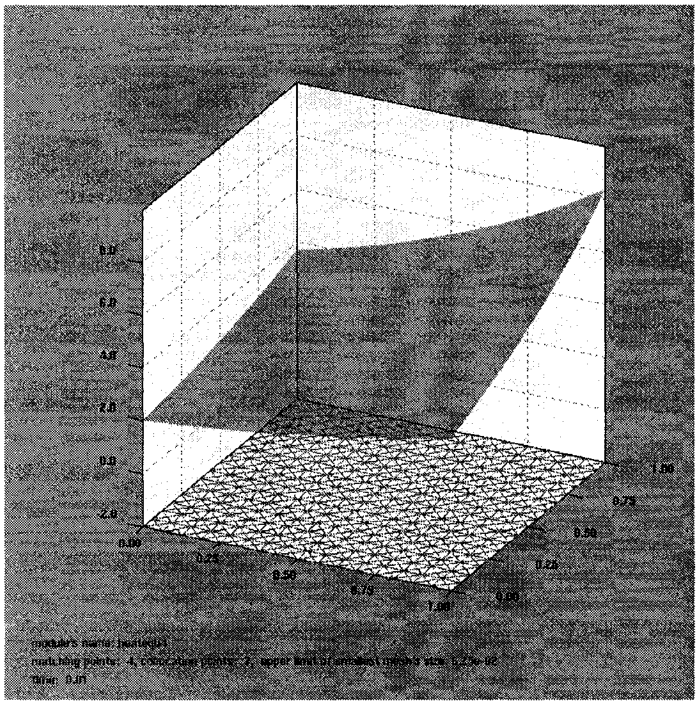


Figure 6.3: Solutions of the time-independent PDE, when $d_m = 0.0625$, $n = 4$, $m = 7$.

6.3 A time-dependent PDE

Now consider a PDE which whose solution is time-dependent,

$$\begin{aligned} \frac{\partial u}{\partial t} &= \Delta u, \quad (x,y) \in \Omega \subset R^2, t \in [0,1], \\ u(x,y,0) &= e^{x+y} + 1, \quad (x,y) \in \Omega, \\ u(x,y,t) &= e^{2t+x+y}, \quad (x,y) \in \partial\Omega, t \in [0,1]. \end{aligned}$$

where Ω is the same as in Section 6.2. The exact solution is

$$u(x,y,t) = e^{2t+x+y} + 1, \quad (x,y) \in \Omega, t \in [0,1].$$

For this PDE, we again use the collocation points selected in the previous section. Since there can be many time steps, we do not list the solutions at every time step. Instead we only give the errors at the first several steps and at the last several steps. Here Ω is the unit square and the time range is $[0,1]$. The time step is d_t. We determine the maximum error at the matching points only. Table 6.7 gives the maximum absolute errors at the matching

Table 6.7: Maximum absolute error at the matching points, d_t = 0.01 (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=4	n=5
t	m=1	m=3	m=4	m=6	m=7	m=7	m=7	m=9
d_m	1/2	1/8	1/4	1/8	1/8	1/16	1/32	1/8
0.01	5.23e-01	7.87e-04	7.84e-04	7.56e-04	7.56e-04	7.56e-04	7.56e-04	7.55e-04
0.02	5.36e-01	1.41e-03	1.38e-03	1.35e-03	1.35e-03	1.36e-03	1.36e-03	1.35e-03
0.03	5.47e-01	1.94e-03	1.84e-03	1.87e-03	1.87e-03	1.87e-03	1.87e-03	1.87e-03
0.04	5.59e-01	2.40e-03	2.30e-03	2.30e-03	2.30e-03	2.32e-03	2.32e-03	2.30e-03
0.05	5.70e-01	2.80e-03	2.71e-03	2.70e-03	2.70e-03	2.70e-03	2.71e-03	2.70e-03
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.96	3.53e+00	2.75e-02	2.62e-02	2.64e-02	2.64e-02	2.65e-02	2.65e-02	2.64e-02
0.97	3.60e+00	2.80e-02	2.67e-02	2.70e-02	2.70e-02	2.70e-02	2.70e-02	2.70e-02
0.98	3.67e+00	2.86e-02	2.73e-02	2.75e-02	2.75e-02	2.76e-02	2.76e-02	2.75e-02
0.99	3.74e+00	2.92e-02	2.78e-02	2.81e-02	2.81e-02	2.81e-02	2.81e-02	2.81e-02
1.00	3.82e+00	2.98e-02	2.84e-02	2.86e-02	2.86e-02	2.87e-02	2.87e-02	2.86e-02

points.

We see that the maximum absolute error becomes larger with increasing time. This is caused by the accumulation of the errors of each time step. Also, the second derivative with respect to time of the exact solution is an exponential function, so the value of the exact solution becomes very large with increasing time. Therefore it is better to use the maximum relative error, which is the maximum absolute error at each matching point divided by u_k. Table 6.8 gives the maximum relative errors at the matching points.

Table 6.8: Maximum relative error at the matching points, d_t = 0.01 (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=4	n=5
t	m=1	m=3	m=4	m=6	m=7	m=7	m=7	m=9
d_m	1/2	1/8	1/4	1/8	1/8	1/16	1/32	1/8
0.01	9.62e-02	1.53e-04	1.49e-04	1.48e-04	1.48e-04	1.48e-04	1.48e-04	1.48e-04
0.02	9.49e-02	2.93e-04	2.84e-04	2.83e-04	2.83e-04	2.83e-04	2.83e-04	2.83e-04
0.03	9.45e-02	4.18e-04	4.06e-04	4.03e-04	4.03e-04	4.04e-04	4.04e-04	4.03e-04
0.04	9.45e-02	5.29e-04	5.14e-04	5.11e-04	5.11e-04	5.11e-04	5.11e-04	5.11e-04
0.05	9.47e-02	6.24e-04	6.07e-04	6.03e-04	6.03e-04	6.03e-04	6.03e-04	6.03e-04
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.96	1.19e-01	1.38e-03	1.30e-03	1.33e-03	1.33e-03	1.33e-03	1.33e-03	1.33e-03
0.97	1.19e-01	1.38e-03	1.30e-03	1.33e-03	1.33e-03	1.33e-03	1.33e-03	1.33e-03
0.98	1.19e-01	1.38e-03	1.31e-03	1.34e-03	1.34e-03	1.34e-03	1.34e-03	1.34e-03
0.99	1.19e-01	1.39e-03	1.31e-03	1.34e-03	1.34e-03	1.34e-03	1.34e-03	1.34e-03
1.00	1.19e-01	1.39e-03	1.31e-03	1.34e-03	1.34e-03	1.34e-03	1.34e-03	1.34e-03

Next, we half the time, that is $d_t = 0.005$. Then there are a total of 200 time steps. The maximum absolute errors are listed in Table 6.9, and the maximum relative errors are listed in Table 6.10. We half the time step again, so the number of time steps will be 400. The corresponding maximum absolute errors and the maximum relative errors are listed in Table 6.11 and Table 6.12, respectively.

We see that the accuracy of this problem is much lower than that of the previous time-independent problem, and the maximum absolute error is about $O(dt)$. Also note that when we add more matching points and collocation points, or reduce the size of the elements, the accuracy is not improved. It seems that the maximum absolute error does not change. Each time, when the time step is reduced by one half, the error will also be reduced to almost one half. Thus we observe that the error depends on the size of time step. When the part of the error caused by the space mesh becomes very small, the main part of the error will be caused by the time step size. Only when the space mesh is very coarse, will the error be mainly caused by the mesh. This can be seen from Table 6.7 when $n = 1$, $m = 1$ and $d_m = 1/2$. We will also give another example to demonstrate this more clearly. Figure 6.4 shows a $3D$ plot of the solution at the 10th step, when $t = 0.10$, and Figure 6.5 shows the $3D$ plot of the solution at the last step, when $t = 1.00$.

Since the main part of the error is caused by the size of the time step, when it is large, we want to reduce the time step size to a very small quantity, to see what will happen. This time we use $d_t = 10^{-6}$, and a time range $[0, 10^{-4}]$. We select new collocation points to give more precise solutions. Table 6.13 shows the coefficients of the new selected collocation points. The maximum absolute errors of these cases are listed in Table 6.14.

Now we see that the error hardly changes when the space mesh is coarse, and m, n are small. But when the space mesh becomes denser, the error changes with time. Thus we see that the main part of the error is now caused by the space mesh and the number of matching points, as well as the number of collocation points.

Table 6.9: Maximum absolute error at the matching points, $d_t = 0.005$ (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=4	n=5
t	m=1	m=3	m=4	m=6	m=7	m=7	m=7	m=9
d_m	1/2	1/8	1/4	1/8	1/8	1/16	1/32	1/8
0.005	8.19e-02	2.40e-04	2.25e-04	2.19e-04	2.19e-04	2.19e-04	2.19e-04	2.18e-04
0.010	8.21e-02	4.35e-04	4.10e-04	4.03e-04	4.04e-04	4.04e-04	4.04e-04	4.02e-04
0.015	8.33e-02	6.10e-04	5.74e-04	5.66e-04	5.66e-04	5.66e-04	5.66e-04	5.65e-04
0.020	8.46e-02	7.64e-04	7.16e-04	7.13e-04	7.13e-04	7.13e-04	7.13e-04	7.12e-04
0.025	8.58e-02	9.08e-04	8.44e-04	8.47e-04	8.47e-04	8.47e-04	8.48e-04	8.47e-04
0.030	8.70e-02	1.04e-03	9.66e-04	9.72e-04	9.72e-04	9.72e-04	9.72e-04	9.71e-04
0.035	8.80e-02	1.16e-03	1.08e-03	1.09e-03	1.09e-03	1.09e-03	1.09e-03	1.09e-03
0.040	8.90e-02	1.28e-03	1.19e-03	1.20e-03	1.20e-03	1.20e-03	1.20e-03	1.19e-03
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.965	5.67e-01	1.44e-02	1.34e-02	1.34e-02	1.34e-02	1.34e-02	1.34e-02	1.34e-02
0.970	5.72e-01	1.45e-02	1.35e-02	1.35e-02	1.36e-02	1.36e-02	1.36e-02	1.35e-02
0.975	5.78e-01	1.46e-02	1.36e-02	1.37e-02	1.37e-02	1.37e-02	1.37e-02	1.37e-02
0.980	5.84e-01	1.48e-02	1.38e-02	1.38e-02	1.38e-02	1.38e-02	1.38e-02	1.38e-02
0.985	5.90e-01	1.49e-02	1.39e-02	1.40e-02	1.40e-02	1.40e-02	1.40e-02	1.40e-02
0.990	5.96e-01	1.51e-02	1.41e-02	1.41e-02	1.41e-02	1.41e-02	1.41e-02	1.41e-02
0.995	6.02e-01	1.52e-02	1.42e-02	1.42e-02	1.43e-02	1.43e-02	1.43e-02	1.42e-02
1.000	6.08e-01	1.54e-02	1.43e-02	1.44e-02	1.44e-02	1.44e-02	1.44e-02	1.44e-02

Table 6.10: Maximum relative error at the matching points, $d_t = 0.005$ (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=4	n=5
t	m=1	m=3	m=4	m=6	m=7	m=7	m=7	m=9
d_m	1/2	1/8	1/4	1/8	1/8	1/16	1/32	1/8
0.005	1.43e-02	4.18e-05	4.12e-05	3.90e-05	3.90e-05	3.90e-05	3.90e-05	3.90e-05
0.010	1.37e-02	8.15e-05	7.81e-05	7.61e-05	7.60e-05	7.61e-05	7.61e-05	7.60e-05
0.015	1.34e-02	1.19e-04	1.13e-04	1.12e-04	1.12e-04	1.12e-04	1.12e-04	1.11e-04
0.020	1.32e-02	1.55e-04	1.45e-04	1.45e-04	1.45e-04	1.45e-04	1.45e-04	1.45e-04
0.025	1.33e-02	1.89e-04	1.77e-04	1.78e-04	1.78e-04	1.78e-04	1.78e-04	1.78e-04
0.030	1.33e-02	2.21e-04	2.08e-04	2.08e-04	2.08e-04	2.08e-04	2.08e-04	2.08e-04
0.035	1.33e-02	2.52e-04	2.37e-04	2.36e-04	2.36e-04	2.36e-04	2.36e-04	2.36e-04
0.040	1.33e-02	2.80e-04	2.64e-04	2.63e-04	2.63e-04	2.63e-04	2.63e-04	2.63e-04
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.965	1.62e-02	7.15e-04	6.68e-04	6.69e-04	6.69e-04	6.69e-04	6.69e-04	6.69e-04
0.970	1.62e-02	7.16e-04	6.69e-04	6.69e-04	6.69e-04	6.69e-04	6.69e-04	6.69e-04
0.975	1.62e-02	7.16e-04	6.69e-04	6.69e-04	6.70e-04	6.70e-04	6.70e-04	6.69e-04
0.980	1.62e-02	7.16e-04	6.69e-04	6.70e-04	6.70e-04	6.70e-04	6.70e-04	6.70e-04
0.985	1.62e-02	7.17e-04	6.70e-04	6.70e-04	6.70e-04	6.70e-04	6.70e-04	6.70e-04
0.990	1.62e-02	7.17e-04	6.70e-04	6.71e-04	6.71e-04	6.71e-04	6.71e-04	6.70e-04
0.995	1.62e-02	7.18e-04	6.70e-04	6.71e-04	6.71e-04	6.71e-04	6.71e-04	6.71e-04
1.000	1.63e-02	7.18e-04	6.71e-04	6.71e-04	6.71e-04	6.71e-04	6.71e-04	6.71e-04

Table 6.11: Maximum absolute error at the matching points, $d_t = 0.0025$ (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=4	n=5
t	m=1	m=3	m=4	m=6	m=7	m=7	m=7	m=9
d_m	1/2	1/8	1/4	1/8	1/8	1/16	1/32	1/8
0.0025	8.23e-02	7.84e-05	7.28e-05	6.18e-05	6.17e-05	6.18e-05	6.18e-05	6.09e-05
0.0050	8.14e-02	1.37e-04	1.26e-04	1.16e-04	1.16e-04	1.16e-04	1.16e-04	1.15e-04
0.0075	8.15e-02	1.93e-04	1.75e-04	1.65e-04	1.65e-04	1.65e-04	1.65e-04	1.64e-04
0.0100	8.20e-02	2.42e-04	2.20e-04	2.10e-04	2.10e-04	2.10e-04	2.10e-04	2.09e-04
0.0125	8.27e-02	2.91e-04	2.62e-04	2.52e-04	2.52e-04	2.52e-04	2.52e-04	2.52e-04
0.0150	8.35e-02	3.37e-04	3.02e-04	2.92e-04	2.92e-04	2.92e-04	2.92e-04	2.92e-04
0.0175	8.42e-02	3.79e-04	3.39e-04	3.30e-04	3.30e-04	3.30e-04	3.30e-04	3.29e-04
0.0200	8.49e-02	4.19e-04	3.74e-04	3.66e-04	3.66e-04	3.66e-04	3.66e-04	3.65e-04
0.0225	8.56e-02	4.57e-04	4.06e-04	4.00e-04	4.00e-04	4.00e-04	4.00e-04	4.00e-04
0.0250	8.62e-02	4.94e-04	4.36e-04	4.33e-04	4.34e-04	4.34e-04	4.34e-04	4.33e-04
0.0275	8.68e-02	5.31e-04	4.65e-04	4.65e-04	4.65e-04	4.65e-04	4.65e-04	4.65e-04
0.0300	8.73e-02	5.66e-04	4.93e-04	4.96e-04	4.96e-04	4.96e-04	4.96e-04	4.95e-04
0.0325	8.78e-02	5.99e-04	5.19e-04	5.25e-04	5.26e-04	5.25e-04	5.26e-04	5.25e-04
0.0350	8.83e-02	6.31e-04	5.49e-04	5.54e-04	5.54e-04	5.54e-04	5.54e-04	5.54e-04
0.0375	8.88e-02	6.61e-04	5.79e-04	5.81e-04	5.82e-04	5.82e-04	5.82e-04	5.81e-04
0.0400	8.93e-02	6.90e-04	6.07e-04	6.08e-04	6.08e-04	6.08e-04	6.08e-04	6.08e-04
0.0425	8.97e-02	7.19e-04	6.34e-04	6.34e-04	6.33e-04	6.34e-04	6.34e-04	6.33e-04
0.0450	9.02e-02	7.48e-04	6.61e-04	6.59e-04	6.59e-04	6.59e-04	6.59e-04	6.58e-04
0.0475	9.07e-02	7.76e-04	6.86e-04	6.82e-04	6.83e-04	6.83e-04	6.83e-04	6.82e-04
0.0500	9.11e-02	8.03e-04	7.10e-04	7.06e-04	7.06e-04	7.06e-04	7.06e-04	7.06e-04
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.9525	5.54e-01	7.44e-03	6.54e-03	6.55e-03	6.56e-03	6.56e-03	6.56e-03	6.55e-03
0.9550	5.57e-01	7.48e-03	6.57e-03	6.58e-03	6.59e-03	6.59e-03	6.59e-03	6.58e-03
0.9575	5.60e-01	7.52e-03	6.61e-03	6.61e-03	6.62e-03	6.62e-03	6.62e-03	6.62e-03
0.9600	5.62e-01	7.56e-03	6.64e-03	6.65e-03	6.65e-03	6.65e-03	6.65e-03	6.65e-03
0.9625	5.65e-01	7.59e-03	6.67e-03	6.68e-03	6.69e-03	6.69e-03	6.69e-03	6.68e-03
0.9650	5.68e-01	7.63e-03	6.71e-03	6.71e-03	6.72e-03	6.72e-03	6.72e-03	6.72e-03
0.9675	5.71e-01	7.67e-03	6.74e-03	6.75e-03	6.76e-03	6.76e-03	6.76e-03	6.75e-03
0.9700	5.74e-01	7.71e-03	6.77e-03	6.78e-03	6.79e-03	6.79e-03	6.79e-03	6.78e-03
0.9725	5.77e-01	7.75e-03	6.81e-03	6.82e-03	6.82e-03	6.82e-03	6.82e-03	6.82e-03
0.9750	5.80e-01	7.79e-03	6.84e-03	6.85e-03	6.86e-03	6.86e-03	6.86e-03	6.85e-03
0.9775	5.82e-01	7.83e-03	6.88e-03	6.88e-03	6.89e-03	6.89e-03	6.89e-03	6.89e-03
0.9800	5.85e-01	7.86e-03	6.91e-03	6.92e-03	6.93e-03	6.93e-03	6.93e-03	6.92e-03
0.9825	5.88e-01	7.90e-03	6.94e-03	6.95e-03	6.96e-03	6.96e-03	6.96e-03	6.96e-03
0.9850	5.91e-01	7.94e-03	6.98e-03	6.99e-03	7.00e-03	7.00e-03	7.00e-03	6.99e-03
0.9875	5.94e-01	7.98e-03	7.01e-03	7.02e-03	7.03e-03	7.03e-03	7.03e-03	7.03e-03
0.9900	5.97e-01	8.02e-03	7.05e-03	7.06e-03	7.07e-03	7.07e-03	7.07e-03	7.06e-03
0.9925	6.00e-01	8.06e-03	7.09e-03	7.09e-03	7.10e-03	7.10e-03	7.10e-03	7.10e-03
0.9950	6.03e-01	8.10e-03	7.12e-03	7.13e-03	7.14e-03	7.14e-03	7.14e-03	7.13e-03
0.9975	6.06e-01	8.14e-03	7.16e-03	7.17e-03	7.17e-03	7.17e-03	7.17e-03	7.17e-03
1.0000	6.09e-01	8.19e-03	7.19e-03	7.20e-03	7.21e-03	7.21e-03	7.21e-03	7.20e-03

Table 6.12: Maximum relative error at the matching points, $d_t = 0.0025$ (Gauss points, $2D$ case).

	n=1	n=3	n=3	n=3	n=4	n=4	n=4	n=5
t	m=1	m=3	m=4	m=6	m=7	m=7	m=7	m=9
d_m	1/2	1/8	1/4	1/8	1/8	1/16	1/32	1/8
0.0025	1.47e-02	1.23e-05	1.32e-05	1.01e-05	1.01e-05	1.01e-05	1.01e-05	1.01e-05
0.0050	1.42e-02	2.30e-05	2.27e-05	1.99e-05	1.99e-05	1.99e-05	1.99e-05	1.98e-05
0.0075	1.39e-02	3.36e-05	3.17e-05	2.94e-05	2.94e-05	2.94e-05	2.94e-05	2.93e-05
0.0100	1.36e-02	4.40e-05	4.11e-05	3.86e-05	3.86e-05	3.86e-05	3.86e-05	3.86e-05
0.0125	1.35e-02	5.44e-05	5.01e-05	4.77e-05	4.77e-05	4.77e-05	4.77e-05	4.77e-05
0.0150	1.34e-02	6.44e-05	5.89e-05	5.66e-05	5.66e-05	5.66e-05	5.66e-05	5.66e-05
0.0175	1.33e-02	7.41e-05	6.73e-05	6.53e-05	6.53e-05	6.53e-05	6.53e-05	6.53e-05
0.0200	1.32e-02	8.35e-05	7.53e-05	7.37e-05	7.38e-05	7.38e-05	7.38e-05	7.37e-05
0.0225	1.32e-02	9.29e-05	8.32e-05	8.21e-05	8.21e-05	8.21e-05	8.21e-05	8.20e-05
0.0250	1.32e-02	1.02e-04	9.13e-05	9.01e-05	9.01e-05	9.01e-05	9.01e-05	9.01e-05
0.0275	1.32e-02	1.11e-04	9.92e-05	9.80e-05	9.80e-05	9.80e-05	9.80e-05	9.79e-05
0.0300	1.32e-02	1.19e-04	1.07e-04	1.06e-04	1.06e-04	1.06e-04	1.06e-04	1.06e-04
0.0325	1.32e-02	1.28e-04	1.14e-04	1.13e-04	1.13e-04	1.13e-04	1.13e-04	1.13e-04
0.0350	1.32e-02	1.36e-04	1.21e-04	1.20e-04	1.20e-04	1.20e-04	1.20e-04	1.20e-04
0.0375	1.32e-02	1.44e-04	1.28e-04	1.27e-04	1.27e-04	1.27e-04	1.27e-04	1.27e-04
0.0400	1.32e-02	1.51e-04	1.35e-04	1.33e-04	1.33e-04	1.33e-04	1.33e-04	1.33e-04
0.0425	1.32e-02	1.58e-04	1.41e-04	1.40e-04	1.40e-04	1.40e-04	1.40e-04	1.40e-04
0.0450	1.32e-02	1.65e-04	1.47e-04	1.46e-04	1.46e-04	1.46e-04	1.46e-04	1.46e-04
0.0475	1.32e-02	1.72e-04	1.53e-04	1.52e-04	1.52e-04	1.52e-04	1.52e-04	1.52e-04
0.0500	1.32e-02	1.78e-04	1.59e-04	1.57e-04	1.57e-04	1.57e-04	1.57e-04	1.57e-04
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.9525	1.59e-02	3.80e-04	3.34e-04	3.34e-04	3.34e-04	3.34e-04	3.34e-04	3.34e-04
0.9550	1.59e-02	3.80e-04	3.34e-04	3.34e-04	3.34e-04	3.35e-04	3.35e-04	3.34e-04
0.9575	1.59e-02	3.80e-04	3.34e-04	3.34e-04	3.35e-04	3.35e-04	3.35e-04	3.34e-04
0.9600	1.59e-02	3.80e-04	3.34e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9625	1.59e-02	3.80e-04	3.34e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9650	1.59e-02	3.80e-04	3.34e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9675	1.59e-02	3.80e-04	3.34e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9700	1.59e-02	3.80e-04	3.34e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9725	1.59e-02	3.80e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9750	1.59e-02	3.81e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9775	1.59e-02	3.81e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9800	1.59e-02	3.81e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04	3.35e-04
0.9825	1.59e-02	3.81e-04	3.35e-04	3.35e-04	3.36e-04	3.36e-04	3.36e-04	3.35e-04
0.9850	1.59e-02	3.81e-04	3.35e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.35e-04
0.9875	1.59e-02	3.81e-04	3.35e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04
0.9900	1.59e-02	3.81e-04	3.35e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04
0.9925	1.59e-02	3.81e-04	3.35e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04
0.9950	1.59e-02	3.81e-04	3.35e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04
0.9975	1.59e-02	3.82e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04
1.0000	1.59e-02	3.82e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04	3.36e-04

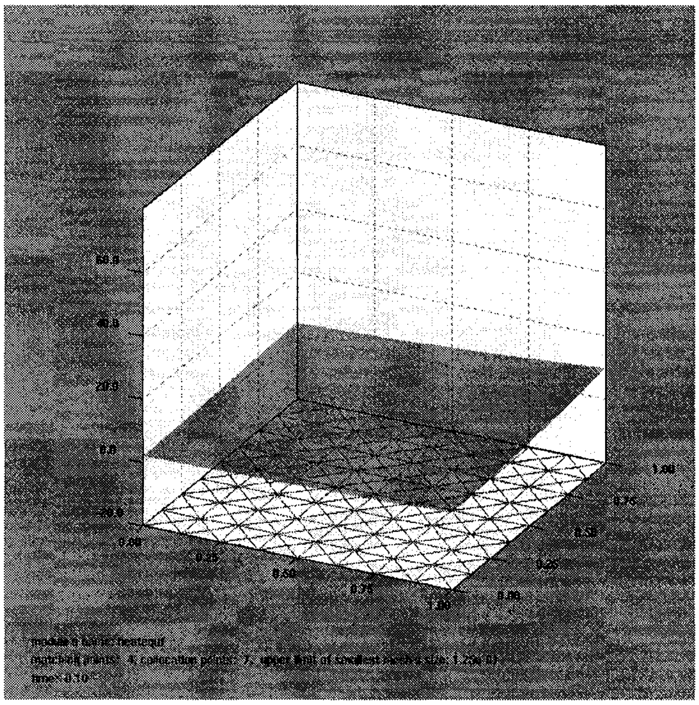


Figure 6.4: Solution of the time-dependent problem, when $d_m = 0.125$, $n = 4$, $m = 7$, $t = 0.10$.

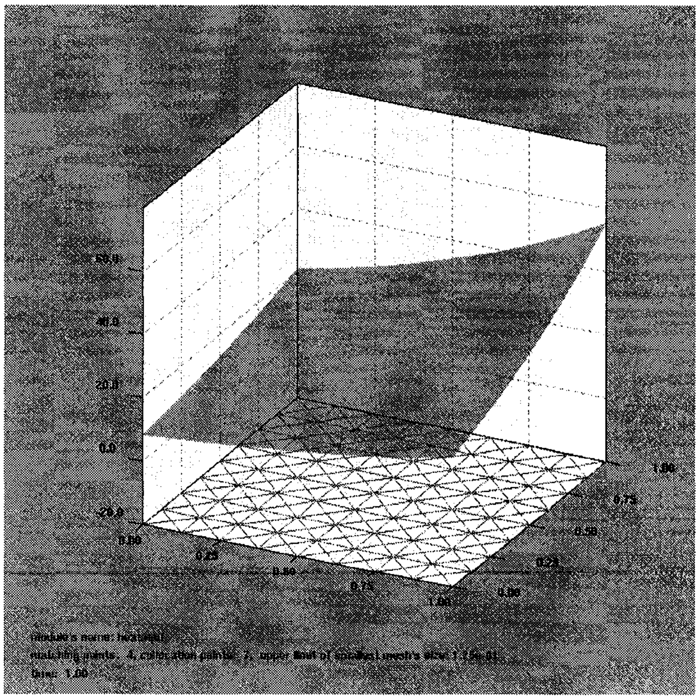


Figure 6.5: Solution of the time-dependent problem, when $d_m = 0.125$, $n = 4$, $m = 7$, $t = 1.00$.

Table 6.13: The coefficients of the newly selected collocation points.

m	c_0	c_1	c_2
4	0.3233434093759132	0.6059183105326176	0.0707382800914692
7	0.3825061085552472	0.5420392280380437	0.0754546634067091
	0.0304868878007339	0.2985127708094274	0.6710003413898387

Table 6.14: Maximum absolute error at the matching points, $d_t = 1e-6$ (Gauss points, $2D$ case).

	n=3	n=3	n=3	n=3	n=4	n=4	n=4	n=4
t	m=4	m=4	m=4	m=4	m=7	m=7	m=7	m=7
d_m	1/2	1/4	1/8	1/16	1/4	1/8	1/16	1/32
0.000001	3.41e-04	2.57e-05	1.73e-06	1.12e-07	9.06e-07	3.40e-08	9.82e-10	2.94e-11
0.000002	3.41e-04	2.56e-05	1.73e-06	1.12e-07	8.96e-07	3.22e-08	7.74e-10	4.72e-11
0.000003	3.41e-04	2.56e-05	1.72e-06	1.12e-07	8.86e-07	3.05e-08	6.24e-10	6.27e-11
0.000004	3.41e-04	2.56e-05	1.72e-06	1.12e-07	8.77e-07	2.89e-08	5.65e-10	7.70e-11
0.000005	3.41e-04	2.56e-05	1.72e-06	1.13e-07	8.67e-07	2.75e-08	5.84e-10	9.07e-11
0.000006	3.41e-04	2.56e-05	1.72e-06	1.14e-07	8.58e-07	2.61e-08	6.01e-10	1.04e-10
0.000007	3.41e-04	2.56e-05	1.72e-06	1.15e-07	8.49e-07	2.49e-08	6.38e-10	1.18e-10
0.000008	3.41e-04	2.55e-05	1.72e-06	1.16e-07	8.39e-07	2.37e-08	6.72e-10	1.31e-10
0.000009	3.41e-04	2.55e-05	1.72e-06	1.17e-07	8.30e-07	2.26e-08	7.03e-10	1.45e-10
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.000091	3.37e-04	2.54e-05	1.98e-06	1.40e-07	5.15e-07	3.26e-08	1.96e-09	1.30e-09
0.000092	3.37e-04	2.54e-05	1.98e-06	1.40e-07	5.19e-07	3.27e-08	1.97e-09	1.31e-09
0.000093	3.37e-04	2.55e-05	1.98e-06	1.40e-07	5.22e-07	3.28e-08	1.99e-09	1.32e-09
0.000094	3.37e-04	2.55e-05	1.98e-06	1.40e-07	5.26e-07	3.29e-08	2.00e-09	1.34e-09
0.000095	3.37e-04	2.55e-05	1.99e-06	1.40e-07	5.30e-07	3.31e-08	2.01e-09	1.35e-09
0.000096	3.37e-04	2.55e-05	1.99e-06	1.40e-07	5.33e-07	3.32e-08	2.03e-09	1.36e-09
0.000097	3.37e-04	2.55e-05	1.99e-06	1.41e-07	5.37e-07	3.33e-08	2.04e-09	1.38e-09
0.000098	3.37e-04	2.55e-05	1.99e-06	1.41e-07	5.41e-07	3.34e-08	2.06e-09	1.39e-09
0.000099	3.37e-04	2.55e-05	1.99e-06	1.41e-07	5.44e-07	3.36e-08	2.07e-09	1.40e-09
0.000100	3.37e-04	2.55e-05	2.00e-06	1.41e-07	5.48e-07	3.37e-08	2.09e-09	1.42e-09

6.4 A problem with a peak in the solution

Now consider a different PDE, namely,

$$\begin{aligned}
\frac{\partial u}{\partial t} &= \Delta u - a(4b^2t((x-r)^2+(y-r)^2 \\
&\quad -4bt-1)e^{-b((x-r)^2+(y-r)^2)}, \quad (x,y)\in\Omega\subset R^2, t\in[0,1] \\
u(x,y,0) &= 1, \quad (x,y)\in\Omega, \\
u(x,y,t) &= ate^{-b((x-r)^2+(y-r)^2)}+1, \quad (x,y)\in\partial\Omega, t\in[0,1],
\end{aligned}$$

where Ω is as before. We denote this problem as $NEQ2$. The exact solution is

$$u(x,y,t) = ate^{-b((x-r)^2+(y-r)^2)}+1, \quad (x,y)\in\Omega, t\in[0,1]$$

For example, let $a = 64.0$ and $b = 10000.0$. Then the exact solution has a sharp peak at $(0.5, 0.5)$, the center of Ω. This time, if we use a uniform mesh, we need very small elements, which will cause much calculation time. Because of this, we apply local refinement at the center, which limits the number of elements. This similar to what we did in Section 3.3.

We choose $n = 4$, $m = 7$ and $d_t = 0.01$, and use the coefficients from Table 6.1. First we set $d_m = 0.04$, $d_r = 0.05$ and calculate the solutions. Table 6.15 lists the maximum errors. The first line gives the maximum absolute errors and the second line gives the maximum relative errors.

Table 6.15: Maximum error at the matching points of problem $NEQ2$, $n = 4$, $m = 7$, $d_t = 0.01$, $d_m = 0.04$, $d_r = 0.05$ (Gauss points, $2D$ case).

t	0.01	0.02	0.03	$\cdots$	0.98	0.99	1.00
e_a	9.52e-02	1.90e-01	2.84e-01	$\cdots$	8.94e+00	9.04e+00	9.13e+00
e_r	6.83e-02	1.35e-01	1.99e-01	$\cdots$	3.54e+00	3.56e+00	3.58e+00

From Table 6.15, we see that the error is large. When $t = 1.00$, the maximum absolute error is $e_a = 9.13e+00$. From the exact solution, we know that its maximum value is 64.0 when $t = 1.00$ and its maximum relative error is too large. Thus the local refinement is still not sufficient. Figure 6.6 shows the solution at $t = 1.00$.

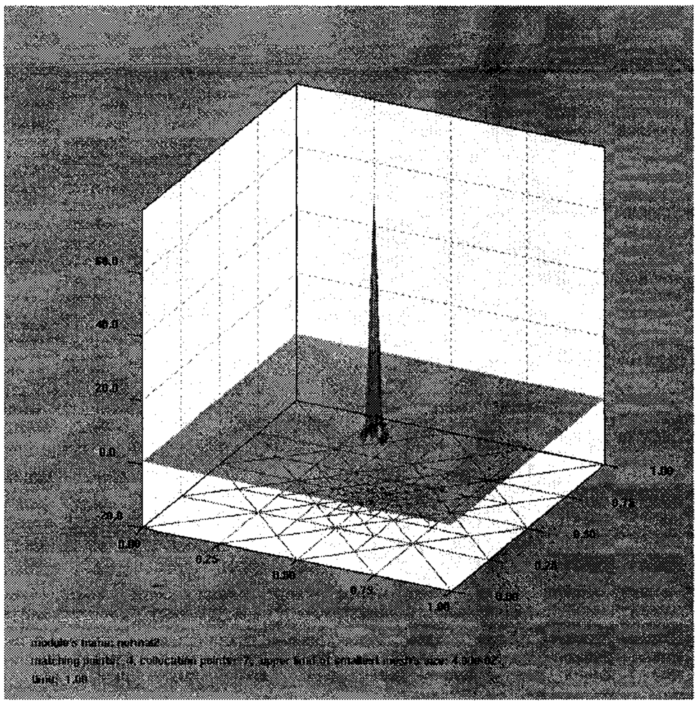


Figure 6.6: Solution of problem $NEQ2$, when $dm = 0.04$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 1.00$.

Table 6.16: Maximum error at the matching points of problem $NEQ2$, $n = 4$, $m = 7$, $d_t = 0.01$, $d_m = 0.01$, $d_r = 0.05$ (Gauss points, $2D$ case).

t	0.01	0.02	0.03	$\cdots$	0.98	0.99	1.00
e_a	2.42e-04	4.85e-04	7.27e-04	$\cdots$	2.38e-02	2.40e-02	2.42e-02
e_r	1.63e-04	2.46e-04	2.96e-04	$\cdots$	2.44e-03	2.45e-03	2.46e-03

Next, we change $d_m = 0.01$ and calculate again. Table 6.16 lists the maximum errors. Figure 6.7 shows the solution at $t = 0.50$, and Figure 6.8 shows the solution at $t = 1.00$.

Also, we want to see the case where the minimal angle of the original mesh in the domain is small and to observe the accuracy. We change the range of one of the coordinates of the domain to 1/4 of the original size, $y \in [0.375, 0.625]$. Figure 6.9 shows the generated mesh, using the same parameters as in the previous case. Table 6.17 lists the maximum errors. We note that the error increases. Figure 6.10 shows the solution at $t = 0.50$, and Figure 6.11 shows the solution at $t = 1.00$ for this case.

For the current PDE, we already know the exact solution. In particular, we know where the solution has a peak. In real problems, we do not know the exact solution, and we do not know, for most PDEs, if there is a peak, and where the peak is. Therefore, we

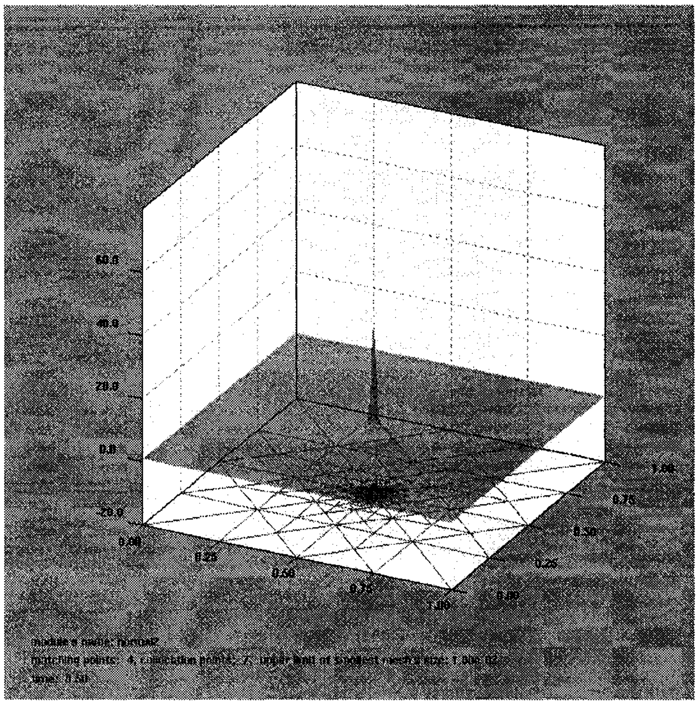


Figure 6.7: Solution of problem $NEQ2$, when $dm = 0.01$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 0.50$

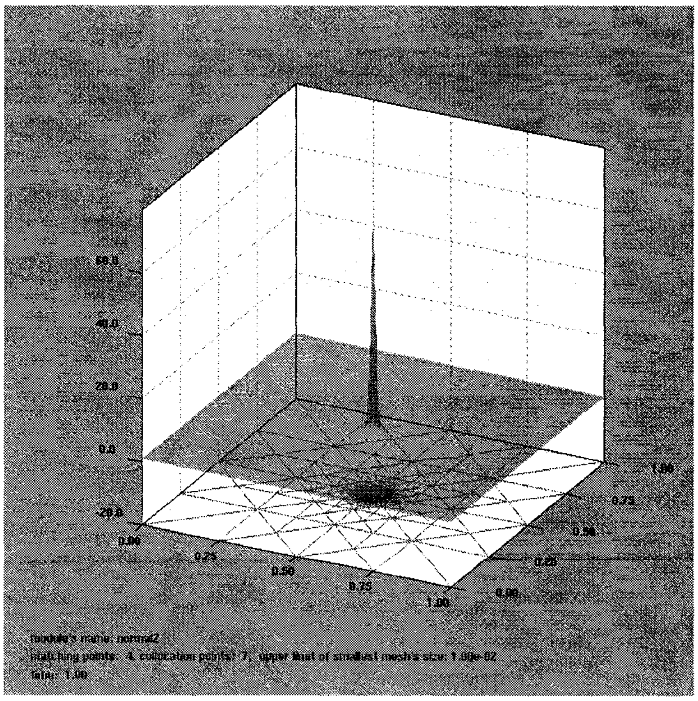


Figure 6.8: Solution of problem $NEQ2$, when $dm = 0.01$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 1.00$

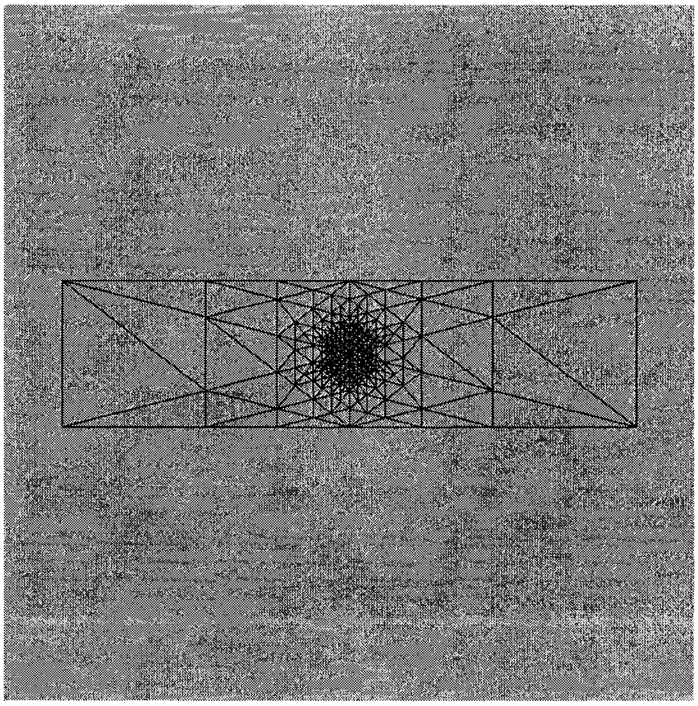


Figure 6.9: The refined mesh of the smaller domain, when $d_m = 0.01$, $d_r = 0.05$. The minimal angle of the elements is smaller.

Table 6.17: Maximum error at the matching points of problem $NEQ2$ of a smaller region, $n = 4$, $m = 7$, $d_t = 0.01$, $d_m = 0.01$, $d_r = 0.05$ (Gauss points, $2D$ case).

t	0.01	0.02	0.03	$\cdots$	0.98	0.99	1.00
e_a	5.51e-04	1.12e-03	1.70e-03	$\cdots$	5.66e-02	5.71e-02	5.77e-02
e_r	3.71e-04	5.68e-04	6.90e-04	$\cdots$	1.37e-02	1.38e-02	1.39e-02

can not predetermine the location where local refinement is needed. An adaptive method to detect the location of peaks and other rapid transitions, and corresponding automatic local mesh refinement is a topic for future work.

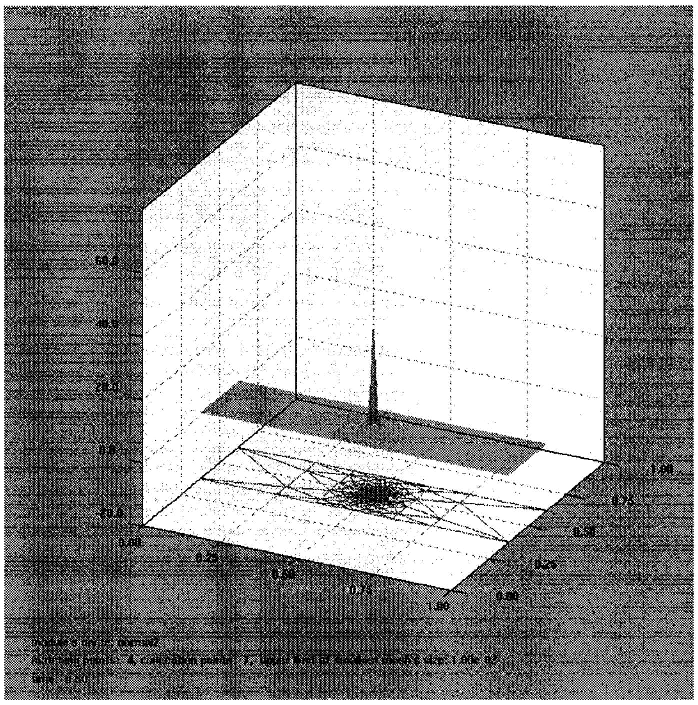


Figure 6.10: Solution of problem $NEQ2$, when $dm = 0.01$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 0.50$.

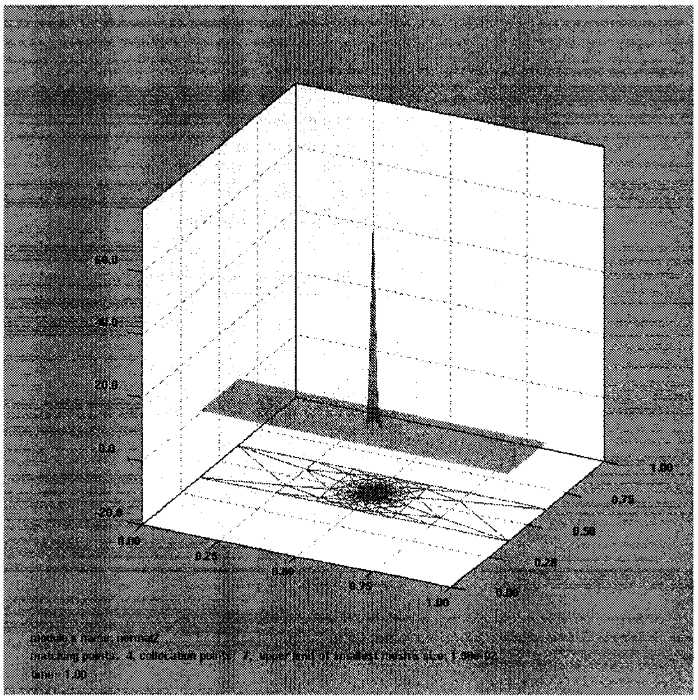


Figure 6.11: Solution of problem $NEQ2$, when $dm = 0.01$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 1.00$.

6.5 A problem with a sharp ridge

Now consider a more complicated PDE,

$$
\begin{aligned}
\frac{\partial u}{\partial t} &= \Delta u - a(t((c\sin(d\pi(x+t))+h)(4b^2(y-r)^2-2b) - \\
&\quad cd\pi(\sin(d\pi(x+t))d\pi - \cos(d\pi(x+t))) + \\
&\quad \sin(d\pi(x+t))+h)e^{-b(y-r)^2}, \quad (x,y)\in\Omega\subset R^2, t\in[0,1] \\
u(x,y,0) &= 10, \quad (x,y)\in\Omega, \\
u(x,y,t) &= at\,(c\sin(d\pi(x+t))+h)\,e^{-b(y-r)^2}+10, \quad (x,y)\in\partial\Omega, t\in[0,1].
\end{aligned}
$$

where Ω is as above. We denote this problem as $NEQ6$. The exact solution is

$$u(x,y,t) = at\,(c\sin(d\pi(x+t))+h)\,e^{-b(y-r)^2}+10, \quad (x,y)\in\Omega, t\in[0,1].$$

As an example, we set $a = 55.0$, $b = 10000.0$, $c = 0.2$, $d = 6.0$ and $h = 0.85$. From the exact solution, we know that the solution has a sharp ridge at the line $y = 0.5$. Therefore, we apply local refinement, but this time at the line $y = 0.5$. Now, d_r denotes the upper limit of the distance from the center of any local region to the refinement line. We choose $n = 4$, $m = 7$ and $d_t = 0.01$, set $d_m = 0.01$ and $d_r = 0.05$ to calculate the solution. The collocation points are selected according to Table 6.1. Table 6.18 lists the the maximum errors. Figure 6.12 shows the solution at $t = 0.55$, and Figure 6.13 shows the solution at $t = 1.00$.

Table 6.18: Maximum error at the matching points of problem $NEQ6$, $n = 4$, $m = 7$, $d_t = 0.01$, $d_m = 0.01$, $d_r = 0.05$ (Gauss points, $2D$ case).

t	0.01	0.02	0.03	$\cdots$	0.98	0.99	1.00
e_a	1.02e-01	1.45e-01	1.73e-01	$\cdots$	4.50e-01	4.48e-01	4.45e-01
e_r	9.67e-03	1.33e-02	1.58e-02	$\cdots$	3.65e-02	3.62e-02	3.58e-02

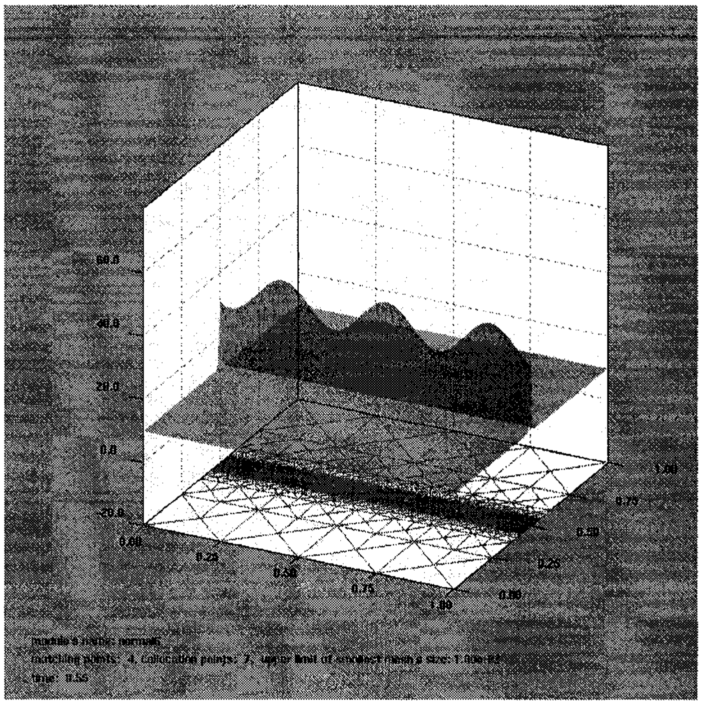


Figure 6.12: Solution of problem $NEQ6$, when $d_m = 0.01$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 0.55$.

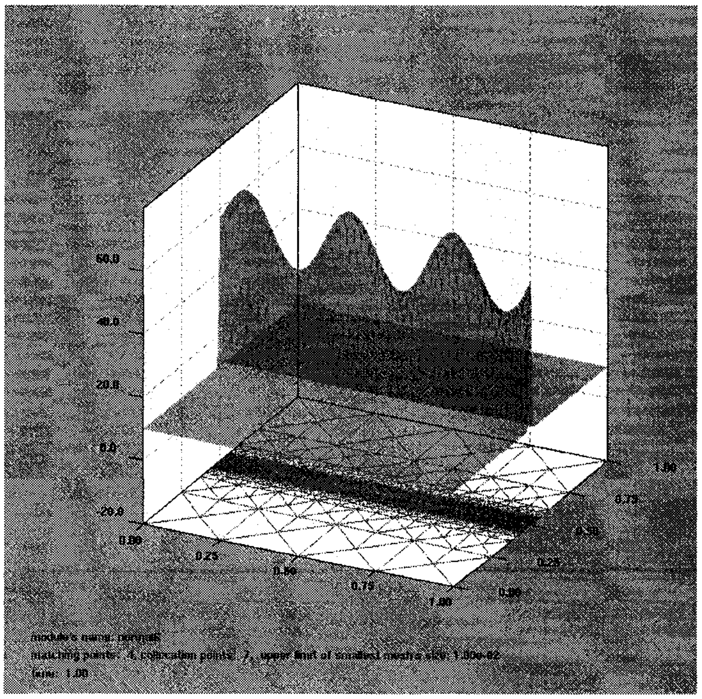


Figure 6.13: Solution of problem $NEQ6$, when $d_m = 0.01$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 1.00$.

6.6 A PDE with unknown exact solution

Consider a heat exchange PDE, namely:

$$
\begin{aligned}
\frac{\partial u}{\partial t} &= \Delta u, \quad (x,y) \in \Omega \subset R^2, t \in [0,1] \\
u(x,y,0) &= 0, \quad (x,y) \in \Omega - [(0,0),(0,1)] \\
u(x,y,t) &= 60, \quad (x,y) \in [(0,0),(0,1)] \\
\frac{\partial u}{\partial \eta} &= -10, \quad (x,y) \in [(1,0),(1,1)] \\
\frac{\partial u}{\partial \eta} &= 0, \quad (x,y) \in [(0,0),(1,0)],\ [(0,1),(1,1)], \\
& \qquad [(0.25,0.25),(0.75,0.25)],\ [(0.75,0.25),(0.75,0.75)], \\
& \qquad [(0.25,0.75),(0.75,0.75)],\ [(0.25,0.25),(0.25,0.75)], \\
& \qquad t \in [0,1].
\end{aligned}
$$

This is a typical real problem. At the beginning, the temperature is zero throughout the domain and 60 at $x = 0$. Heat flows from $x = 0$ to $x = 1$. On other boundaries of the domain, heat cannot flow out since $\frac{\partial u}{\partial \eta} = 0$. We want to know the distribution of the temperature in the domain after a certain time. We denote this problem as $HEA3$. The domain Ω is shown in Figure 6.15.

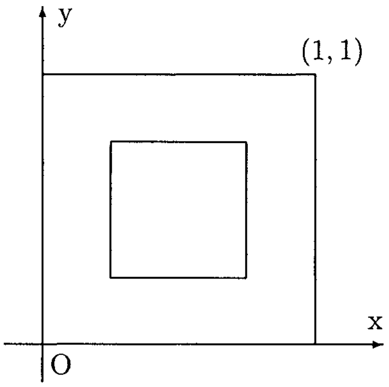


Figure 6.14: The domain Ω defined in problem $HEA3$.

Note that this domain contains a hole in the center. So we need to use the method mentioned in Subsection 5.3.3 to define the original region. In this case, the situation arises where two local regions, which are the children of Ω, share two common faces, as shown in Figure 6.15.

First, we use a uniform mesh to calculate the solutions. We use the same collocation points as in the previous example, and we set $n = 4$, $m = 7$ and $d_m = 0.25$. Figure 6.16 shows the solution at $t = 0.45$.

Note that there are differences between the solutions at certain points, for example, $(0.25, 0.25)$. This means that the solutions corresponding to local polynomials of neighbouring elements are different. Therefore, we want to apply local refinement at these points. Also, we find that the heat flow is very fast at the first few time steps, and there is a peak at $x = 0$. Thus, we want to apply refinement near the four points, $(0.25, 0.25)$, $(0.75, 0.25)$, $(0.75, 0.75)$, $(0.75, 0.75)$ and near the line $x = 1$. We let $d_m = 0.025$, $d_r = 0.05$. Figure 6.17 shows the refined mesh. Since the exact solution is unknown, we only give a $3D$ plot of the numerical solution. Figure 6.18 shows the solution at $t = 0.01$, Figure 6.19 shows the solution at $t = 0.50$ and Figure 6.20 shows the solution at $t = 1.00$.

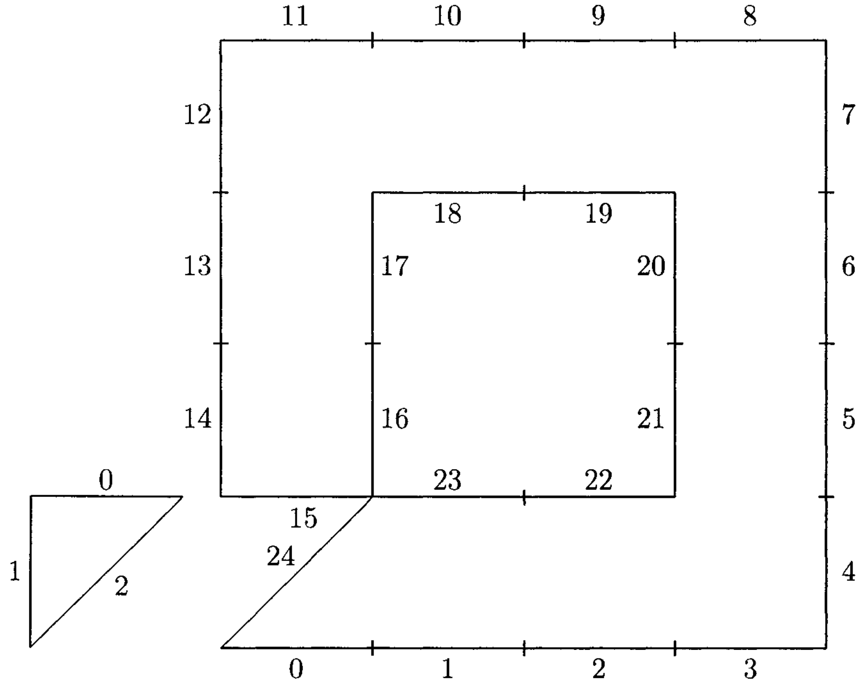
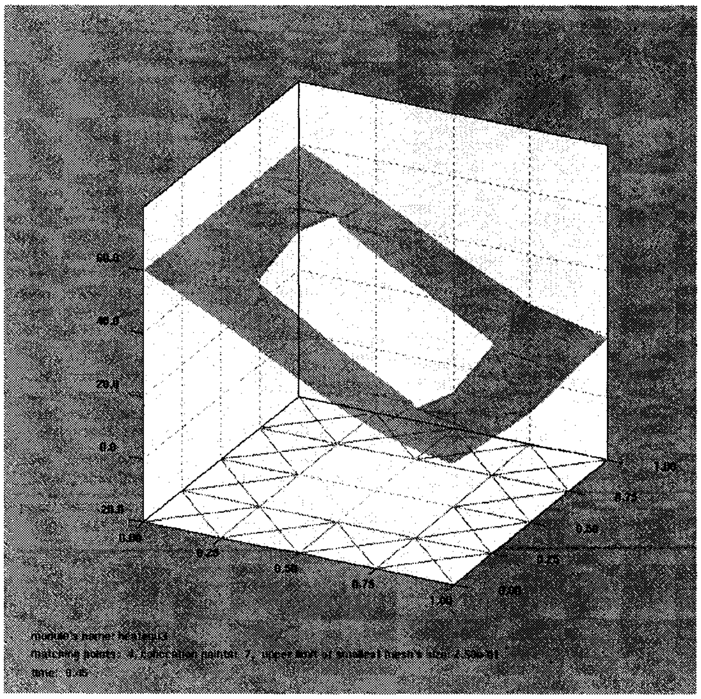


Figure 6.15: Two local regions, which are the children of Ω, share two common faces and the labels of their faces.



module's name: hea3equ3
matching points: 4, collocation points: 7, upper limit of smallest mesh's size: 2.50e-01
time: 0.45

Figure 6.16: Solution of problem $HEA3$, when $d_m = 0.25$, $n = 4$, $m = 7$, $t = 0.45$.

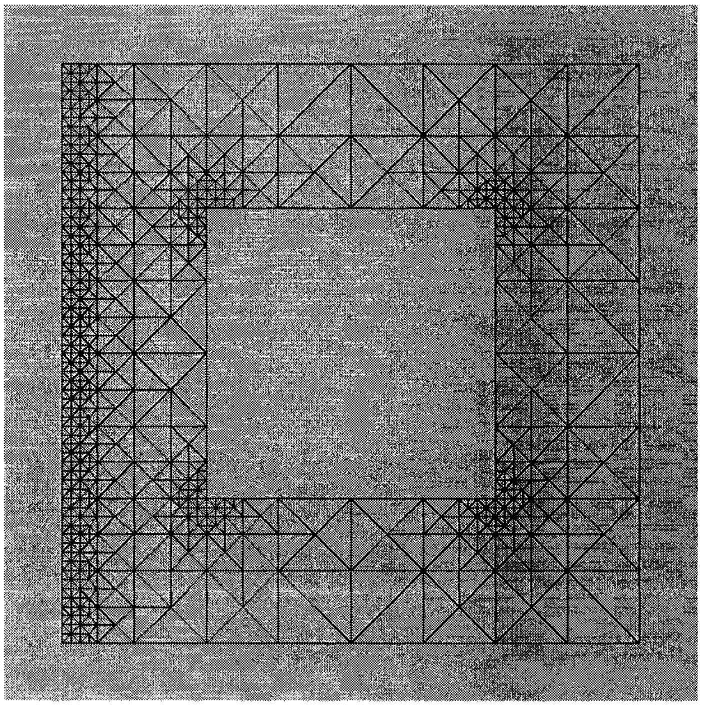


Figure 6.17: The locally refined mesh of problem $HEA3$ near the points (0.25, 0.25), (0.75, 0.25), (0.75, 0.75), (0.75, 0.75) and near the line $x = 0$, when $d_m = 0.025$, $d_r = 0.05$.

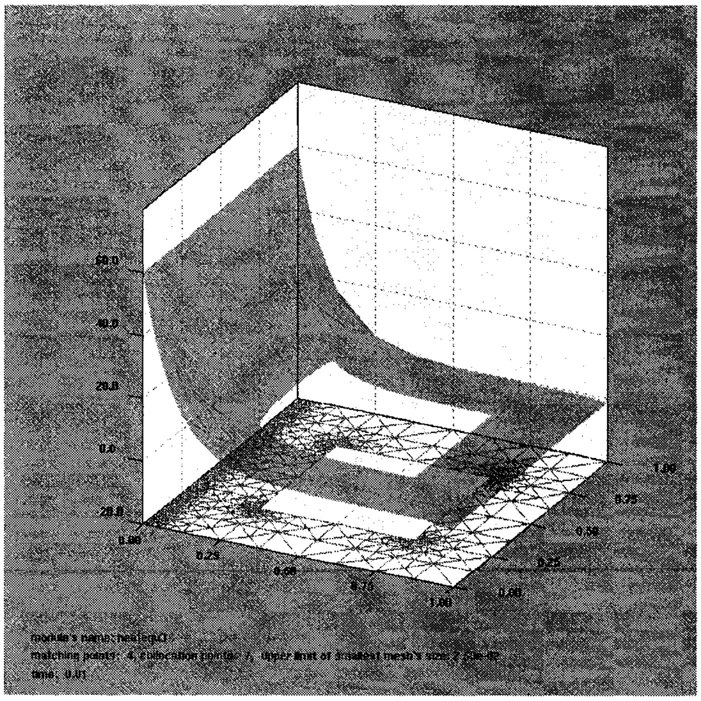


Figure 6.18: Solution of problem $HEA3$, when $d_m = 0.025$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 0.01$.

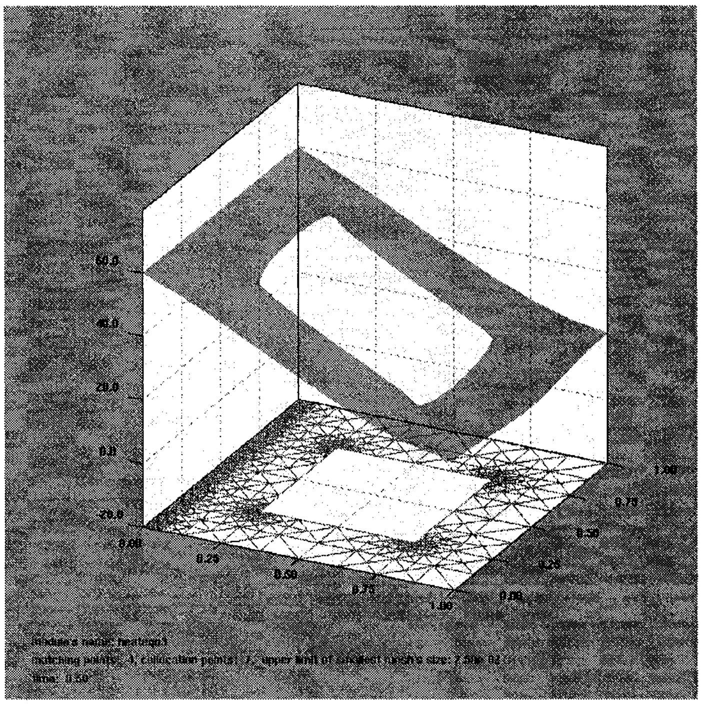


Figure 6.19: Solution of problem $HEA3$, when $d_m = 0.025$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 0.50$.

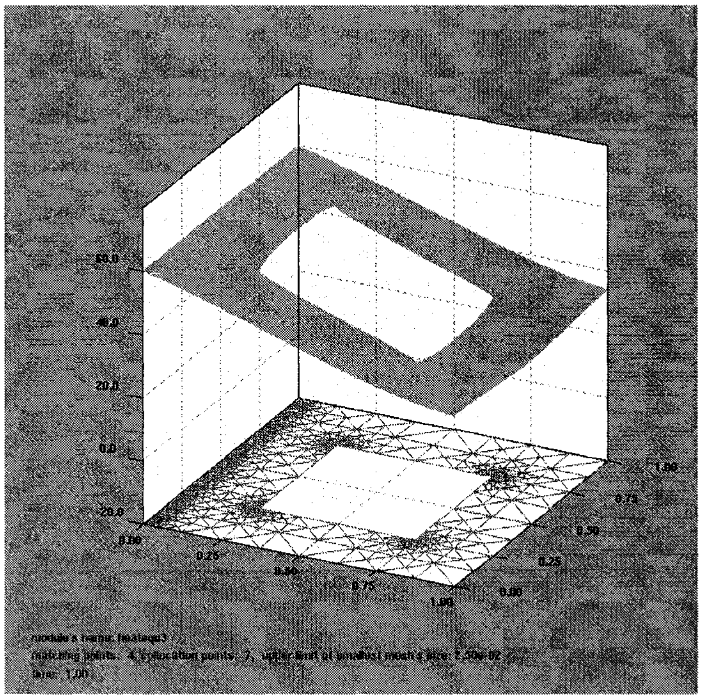


Figure 6.20: Solution of problem $HEA3$, when $d_m = 0.025$, $d_r = 0.05$, $n = 4$, $m = 7$, $t = 1.00$.

Chapter 7

Implementation of the Visualization Tool

Since typical solutions of a PDE give large amounts of data, one cannot easily interpret these data and observe the changing solution. A good graphics tool helps us to understand the results better for further analysis.

7.1 Objectives

There are two main objectives to be realized. One is to visualize the mesh of a certain domain Ω generated by the Rivara algorithm, the other is to visualize the solutions. Since the plots of solutions of PDEs in $2D$ space will be three-dimensional, there should be functions for moving, rotating and scaling, which can help us to observe the solution more precisely.

7.2 Graphics interface

For a visual application, the first consideration is to choose a suitable graphics frame interface. At present, there are many kinds of interfaces offered by different operating systems. Since the program is primarily designed to run on Linux systems, we have chosen Motif, which is a widely used X11 graphics library, as the frame of the visualization tool. To draw

3*D* graphics, we have chosen OpenGL as the 3*D* rendering interface. C/C++ is again the language used to implement the code. Figure 7.1 shows the architecture of the visualization tool.

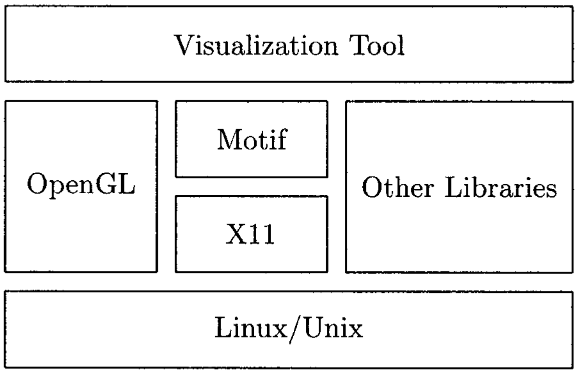


Figure 7.1: Architecture of the visualization tool.

To integrate OpenGL and X11/Motif, we need to use "drawing area widgets" of OpenGL [39]. We should first find the color map of the current X11 display, then create the widget as follows:

```
int n;
Widget wGlxArea;
Colormap cMap;
Arg args[10];
......
XtSetArg(args[n], XtNcolormap, cMap); n++;
wGlxArea = XtCreateWidget(<name of widget>,
           glwMDrawingAreaWidgetClass, <parent widget>, args, n);
```

Now we can create an OpenGL rendering context and we are ready to render the 3*D* graphics [41]. Creating a rendering context is done as follows:

```
Widget w;
XVisualInfo *visinfo;
Display *dpy;
GLXContext glxcontext;
```

```
......
XtVaGetValues(w, GLwNvisualInfo, &visinfo, NULL);
glxcontext = glXCreateContext(XtDisplay(w), visinfo, 0, True);
```

Every time when we need to render the graphics, we set the rendering context in which we want to draw graphics as the current context:

```
glXMakeCurrent(dpy, XtWindow(w), glxcontext);
```

Anything that is drawn in the following steps will be rendered into this context.

7.3 Integration of visualization and calculation

The visualization tool will first load the module "configuration parameters reading", presented in Section 5.11, into memory and use it to read in the configuration parameters.

If the user wants the visualization tool to draw the mesh created by the Rivara algorithm, the application will use the module "PDEs calculation" to generate the mesh. It then recursively searches the binary tree to find the coordinates of the mesh and plots the mesh into the OpenGL rendering context.

If the user wants the visualization tool to plot the solutions of the PDE, the application will also use the module "PDEs calculation" to calculate the solutions. The module "PDEs calculation" will generate the solutions of one time step and put them into a certain block of memory, then continue to process the next time step, *etc.*, until the end. Once there are valid solution data in memory, the graphics tool can use them to plot the graphics. If there are no data, the graphics tool will wait and do nothing, except when the user wants it to do something, such as moving or rotating the coordinates. Therefore, we need a suitable multiple processing method to accomplish this task. On Linux systems, we use Posix thread to do this [32, 33]. Creating a thread is accomplished as follows:

```
pthread_t *ptid;
......
pthread_create(ptid, NULL, <function of calculation thread>,
               <input parameter>);
```

We use a semaphore to notify the graphics tool that solution data are ready. A semaphore is created as follows:

```
sem_t *psem;
......
sem_init(psem, 0, 0);
```

Generally, if the working process needs to wait for something, it will be hung up and will do nothing. However, a GUI process can not be hung up. Otherwise, the user can not interact with the graphics interface. Therefore, the semaphore must not block the OpenGL GUI thread. The GUI thread will repeat detecting if any solution data are ready. If there are solution data ready, it performs plotting. Repeating detection is done as follows:

```
int iret;
......
iret = sem_trywait(psem);
if(!iret){  /* data ready, semaphore is unlocked */
    /* get data and do plotting */
   ....
}
else{ /* data is not ready or some exception happens */
    if(errno != EAGAIN) /* exception happens */
        return errcode;
}
```

On the other hand, the calculation thread should notify the GUI thread when the solution data are ready:

```
sem_post(psem);
```

Now the GUI thread and the calculation thread can exchange information.

7.4 OpenGL implementation

Using OpenGL to draw the mesh is easier than the $3D$ plot of the solutions. The graphics tool first draws the domain, and then searches the binary tree to find the midpoint of a certain local region and draws the common face. All these operations are in $2D$. Drawing $2D$ graphics using OpenGL is done as follows:

```
gluOrtho2D(left, right, bottom, top);
```

We have already shown pictures of a $2D$ mesh in Section 3.3.

$3D$ graphics is used to plot the solutions calculated by the module "PDEs calculation". If only the solutions at the matching points are used, then it will be difficult to draw the graphics since the matching points are on the edges of the elements. This is why we also calculate the solution at the vertices of the elements. This helps us to draw $3D$ polygons. We send the solution values at the matching points and at the vertices to the graphics tool, and let it do the 3D plotting.

One thing that needs to be mentioned here is the rotation of $3D$ coordinates. For the current program, the rotation axis does not pass the origin of the coordinate system, so that it is not possible to directly use the "glRotate" function offered by OpenGL. We first need to read the OpenGL model matrix. The first three rows of the 4×4 model matrix read are the directions of the current coordinates that have been translated. So we must do a rotation according to these new coordinates, and then plot the images. Figure 7.2 is an example to show the effect of a $3D$ plot, Figure 7.3 is the $3D$ plot after a certain rotation and Figure 7.4 is the $3D$ plot after a certain rotation and a zoom.

7.5 Animation generation

A typical parabolic PDE is time-dependent, *i.e.*, the solution will change with time. This also means that a single picture cannot show all solutions of the problem. For each time step, there will be a picture which shows the distribution of the solutions at that time. Showing one frame after another will give an animation.

The simplest way to generate an animation is to let the visualization tool perform instant

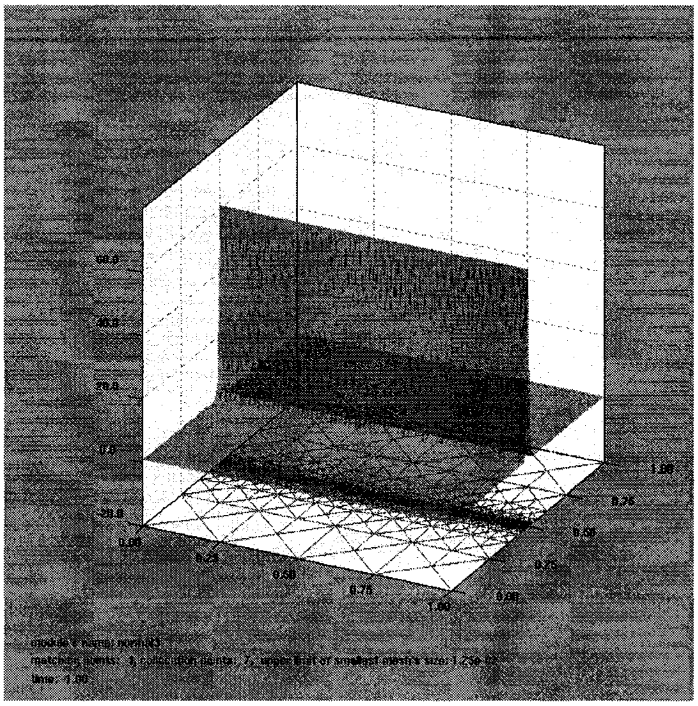


Figure 7.2: Solution of problem $NEQ3$, when $dm = 0.0125$, $d_r = 0.015$, $n = 4$, $m = 7$, $t = 1.00$.

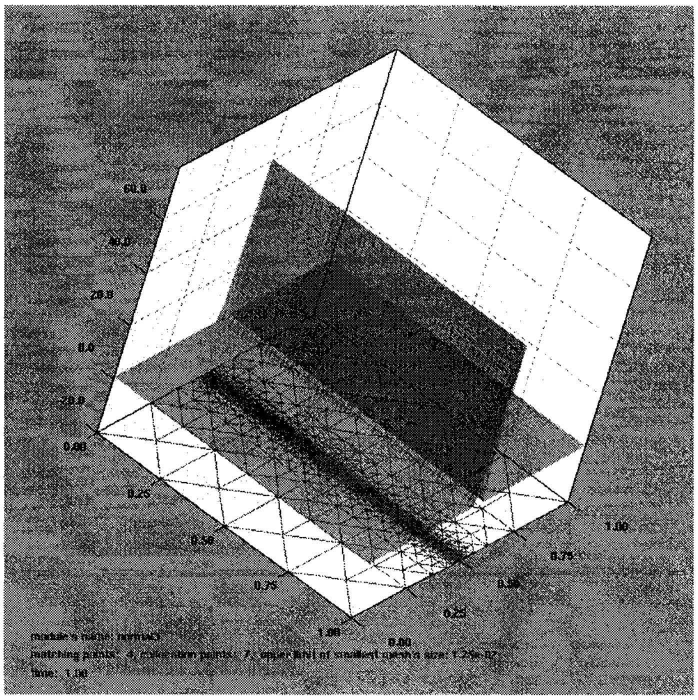


Figure 7.3: Solution of problem $NEQ3$, when $dm = 0.0125$, $d_r = 0.015$, $n = 4$, $m = 7$, $t = 1.00$, rotated.

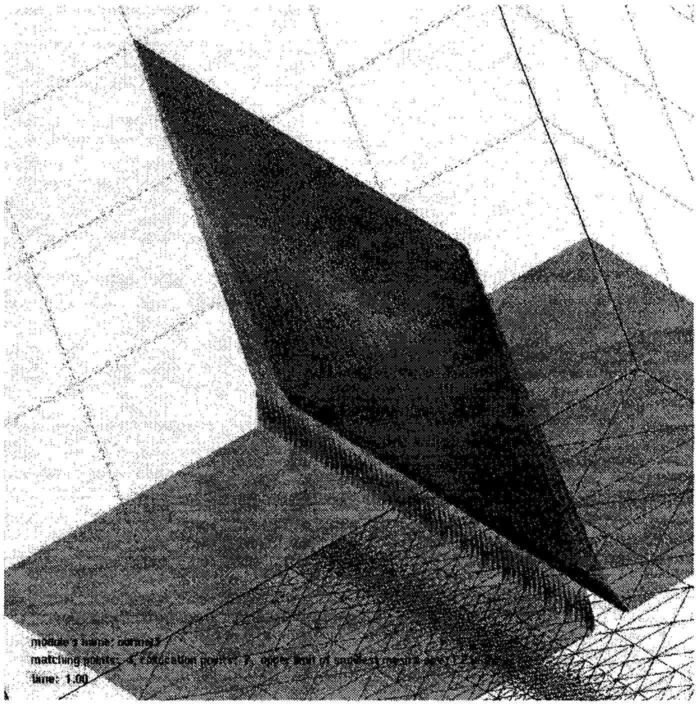


Figure 7.4: Solution of problem $NEQ3$, when $dm = 0.0125$, $d_r = 0.015$, $n = 4$, $m = 7$, $t = 1.00$, rotated and zoomed

plotting. This means that the visualization tool will draw a picture of a certain time step once the solutions are calculated. Whenever we need to observe the solutions graphically for the same PDE, we need to recompute the problem, since the solutions are lost when the program exits.

A better way to generate an animation is to save the information of the pictures into a file. There are two ways of implementing this function. One way is directly saving the plotted images. The other is saving data that describes the structure of the $3D$ plots of the solutions. At present, the solution used by the graphics tool is to save images.

To generate animation files, we first use OpenGL to plot the solutions, which produces an image in the frame buffer. The next step is to get the image out of the frame buffer and save it in the memory obtained by the graphics tool. This can be done through the following function:

```
GLvoid *pixels;
......
```

```
glReadPixels(x, y, width, height, <format>, <type>, *pixels);
```

The image is now in the memory pointed by parameter "pixels".

Next, we need to save the image into an animation file, one frame after another. There are several kinds of formats of animation files that can be used. GIF is the most widely used animation file, and many programs can play GIF files. GIF supports not more than 256 colors [42, 43]. If the image we get from the color buffer is a true color image, then we need to do quantification first. This process may lose some color information of the original image, which is a shortcoming of GIF. When the image data are ready, we need to change it into the GIF file format and store it into a file. Thus an animation file has been created.

An alternative way to solve the problem of 256 colors of the GIF file format is to use some type of moving pictures encoding method to convert the images into multimedia files, for example, AVI files. This is a topic for future work.

7.6 Data storage

We can directly store the solution data into data files for later plotting. The solution data already contains enough information to regenerate all graphics and animations. At present, this is what the graphics tool does.

One problem is that the data can only be recognized by our graphics tool, and tools written by other people do not understand the format of the data. A solution to this is to change the data into a more commonly used $3D$ data format and to store the data into a type of standard graphics data file. At present, many formats of $3D$ data files already exist. Many $3D$ representation programs offer $3D$ files, *i.e.*, 3DMax, Maya, Direct3D, *etc.*. If we store the solution into these standard $3D$ files, we can use the $3D$ plot tools mentioned above to draw the graphics. Some work has been done in that direction, but further work is needed.

Chapter 8

Ideas for Nonlinear PDEs and for the $3D$ Case

8.1 Nonlinear parabolic PDEs

The idea of our collocation methods for nonlinear parabolic PDEs comes from the idea used for elliptic PDEs by Doedel [4, 5]. Further work for systems of nonlinear elliptic PDEs is presented in the Ph.D thesis of Sharifi [7]. Here we discuss the calculation procedure, but not the implementation.

Consider Equation (2.1), with the initial conditions (2.6) and boundary conditions (2.7), (2.8) or (2.9). As for the case of linear PDEs, we use finite difference to replace the partial derivative with respect to t. Then (2.1) is of the form

$$\frac{u_k - u_{k-1}}{dt} = \Delta u_k - q(X, t_k, u_k, \nabla u_k), \quad k = 1, 2, \cdots, n \tag{8.1}$$

We rewrite it using a nonlinear operator,

$$Nu_k \equiv \Delta u_k - [q(X, t_k, u_k, \nabla u_k) + \frac{1}{dt}u_k - \frac{1}{dt}u_{k-1}]. \tag{8.2}$$

This is a nonlinear elliptic PDE problem at given time t_k,

$$Nu_k \equiv \Delta u_k + f_k(X, u_k, \nabla u_k) = 0, \tag{8.3}$$

where

$$f_k(X, u_k, \nabla u_k) = -[q(X, t_k, u_k, \nabla u_k) + \frac{1}{dt}u_k - \frac{1}{dt}u_{k-1}]. \tag{8.4}$$

As in Section 2.2, we recursively subdivide Ω and associate a local polynomial $p_k(X) \in P_{n+m}$ to each finite element. The polynomial $p_k(X)$ must satisfy the collocation equations

$$Np_k(z_j) = 0, \quad j = 1, 2, \cdots, m,$$

and for any two neighboring finite elements, it is required that the values and normal derivates of the neighboring local polynomials match at matching points x_i on the common boundary.

The local polynomial is of the form $p_k(X) = \sum_{i=1}^{n+m} c_i^k \phi_i(X)$, where

$$Span\{\phi_1, \phi_2, \cdots, \phi_{n+m}\} = P_{n+m}.$$

Therefore, the collocation equations will be of the form

$$N\left(\sum_{i=1}^{n+m} c_i^k \phi_i(z_j)\right) = 0, \quad j = 1, 2, \cdots, m. \tag{8.5}$$

By way of the continuity requirements mentioned above, we can associate unique variables u_i^k and v_i^k to each matching point x_i, and require that

$$\begin{aligned} p_k(x_i) &= u_i^k, \\ \nabla p_k(x_i)^* \eta_i &= v_i^k. \end{aligned}$$

Using the notation of Section 2.2, we can write

$$u_k - \Phi^* c_k = 0, \quad v_k - R_\Phi^* c_k = 0. \tag{8.6}$$

From the equations (8.5) and (8.6), we see that the unknown variables are $c_k \in R^{n+m}$, for each element, and the u_i, v_i associated with the interior matching points x_i of the domain Ω. To solve (8.5) and (8.6), we use Newton's method.

Newton's method for a system of equations is

$$\begin{aligned} J(F(x_n))\delta x &= -F(x_n), \quad x_n \in R^N, F(x) \in R^N, \\ x_{n+1} &= x_n + \delta x. \end{aligned} \tag{8.7}$$

Here, J is the Jacobi matrix. Applying Newton's method to (8.5) and (8.6), omitting iteration indices, we obtain

$$L^*_\Phi \delta c_k = -r^k_N, \tag{8.8}$$

$$\delta u_k - \Phi^* \delta c_k = -r^k_u, \tag{8.9}$$

$$\delta v_k - R^*_\Phi \delta c_k = -r^k_v. \tag{8.10}$$

Here,

$$L^*_\Phi = \begin{pmatrix} L[p_k(z_1)]\phi_1(z_1) & L[p_k(z_1)]\phi_2(z_1) & \cdots & L[p_k(z_1)]\phi_{n+m}(z_1) \\ L[p_k(z_2)]\phi_1(z_2) & L[p_k(z_2)]\phi_2(z_2) & \cdots & L[p_k(z_2)]\phi_{n+m}(z_2) \\ \vdots & \vdots & & \vdots \\ L[p_k(z_m)]\phi_1(z_m) & L[p_k(z_m)]\phi_2(z_m) & \cdots & L[p_k(z_m)]\phi_{n+m}(z_m) \end{pmatrix},$$

where L is the linearization of N, *i.e.*, $L[p(x)]\phi(x)$ is the linearization of N about $p(x)$, acting on ϕ at x; in detail,

$$L[p(x)]\phi(x) = \Delta\phi(x) + D_2 f_k(x, p(x), \nabla p(x))\phi(x) + [D_3 f_k(x, p(x), \nabla p(x))]^* \nabla\phi(x),$$

where $D_2 f_k$ denotes the partial derivative of f_k with respect to the second variable, $[D_3 f_k]^*$ denotes the transpose of the vector of the partial derivatives of f_k with respect to the third variable and the variables following the third. Also define

$$\delta c_k = \begin{pmatrix} \delta c_1^k \\ \delta c_2^k \\ \vdots \\ \delta c_{n+m}^k \end{pmatrix}, \quad r_N^k = \begin{pmatrix} Np_k(z_1) \\ Np_k(z_2) \\ \vdots \\ Np_k(z_m) \end{pmatrix}, \quad \delta u_k = \begin{pmatrix} \delta u_1^k \\ \delta u_2^k \\ \vdots \\ \delta u_n^k \end{pmatrix}, \quad \delta v_k = \begin{pmatrix} \delta v_1^k \\ \delta v_2^k \\ \vdots \\ \delta v_n^k \end{pmatrix},$$

and

$$r_u^k = u_k - \Phi^* c_k, \quad r_v^k = v_k - R_\Phi^* c_k.$$

Equations (8.8) and (8.9) can be written as

$$\begin{pmatrix} \Phi^* \\ L_\Phi^* \end{pmatrix} \delta c_k = \begin{pmatrix} \delta u_k + r_u^k \\ -r_N^k \end{pmatrix}. \tag{8.11}$$

We use (8.11) to eliminate δc_k in (8.10), to obtain

$$\delta v_k = R_\Phi^* \begin{pmatrix} \Phi^* \\ L_\Phi^* \end{pmatrix}^{-1} \begin{pmatrix} \delta u_k + r_u^k \\ -r_N^k \end{pmatrix} - r_v^k.$$

We define A and B exactly as in the linear case, so

$$(\Phi | L_\Phi) \begin{pmatrix} A^* \\ B^* \end{pmatrix} = R_\Phi. \tag{8.12}$$

Following this, the equation for δv_k can be rewritten as

$$\delta v_k = (A|B) \begin{pmatrix} \Phi^* \\ L_\Phi^* \end{pmatrix} \begin{pmatrix} \Phi^* \\ L_\Phi^* \end{pmatrix}^{-1} \begin{pmatrix} \delta u_k + r_u^k \\ -r_N^k \end{pmatrix} - r_v^k,$$

so, finally,

$$\delta v_k = A\delta u_k - Br_N^k - r_v^k + Ar_u^k. \tag{8.13}$$

This equation has the same form as (2.15), which is for linear problems. The complete collocation method for nonlinear parabolic PDEs with Newton iteration now consists of the following steps:

1. Use the Rivara algorithm to refine the domain Ω, in order to create a mesh, and then select matching points and collocation points.
2. Use the initial conditions to calculate the initial values of u_0 at the collocation points.
3. For each finite element, calculate the matrix $(A|B)$, using (8.12).
4. Provide approximations to u_k, v_k and c_k.
5. Solve the global set of equations (8.13) for δu_k and δv_k. This can be done through the nested dissection algorithm described in Chapter 4.
6. For each finite element, calculate δc_k using (8.11).
7. Update $u_k \rightarrow u_k + \delta u_k$, $v_k \rightarrow v_k + \delta v_k$ and $c_k \rightarrow c_k + \delta c_k$.
8. Repeat step 5, step 6 and step 7 for a certain number of iterations, until u_k, v_k and c_k are accurate enough.
9. Move to the next time step, and calculate u_{k+1}, v_{k+1}, and c_{k+1}, until the desired find time is reached.

As mentioned in Section 2.4, if the time step is constant, certain types of linear PDEs, to which all the examples given in this thesis belong, only require the matrix $(A|B)$ to be calculated once for each element, since L_Φ is time-independent. The matrix L_Φ in (8.8) is time-dependent, so that the matrix $(A|B)$ needs to be calculated at each time step, and the LU-decomposition of the matrix $(\Phi|L_\Phi)$ is also required for each finite element at each time step.

8.2 $3D$ problems

The parabolic PDEs given in previous chapters are for the $2D$ space-dimension case. Many real PDEs have higher space-dimensions. Using collocation methods for problems with higher dimensions is more difficult. Many problems arise naturally in $3D$, for example, the heat exchange equation in $3D$, which reflects a real physical phenomenon.

First, we want to apply refinement to a domain Ω in $3D$. To subdivide a $3D$ cube into small cubes is easy. But, as mentioned in Section 2.2, cubic elements are not suitable for general domains. Thus we need to use tetrahedral elements.

The problem arises of how to subdivide a tetrahedral region into two parts, as the Rivara algorithm does in 2D via bisection. It has been proven that the Rivara algorithm generates stable triangular meshes, since the minimal angle of the generated mesh is not less than one half of the minimal angle of the original triangular domain. However, we do not know if this is possible in the $3D$ case. The basic idea of the bisection of a tetrahedron is as shown in Figure 8.1.

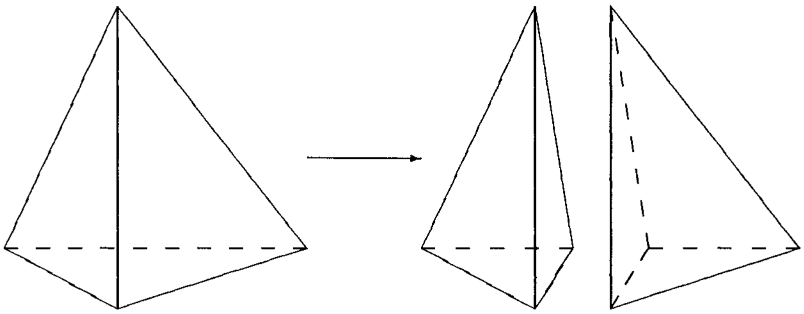


Figure 8.1: A bisection of a tetrahedron.

Many people have studied this problem [44, 45, 46, 47, 48, 49, 50, 51, 52]. Some researchers even used methods of four or eight partitions, which is different from what we want to use. The bisection method described in [47] is the one most similar to the Rivara algorithm.

This method first defines the vertices, edges and faces of a tetrahedron τ as $\mathcal{V}(\tau)$, $\mathcal{E}(\tau)$ and $\mathcal{F}(\tau)$. For a face $\varphi \in \mathcal{F}(\tau)$, $\mathcal{E}(\varphi)$ denotes the edges in φ. A edge is specified as the refinement edge of τ, if it will be divided by a face passing its midpoint and the opposite edge in τ. The two faces intersecting at the refinement edge are called the refinement faces of τ. The method then selects the refinement edge and a particular edge of each of the two nonrefinement faces. These three edges are called marked edges of these faces. So the refinement edge itself is the marked edge of each of the two refinement faces. Also the method gives the tetrahedron a flag, which is unset, except when the marked edges are coplanar.

In this case, the flag may or may not be set. Thus, this method defines this tetrahedron as a marked tetrahedron τ.

This method classifies marked tetrahedra into four types, according to the cases that each marked nonrefinement edge of a marked tetrahedron is either adjacent or opposite to the refinement edge, as shown in Figure 8.2.

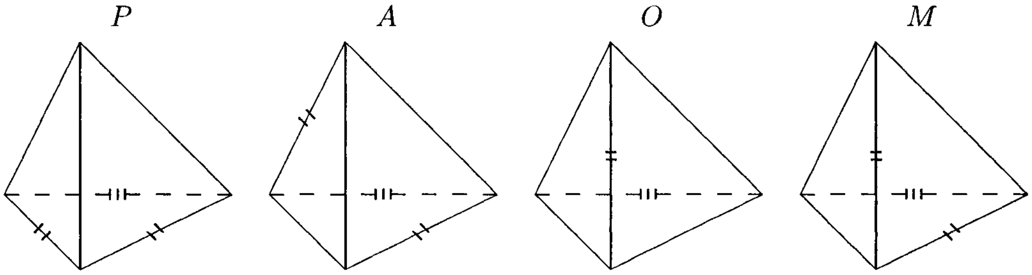


Figure 8.2: Types of marked tetrahedra: P, A, O and M. Each marked edge is indicated by two short bars, while the refinement edge is indicated by three short bars.

- Type P: the marked edges are coplanar. This type tetrahedron is further classified as type P_f or type P_u according to whether the flag is set or not.
- Type A: the marked edges intersect the refinement edge, but are not coplanar.
- Type O: the marked edges of the nonrefinement faces do not intersect the refinement edge and they are the same edge.
- Type M: only one of the the marked edges of the nonrefinement faces intersects the refinement edge.

When a tetrahedron τ is divided into two children, τ_1 and τ_2, a face of one of the children is called an inherited face if it is also a face of τ. A face of τ is called a cut face if it is divided, and the common face of the children is called a new face. Thus, each child has one inherited face, two cut faces and one new face.

Next, this method defines the algorithm *BisectTet*.

Algorithm $\{\tau_1, \tau_2\} = BisectTet(\tau)$

1. Bisect τ by adding a new face passing the midpoint of its refinement edge and the opposite edge. This defines τ_1 and τ_2.

2. The inherited face inherits the marked edge from τ, and this marked edge is the refinement edge of the child.

3. On the cut faces of τ_1 and τ_2, mark the edge which is opposite to the new vertex as marked edge in each cut face.

4. The new face is marked the same way for both τ_1 and τ_2. If τ is type P_f, the marked edge is the edge connecting the new vertex to the new refinement edge. If τ is type P_u, it is the edge which is opposite to the new vertex.

5. The flag is set in τ_1 and τ_2 if and only if τ is type P_u.

This algorithm shows that the tetrahedra τ_1 and τ_2 generated by *BisectTet*(τ) will be of the same type, and they can only be type P or type A.

Finally, this method gives a local refinement procedure which is suitable for calculation of PDE problems in $3D$. It gives the concept of "hanging node", which is similar to the hanging point that we mentioned in Section 3.1. Also, it defines that a $3D$ mesh is conforming if no tetrahedron in it has a hanging node and each face of every tetrahedron in the mesh either belongs to the boundary or is a face of another tetrahedron in the mesh. A mesh is marked if each tetrahedron in it is marked. A marked conforming mesh is conformingly-marked if each face has a unique marked edge. Define a conformingly-marked mesh as $\mathcal{T}$ and define $\mathcal{S}$ as the set of all the marked tetrahedra that need to be refined. The refinement edge of each tetrahedron in $\mathcal{S}$ can be selected based on some criterion, *e.g.*, the longest edge of each tetrahedron. Thus the final refinement algorithm $RefineToConformity$ is

Algorithm $\mathcal{T}' = RefineToConformity(\mathcal{T})$

1. Refine all the marked tetrahedra in $\mathcal{T}$ that need to be refined, which is $\mathcal{S}$, using the algorithm $BisectTet$.

2. If any marked tetrahedron in $\mathcal{T}$ has a hanging node, continue to do *BisectTet*.

3. Repeat until there is no marked tetrahedron having a hanging node. This gives conformingly-marked mesh $\mathcal{T}'$.

There is a proof in [47] to show that procedure *RefineToConformity* will terminate, like the Rivara algorithm.

As to the implementation of the $3D$ case, the present data structures are already suitable. To label the vertices and the faces of each local region, we can use the procedure of the $2D$ case. For example, Figure 8.3 shows how to label a divided local tetrahedral region.

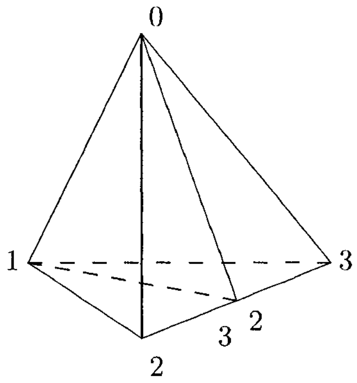


Figure 8.3: Labeling a divided local tetrahedral region.

Here, we do not give the labels of the faces, since they will make the picture unclear. The label of each face of the tetrahedron should be the same as that of the opposite vertex. Next, we need to select the matching points and the collocation points. This will be different from the $2D$ case. The matching points should now be on the faces of the tetrahedra. There will be various ways to select the matching points. As for the collocation points, we can still use equations such as

$$x_p = c_0x_0 + c_1x_1 + c_2x_2 + c_3x_3, \quad c_0 + c_1 + c_2 + c_3 = 1$$

$$c_0, c_1, c_2, c_3 \in R, \quad x_p, x_0, x_1, x_2, x_3 \in R^3,$$

to select random points as collocation points. Thus we can solve (2.19). Since each vertex has been assigned a global index, there will be a way to determine a unique order of the matching points on each face of each local region. Therefore, it is possible to apply the nested dissection method. The procedure will be the same as that for the $2D$ case.

For the definition of the domain Ω, if it is a polyhedron, we can divide Ω into several tetrahedra. Labeling the vertices and faces, like in Section 5.3, is easy for a simple polyhedral domain, as shown in Figure 8.4, where the two tetrahedra share only one common face.

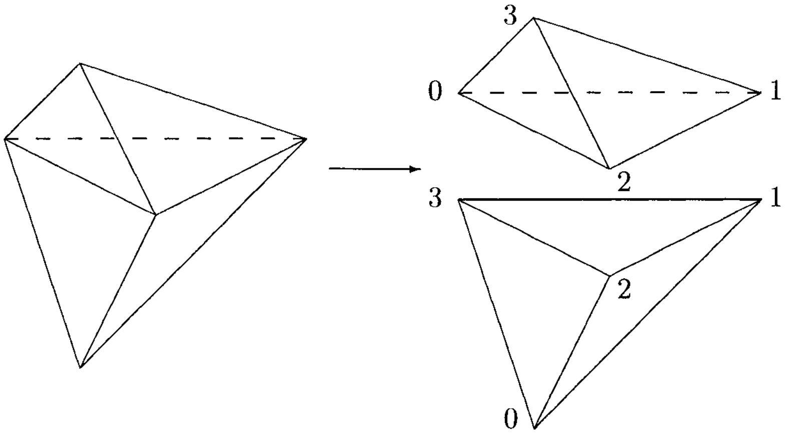


Figure 8.4: A combined region contains two tetrahedra.

Matters will become more difficult when labeling a complicated polyhedral domain, such as the one shown in Figure 8.5. This is a topic for future study.

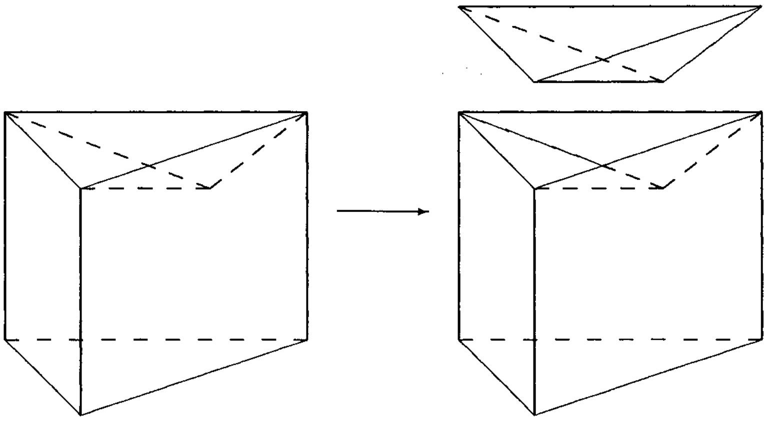


Figure 8.5: A combined region of a more complicated polyhedron, where the two subregions share three common faces.

Chapter 9

Conclusion and Discussion

9.1 Conclusion

In this thesis, we have introduced a new class of collocation methods for the approximate numerical solutions of linear parabolic PDEs. Although the examples solved here are of the form (2.5) for the $2D$ case, the method can solve more general linear parabolic PDEs. The boundary conditions used are Dirichlet boundary conditions and Neumann boundary conditions.

The data structures and many functions, such as allocation, basis generation, *etc.*, are flexible and can be adapted to many problems. This offers a good basis for future work.

At each time step we use a finite difference in time to replace the partial derivative with respect to t, namely, the implicit Euler method. This produces an elliptic PDE at each time step. Using the initial conditions and the boundary conditions, we use a collocation method to solve this elliptic PDE at each time step. At each time step, we use the space mesh generated by the Rivara algorithm. Matching points and collocation points are selected to form a system of finite difference equations for each element. Using nested dissection and applying the boundary conditions, we form a linear system of equations, whose solution gives the solution values at the matching points on the boundary. Next, we perform back substitution to get the solutions at the interior matching points. We can also determine the coefficients of the local polynomial of each element. Thus we can get the solution at any point in the domain. This is needed, in particular, to calculate the solution at the next time step, since we need to know the solution at the collocation points at the present time step.

Solutions of time-independent problems have a higher order of accuracy, because the errors only depend on the space discretization. The accuracy of the solutions of time-dependent problems mainly depend on the time step size because the implicit Euler method used here has the accuracy of $O(dt)$, except when the space mesh is very coarse. Examples in this thesis illustrate this behavior.

Solutions are typically best understood by their graphical representation. This gives us a direct image of the distribution of the solutions, and hence a better understanding of the problem. We can also create animations to show the dynamic changes of the solutions of parabolic PDEs. Animations can be drawn instantly, or be saved in animation files. The solutions can also be saved in certain format files for later reconstruction.

9.2 Future development

At present, our collocation methods can solve general linear parabolic PDEs in $2D$. For real problems, things can be much more complicated. We need to extend the method to be able to solve more general problems.

Selection of the collocation points greatly influence the accuracy of the final solution. The collocation points given in the implementation of this thesis are random points in or near the elements. In test calculations, we find that inappropriate ways of selecting the matching points and the collocations points will cause the matrix $(\Phi|L_\Phi)$ to be singular or have small pivots. The points given in Chapter 6 have given good results. However, the solutions are not as accurate as those from square meshes. Determining more suitable collocation points for triangular meshes is a topic to study.

Most real problems are nonlinear PDEs. Extending the collocation method to nonlinear PDEs will be very useful. In Section 8.1, we have discussed this extension. What needs to be done next is to extend the present software to nonlinear PDEs. Since nonlinear PDEs can be complicated, it is best to start with relatively simple cases.

Extending the software to PDEs in higher space dimensions will also be very useful for real problems. Although square meshes for solving higher dimension PDEs can be directly implemented, using triangular meshes will be much more challenging. In Section 8.2, we have discussed some aspects of the $3D$ case. The first problem is how to refine a $3D$ domain into a fine mesh of acceptable tetrahedral elements. We gave some indications and references on how to do this, but there is no theory that matches that of the Rivara algorithm in $2D$. The process of solving the collocation equations and using the nested dissection algorithm in $3D$ is also a challenge. Although the present data structure is ready for any dimension, the numerical implementation is not easy and needs future work.

Currently, the original triangular mesh is defined manually. If the domain is complicated, defining a corresponding mesh will be difficult. There should be a method to divide the given domain into a suitable initial triangular mesh automatically. This would be very useful for solving real problems.

Adaptive time steps will reduce the cost of the calculations. We need to find practical ways to deal with this problem. Higher order discretization in time is clearly another way to further improve efficiency

Better mesh selection will give more accurate solutions. Also, it will take less computation. An example in Chapter 6 shows a solution of a PDE that has a peak at the center of the domain. Since we already know the exact solution, this can be predetermined. Thus, we use local refinement at the center, to obtain an accurate solution. In real problems, we do not know the solution in advance, so we do not know in advance if a solution is steep at some place in the domain. One choice is to use a fine uniform mesh, but this will cause much calculation time. An adaptive mesh can solve this problem by automatically relocating the mesh to obtain solutions of a specific accuracy, and cost less calculation. The computation can start with a trial solution on a coarse mesh with a basis of low order. Based on error estimates, the mesh can then be refined to obtain the desired accuracy. This problem has been studied by many people [58, 59, 60, 61, 62, 63], but little is known about optimal choices of the mesh. Common procedures studied include:

- local refinement or coarsening of the mesh,

- relocating or moving the mesh (the method of "moving meshes"),

- varying the degrees of the polynomials associated with the elements.

Relocating or moving the mesh is useful for time-dependent problems, such as parabolic PDEs studied here. This is a topic of future study.

The present application can run on Linux systems, and on most Unix systems with few changes. The visualization tool is based on X11/Motif and OpenGL. In order to make the program more flexible, it needs to be ported to other systems. In fact, the present calculation part can be run under most operating systems. This part of the code has already been tested on Win32 systems. Since OpenGL is an operating system independent $3D$ interface, the graphics code can be easily ported to other platforms. What needs change is the graphics frame. Most graphics systems differ, especially for X11 and Windows. Even for X11, there are many different graphics frames, such as Motif, KDE and Gnome. Motif is the most widely used interface on most Unix systems, although it is not so flexible as others.

The current visualization tool can draw the mesh, give $3D$ plots of the numerical solutions, and create animation files. In order to observe the solutions better for further analysis, there need to be more ways to show the results, such as contours, sections, and an animation that shows the process of creating the mesh. These functions need further development.

Parallel processing is a general way to deal with problems that need much calculation time. At present, there are many ways to carry out parallel calculations, such as on mainframes with many nodes, clusters, SMP servers, processors having SIMD instructions, and the multi-core processors, introduced recently. In the implementation of the methods, the easiest place to implement parallel processing is in the matrix calculations. In fact, part of a code that does matrix calculations has already been implemented in Intel's SSE2 codes [65, 66, 67], and it has been proven that SSE2 code can improve the efficiency of the calculations, though only slightly, since the matrix calculations is not the main part of the whole processing. Therefore, implementing parallel calculation needs further consideration.

Bibliography

[1] S.S. Rao. "Applied Numerical Methods for Engineers and Scientists." Prentice Hall, Chapter 11, 2002.

[2] Ames, William F. "Numerical Methods for Partial Differential Equations." Academic Press, Chapter 2, 1977.

[3] "MATLAB Help." MATLAB Release 13, The MathWorks, Inc. 2002.

[4] E.J. Doedel. "On the Construction of Discretizations of Elliptic Partial Differential Equations." Journal Difference Equations and Applications, Vol. 3, pp.389-416, 1998.

[5] E.J. Doedel and Hamid Sharifi. "Collocation Methods for Continuation Problems in Nonlinear Elliptic PDEs." Issue on Continuation Methods in Fluid Mechanics, D. Henry and A. Bergeon, eds., Notes on Numer. Fluid. Mech., Vol. 74, pp.105-118, Vieweg, 2000.

[6] E.J. Doedel. "Finite Difference Collocation Methods for Nonlinear Two Point Boundary Value Problems." SIAM J. Numer. Anal., Vol. 16, pp.173-185, 1979.

[7] Hamid Sharifi. "Collocation Methods for the Numerical Bifurcation Analysis of Systems of Nonliniear Partial Differential Equations." Ph.D thesis, Concordia University, 2005.

[8] C.de Boor and B. Swartz. "Collocation at Gaussian points." SIAM J. Numer. Anal., Vol. 10, pp.582-606, 1973.

[9] U. Ascher and G. Bader. "Stability of Collocation at Gaussian Points." SIAM J. Numer Anal., Vol. 23, pp.412-422, 1986.

[10] U. Ascher, J. Christiansen, and R.D. Russell. "Collocation Software for Boundary-Value ODEs. " ACM Trans. Math. Software, Vol. 7, pp.209-222, 1981.

[11] U. Ascher, J. Christiansen, and R.D. Russell. "COLSYS: Collocation Software for Boundary-Value ODEs [D2]." ACM Trans. Math. Software, Vol. 7, pp.223-229, 1981.

[12] C.E. Greenwell-Yanik and G. Fairweather. "Analyses of Spline Collocation Methods for Parabolic and Hyperbolic Problems in Two Space Variables." SIAM J. Numer. Anal., Vol. 23, pp.282-296, 1986.

[13] P.M. Prenter and R.D.Russell. "Orthogonal Collocation for Elliptic Partial Differential Equations." SIAM J. Numer. Anal., Vol. 13, pp.923-939, 1976.

[14] Jim Douglas, Jr. and Todd Dupont. "A Finite Element Collocation Method for Quasilinear Parabolic Equations." Mathematics of Computation, Vol. 27, pp.17-28, 1973.

[15] R.E. Bank. "The Efficient Implementation of Local Mesh Refinement Algorithms." InI. Babuška, J. Chandra, and J.E. Flaherty, editors, Adaptive Computational Methods for Partial Differential Equations, pp.74-81, Philadelphia, 1983. SIAM.

[16] R.E. Bank. "PLTMG: A Software Package for Solving Elliptic Partial Differential Equations.Users' Guide 7.0." Volume 15 of Frontiers in Applied Mathematics. SIAM, Philadelphia, 1994.

[17] R.E. Bank. A.H. Sherman, and A. Weiser. "Refinement algorithms and data structures for regular local mesh refinement." Scientific Computing, pp.3-17, Brussels, 1983, IMACS/North Holland, Netherland.

[18] R.E. Bank and R. Kent Smith. "A Posteriori Error Estimates Based on Hierarchical Bases." SIAM J. Numer. Anal., Vol. 30, pp.921-935, 1993.

[19] R.E. Bank and R. Kent Smith. "Mesh Smoothing Using a Posteriori Error Estimates." SIAM J. Numer. Anal., Vol. 34, pp.979-997, 1997.

[20] M.G. Larson. "Notes on Triangulations and Refinements" available by HTTP from http://www.phi.chalmers.se/education/courses/2000/detb-a-kf/Triang.ps, Chalmers University of Technology, Sweden, May 8, 2001.

[21] M.-C. Rivara. "Mesh Refinement Porcesses Based on The Generalized Bisection of Simplices." ACM Transactions on Mathematical Software, Vol. 10, pp.242-264, 1984.

[22] M.-C. Rivara. "Design and Data Structure of Fully Adaptive, Multigrid, Finite-Element Software." SIAM Journal on Numerical Analysis, Vol. 21, pp.604-613, 1984.

[23] J.E. Flaherty, "Finite Element Analysis." Lecture 8, available by HTTP from http://www.cs.rpi.edu/~flaherje/pdf/fea8.pdf, Rensselaer Polytechnic Institute, 2005.

[24] I.G. Rosenberg and F. Stenger. "A Lower Bound on the Angles of Triangles constructed by bisecting the longest side." Mathematics of Computation, Vol. 29, pp.390-395, 1975.

[25] Martin Stynes. "On Faster Convergence of the Bisection Method for Certain Triangles." Mathematics of Computation, Vol. 33, No. 146, pp.717-721, 1979.

[26] Martin Stynes. "On Faster Convergence of the Bisection Method for all Triangles." Mathematics of Computation, Vol. 35, No. 152, pp.1195-1201, 1980.

[27] Kenneth Hoffman and Ray Kunze. "Linear Algebra." Prentice-Hall, 1971.

[28] E.J. Doedel. "Elementary Numerical Methods, Lecture Notes." available by HTTP from http://cmvl.cs.concordia.ca/courses/comp-361/fall-2005/notes.pdf, Concordia University, 2005.

[29] "GCC 4.0.1 Manual." available by HTTP from http://gcc.gnu.org/onlinedocs/gcc-4.0.1/gcc/, GNU, 2005.

[30] Arthur Griffith. "GCC:the Complete Reference." McGraw-Hill, 2002.

[31] "Using LD, the GNU linker." available by HTTP from http://www.gnu.org/software /binutils/manual/ld-2.9.1/ld.html, GNU, 1998.

[32] "Fedora3 Linux Online Help." available by HTTP from http://fedora.redhat.com, Fedora3 manual, 2004.

[33] Ulrich Drepper and Ingo Molnar. "The Native POSIX Thread Library for Linux." Redhat, Inc. February 2003.

[34] Arnaud Desitter."Using Static and Shared Libraries Across Platforms." available by HTTP from http://fortran-2000.com/ArnaudRecipes/sharedlib.html, 2005

[35] Amal Shah and Hong Xiao."Using Shared Libraries across Platforms." available by HTTP from http://www.cuj.com/documents/s=8065/cuj9805shahxiao/, May 1998.

[36] "Development Tools and Languages/Visual Studio 6.0/Visual C++ Programmer's Guide/Compiling and Linking." MSDN Library, available by HTTP from http://msdn.microsoft. com/library, Microsoft, 2005.

[37] "Win32 and COM Development/System Services/DLLs, Processes, and Threads/SDK Documentation/DLLs, Processes, and Threads/Dynamic-Link Libraries." MSDN Library, available by HTTP from http://msdn.microsoft.com/library, Microsoft, 2005.

[38] Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis. "OpenGL Programming Guide, Second Edition." Addison-Wesley, 1997.

[39] M.J. Kilgard."OpenGL Programming for the X Window System." Addison-Wesley, 1996.

[40] Mark Segal and Kurt Akeley. "The OpenGL Graphics System: A Specification, Version 1.5." Silicon Graphics, Inc. 2003.

[41] Paula Womack and Jon Leech. "OpenGL Graphics with the X Window System, Version 1.3." Silicon Graphics, Inc. 1998.

[42] "GIF(tm), Graphics Interchange Format(tm), A standard defining a mechanism for the storage and transmission of raster-based graphics information." CompuServe Incorporated, 1987.

[43] "Graphics Interchange Format(sm), Version 89a." CompuServe Incorporated, 1989.

[44] Angel Plaza, M.A. Padrón and G.F. Carey. "A 3D Refinement/Derefinement Algorithm for Solving Evolution Problems" Applied Numerical Mathematics, Vol. 32, pp.401-418, 2000.

[45] Ronaldo Marinho Persiano, João Luiz Dihl Comba and Valéria Barbalho. "An Adaptive Triangulation Refinement Scheme and Construction." available by HTTP from http://graphics.stanford.edu/~comba/papers/adptri93.pdf, Proceedings of the VI Sibgrapi (Brazilian Symposium on Computer Graphics and Image Processing), Recife, Brazil, October 1993.

[46] Angel Plaza and G.F. Carey. "About Local Refinement of Tetrahedral Grids Based on Bisection." available by HTTP from http://www.andrew.cmu.edu/user /sowen/abstracts/Pl246.html, 5th International Meshing Roundtable, Sandia National Laboratories, pp.123-136, October 1996.

[47] D.N. Arnold, Arup Mukherjee and Luc Pouly. "Locally Adapted Tetrahedral Meshes Using Bisection" SIAM J. Sci. Comput. Vol. 22, No. 2, pp.431-448, 2000.

[48] Angel Plaza and M.-C., Rivara. "Mesh Refinement Based on The 8-Tetrahedra Longest-edge Partition." available by HTTP from http://www.andrew.cmu.edu/user /sowen/abstracts/Pl977.html, 2th International Meshing Roundtable, Sandia National Laboratories, pp.67-78, Sept. 2003.

[49] A. Plaza and G.F. Carey. "Local Refinement of Simplicial Grids Based on the Skeleton." Applied Numerical Mathematics, Vol 32, pp.195-218, 2000.

[50] Anwei Liu and Barry Joe. "On the Shape of Tetrahedra from Bisection." Mathematics of Computation, Vol. 63, No. 207, pp.141-154, 1994.

[51] Anwei Liu and Barry Joe. "Quality Local Refinement of Tetrahedral Meshes Based on 8-Subtetrahedron Subdivision." Mathematics of Computation, Vol. 65, No. 215, pp.1183-1200, 1996.

[52] M.-C. Rivara and Gabriel Iribarren. "The 4-Triangles Longest-side Partition of Triangles and Linear Refinement Algorithms." Mathematics of Computation, Vol 65, No 216, pp.1485-1502, October 1996.

[53] J.P. Suárez, A. Plaza and G.F. Carey. "Propagation Path Properties in Iterative Longest-edge Refinement." available by HTTP from http://www.andrew.cmu.edu /user/sowen/abstracts/Su983.html, 12th International Meshing Roundtable, Sandia National Laboratories, pp.79-90, Sept. 2003.

[54] R.D. Russell and J. Christiansen. "Adaptive Mesh Selection Strategies for Solving Boundary Value Problems." SIAM J. Numer. Anal., Vol. 15, pp.59-80, 1978.

[55] Kenneth Eriksson and Claes Johnson. "An Adaptive Finite Element Method for Linear Elliptic Problems." Mathematics of Computation, Vol. 50, No. 182, pp.361-383, 1988.

[56] Kenneth Eriksson and Claes Johnson. "Adaptive Finite Element Methods for Parabolic Problems I: A Linear Model Problem." SIAM J. Numer. Anal., Vol. 28, pp.43-77, 1991.

[57] Kenneth Eriksson and Claes Johnson. "Adaptive Finite Element Methods for Parabolic Problems IV: Nonlinear Problems." SIAM J. Numer. Anal., Vol. 32, pp. 1729-1749, 1995.

[58] I. Babuška, J. Chandra and J.E. Flaherty, editors. "Adaptive Computational Methods for Partial Differential Equations." SIAM, Philadelphia, 1983.

[59] I. Babuška, J.E. Flaherty, W.D. Henshaw, J.E. Hopcroft, J.E. Oliger, and T.Tezduyar, editors. "Modeling, Mesh Generation and Adaptive Numerical Methods for Partial Differential Equations." Volume 75 of The IMA Volumes in Mathematics and its Applications, Springer-Verlag, New York, 1995.

[60] I. Babuška, O.C. Zienkiewicz, J. Gago and E.R. de A. Oliveira, editors, "Accuracy Estimates and Adaptive Refinements in Finite Element Computations." John Wiley and Sons, Chichester, 1986.

[61] J.E. Flaherty, P.J. Paslow, M.S. Shephard and J.D. Vasilakis, editors, "Adaptive methods for Partial Differential Equations." SIAM, Philadelphia, 1989.

[62] R. Verfürth. "A Review of Posteriori Error Estimation and Adaptive Mesh Refinement Techniques." Teubner-Wiley, Stuttgart, 1996.

[63] M.W. Bern, J.E. Flaherty and M. Luskin, editors, "Grid Generation and Adaptive Algorithms." Volume 113 of The IMA Volumes in Mathematics and its Applications, New York, 1999, Springer.

[64] W. Dorfler and M. Rumpf. "An Adaptive Strategy for Elliptic Problems Including a Posteriori Controlled Boundary Approximation." Mathematics of Computation, Vol. 67, No. 224, pp.1361-1382, 1998.

[65] "IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture." http://www.intel.com, Intel, Inc. 2004.

[66] "IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference." http://www.intel.com, Intel, Inc. 2004.

[67] "IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide." http://www.intel.com, Intel, Inc. 2004.

Appendix A

Data Structures

A.1 Vertex structure

```
typedef struct Vertex{
    int index;
    int ecount;
    int ecountc;
    int mcount;
    int ndim;
    double minsol;
    double maxsol;
    double solerr;
    double solution;
    double *coord;
}VERTEX;
```

Members:

index: The global index of this vertex, starting from zero.

ecount: The number of the elements containing this vertex.

ecountc: The current count of the elements found containing this vertex when going through the binary tree.

mcount: The number of local regions having this vertex as their longest edges' middle points. For $2D$ case, it is two. This member is used when releasing a middle point. Each time when trying to release the vertex, it reduces one. When it gets to zero, this vertex will be released.

ndim: The dimension of the vertex.

minsol: The minimal solution solved from the local polynomials of the elements containing this vertex.

maxsol: The maximal solution solved from the local polynomials of the elements containing this vertex.

solerr: The difference between maxsol and minsol.

solution: The average of all the solutions solved from the local polynomials of the elements containing this vertex.

coord: Pointer to the coordinates of this vertex; the size of this member can be variable according to different dimension.

A.2 Face structure

```
typedef struct Face FACE;
struct _Face{
    int index;
    int nmatp;
    MATPOINT  **matpts;
    REGION *region[2];
    FACE *next[2];
    VERTEX *comvert;
    int ndim;
    VERTEX **vertex;
    double *normal;
};
```

Members:

index: The global index of this face, starting from zero.

nmatp: The number of the matching points contained in this face.

matpts: The pointer to the list of the structures of the matching points contained in this face.

region: The pointers to the structures of the two regions which share this face as the common face. If this face is a part of the boundary, the pointer pointing to the outside region will be NULL.

next: The pointers to the structures of the two child faces. If this face is a face of the element, these two pointers will be NULL.

comvert: Pointer to the structure of the vertex which divides this face. If this face is a face of the element, the pointer will be NULL.

ndim: The dimension of this face.

vertex: Pointer to the structures of the vertices contained in this face.

normal: Pointer to the structure of the exterior normal vector of this face.

```
typedef struct MatPoint{
    double *coord;
    double u, du;
    double v, dv;
}MATPOINT;
```

Members:

coord: Pointer to the coordinates of this matching point.

u: The value of u_k at this matching point.

du: The value of δu_k at this matching point. This variable is used for nonlinear PDEs.

v: The value of v_k at this matching point.

dv: The value of δv_k at this matching point. This variable is used for nonlinear PDEs.

A.3 Region structure

```
typedef struct Region REGION;

struct Region{

    int index;

    int drawn;

    REGIONTYPE regiontype;

    int nmatp;

    double **A, **B;

    double *abdata;

    int *ani, *aci;

    int va;

    double *fvp;

    REGION *next[2];

    int ndim;

    VERTEX **vertex;

    FACE **face;

    int nfaces;

    int *vertorder;

    int *comface;

    int ncommon;

    int nnvert;

    BASICREGIONDATA *bgdata;

    ELEMENTREGIONDATA *egdata;

    NESTEDREGIONDATA *ngdata;

};
```

Members:

index: The global index of this region, starting from zero.

drawn: The indicator to show if this region has been drawn. It is used for plotting mesh.

regiontype: The type of this region. If this region is a local triangular region, the type is BASIC, if this region is a polygonal region, the type is COMBINED.

nmatp: The number of the matching points contained in this region.

A: Pointer to the matrix A of this region.

B: Pointer to the matrix B of this region.

abdata: Pointer to the data area of the matrix $(A|B)$ of this region.

ani: Pointer to list of the labels of the matching points in the non common faces.

aci: Pointer to list of the labels of the matching points in the common faces.

va: The indicator to show if the matrix A has been calculated. If it has been calculated, it needs not to be recalculated.

fvp: Pointer to the vector f_{ck} of this region.

next: The pointers to the structures of the two child regions. If this region is an element, these two pointers will be NULL.

ndim: The dimension of this region.

vertex: Pointer to the structures of the vertices contained in this region.

face: Pointer to the structures of the faces contained in this region.

nfaces: The number of the faces contained in this region.

vertorder: Pointer to the list of the order of the vertices contained in this region.

comface: Pointer to the structures of the common faces of this region.

ncommon: The number of the common faces of this region.

nnvert: The number of the new vertices used to create this region. In some cases, when this has more than one common face, no new vertex is needed to create this region.

bgdata: Pointer to the structure of the basic data of this region, if this region is a local triangular region.

egdata: Pointer to the stucture of the element data of this region, if this region is an element.

ngdata: Pointer to the structure of the non element data of this region, if this region is not an element.

```
typedef struct BasicRegionData{
    int longv0, longv1;
    double longlength;
    double *center;
    double *weight;
    double *midpoint;
    int *ndface;
    double *xmin, *xmax;
    double *grad;
}BASICREGIONDATA;
```

Members:

longv0: The global index of one of the vertex of the longest edge of this region.

longv1: The global index of the other vertex of the longest edge of this region.

longlength: The length of the longest edge of this region.

center: Pointer to the coordinates of the center of this region.

weight: Pointer to the intermediate coefficients to calculate the coordinates of the center of this region.

midpoint: Pointer to the coordinates of the point of the longest edge of this region.

ndface: Pointer to the list of global indices of the faces of this region which are divided.

xmin: Pointer to the intermediate data helping to do local refinement.

xmax: Pointer to the intermediate data helping to do local refinement.

grad: Pointer to the intermediate data helping to calculate matrix R_Φ.

```
typedef struct NestedRegionData{
    int *g0i, *g1i;
    int nr0ematp;
    int nr1ematp;
    double **bmlu;
    double *bmdata;
    int *bmri, *bmci;
    double *fc;
};
```

Members:

g0i: Pointer to the list of the labels of the matching points coming from the left child of this region.

g1i: Pointer to the list of the labels of the matching points coming from the right child of this region.

nr0ematp: The number of the matching points coming from the left child of this region.

nr1ematp: The number of the matching points coming from the right child of this region.

bmlu: Pointer to the matrix B_m of this region.

bmdata: Pointer to the data area of the matrix B_m of this region.

bmri: Pointer to the list of the labels of the rows of the matrix B_m of this region.

bmci: Pointer to the list of the labels of the columns of the matrix B_m of this region.

fc: Pointer to the vector f_{ck}^m of this region.

```
typedef struct ElementRegionData{
    double **colpts;
    double **phillu;
    double **lphi;
    double *phildata;
    int *philri, *philci;
    double *C;
    double *fe;
    int nvplot;
    double *solvplot;
}ELEMENTREGIONDATA;
```

Members:

colpts: Pointer to the coordinates of the collocation points of this region.

phillu: Pointer to the matrix $(\Phi|L_\Phi)$ of this region.

lphi: Pointer to the matrix L_Φ of this region.

phildata: Pointer to the data area of the matrix $(\Phi|L_\Phi)$ of this region.

philri: Pointer to the list of the labels of the rows of the matrix $(\Phi|L_\Phi)$ of this region.

philci: Pointer to the list of the labels of the columns of the matrix $(\Phi|L_\Phi)$ of this region.

C: Pointer to the vector of coefficients of the local polynomial of this region.

fe: Pointer to the vector f_k of this region.

nvplot: The number of the points to plot this region. It is used for plotting 3D graphics.

solvplot: Pointer to the solutions of the points (x, y, u) to plot this region. It is used for plotting 3D graphics.

A.4 PDE parameters structure

```
typedef struct _PdeParam
{
    int vdata;
    int ndimen;
    int nmatch;
    int ncollo;
    int nmatel;
    int nbasfun;
    int nvert;
    int nfaces;
    int nedges;
    double meps;
    int mrefcr;
    double mrefrad;
    int nverts;
    double *rgvcd;
    int nrgs;
    int *rgncf;
    int *rgrlt;
    int *rgnnv;
    int nrevs;
    int *rgrev;
    int ncfvs;
    int *rgcfv;
    double titval[2];
    double tstep;
    char *pdefmn;
    int colcr;
    int mtpcr;
    void *pdemod;
    int cpuid;
    USERFUNC userf;
    USERINITFUNC useri;
    USERBOUNDFUNC userb;
    USERGEFUNC gradexact;
    USEREXACTFUNC exact;
    USERCOLFUNC usercol;
    USERPLOTFUNC userplot;
    USERENDFUNC userend;
    void *plotparam;
    int nplot;
    void *outparam;
}PDEPARAM;
```

Members:

vdata: The indicator to show if the parameters have been initialized.

ndimen: The dimension of the space.

nmatch: The number of matching points per face of a element.

ncollo: The number of collocation points per element.

nmatel: The number of matching points per element.

nbasfun: The number of basis functions.

nvert: The number of vertices per element.

nfaces: The number of faces per element.

nedges: The number of edges per element.

meps: The upper limit of the longest edge of the elements, d_m.

mrefcr: The indicator to show the criterion of mesh refinement.

mrefrad: The upper limit of the distance from the center of any region that needs to be refined to the reference point or line, d_r.

nverts: The number of vertices which define the original mesh in the domain.

rgvcd: Pointer to the coordinates of the vertices which define the original mesh in the domain.

nrgs: The number of triangular regions in the original mesh. These regions can be $3D$ or higher.

rgncf: Pointer to the list of the number of common faces shared by the triangular regions in the original mesh. These regions can be $3D$ or higher.

rgrlt: Pointer to the list of indicators of the relation of the triangular regions in the original mesh. These regions can be $3D$ or higher.

rgnnv: Pointer to the list of the number of new vertices used to create the triangular regions in the orginal mesh. These regions can be $3D$ or higher.

nrevs: The number of the old vertices used to create triangle regions which do not need to use new vertices.

rgrev: Pointer to the global indices of the old vertices used to create triangle regions which do not need to use new vertices.

ncfvs: The number of the local labels of the vertices to determine the second common face in triangular regions if these elements have two common faces. At present, this member is only used for $2D$ case.

rgcfv: Pointer to the local labels of the vertices to determine the second common face in triangular regions if these elements have two common faces. At present, this member is only used for $2D$ case.

titval: Pointer to the time domain.

tstep: The time step size.

pdefmn: The name of the module file which offers the user defined function, the initial conditions function, the boundary conditions function and the exact solution function if it is know.

colcr: The indicator to show the criterion of the collocation points generation.

mtpcr: The indicator to show the criterion of the matching points generation.

pdemod: Pointer to the address of the module which offers the user defined function, the initial conditions function, the boundary conditions function, and the exact solution function if it is know.

cpuid: The indicator to show the CPU type. This is used for calculation with SSE2 instructions.

userf: Pointer to the user defined function.

useri: Pointer to the initial condition function.

userb: Pointer to the boundary condition function.

gradexact: Pointer to the exact gradient solution function.

exact: Pointer to the exact solution function.

usercol: Pointer to the collocation points generation function.

userplot: Pointer to the plotting function.

userend: Pointer to the exiting function. This is used for terminating the calculation thread.

plotparam: Pointer to the data of the solutions to do $3D$ plotting.

nplot: The number of the solutions.

outparam: Pointer to the semaphore.

A.5 Configuration file

The configuration file gives the values of some members of the PDEPARAM structure. In the configuration, if a line starts with "#", it is a comment line, otherwise, the value is a valid parameter. The format of the parameter is

```
<parameter's name> = value
```

The following are the names of the parameters and their meaning.

```
NDIM    NMAT    NCOL    MEPS
MREF    MRRD    RVCD    RNCF
RRLT    RNNV    RREV    RCFV
```

```
TINV      TSTP      PMOD      COLC
MTPC
```

Parameters:

NDIM: The dimension of the space.

NMAT: The number of matching points per face of a element.

NCOL: The number of collocation points per element.

MEPS: The upper limit of the longest edge of the elements, d_m.

MREF: The indicator to show the criterion of mesh refinement.

MRRD: The upper limit of the distance from the center of any region that needs to be refined to the reference point or line, d_r.

RVCD: Pointer to the coordinates of the vertices which define the original mesh in the domain.

RNCF: Pointer to the list of the number of common faces shared by the triangular regions in the original mesh. These regions can be $3D$ or higher.

RRLT: Pointer to the list of indicators of the relation of the triangular regions in the original mesh. These regions can be $3D$ or higher.

RNNV: Pointer to the list of the number of new vertices used to create the triangular regions in the orginal mesh. These regions can be $3D$ or higher.

RREV: Pointer to the global indices of the old vertices used to create triangle regions which do not need to use new vertices.

RCFV: Pointer to the local labels of the vertices to determine the second common face in triangular regions if these elements have two common faces. At present, this member is only used for $2D$ case.

TINV: Pointer to the time domain.

TSTP: The time step size.

PMOD: The name of the module file which offers the user defined function, the initial conditions function, the boundary conditions function and the exact solution function if it is know.

COLC: The indicator to show the criterion of the collocation points generation.

MTPC: The indicator to show the criterion of the matching points generation.

Appendix B

Useful Functions

B.1 Matrix allocation

B.1.1 Alloc2D

```
/* Allocate a matrix and return the address.
   Input:  n1, rows. n2, columns. usize, the size of each item.
   Output: dp, pointer to the data area. */
void *Alloc2D(int n1, int n2, int usize, void **dp)
{
    void *p1, **p2;
    char *p3;
    register int i;
    p1 = malloc(n1 * sizeof(void *) +  n1 * n2 * usize);
    if(p1 == NULL)
        return p1;
    p2 = (void **)p1;
    p3 = (char *)p1 + n1 * sizeof(void *);
    for(i = 0; i < n1; i++)
        p2[i] = (void *)(p3 + i * n2 * usize);
    *dp = (void *)p3;
    return p1;
}
```

B.1.2 AllocMulCol2D

```
/* Allocate a matrix divided into parts of same rows and return
   the address.
   Input:  n1, rows. n2, columns. usize, the size of each item.
           ncol, the numbers of columns of partial matrices.
           nm, the number of partial matrices.
   Output: mt, pointer to the partial matrices.
           dp, pointer to the data area. */
void *AllocMulCol2D(int n1, int n2, int usize, void*** mt,
                    int *ncol, int nm, void **dp)
{
    void *p1, **p2;
    char *p3;
    register int i, j, offset;
    p1 = malloc(nm * n1 * sizeof(void *) +  n1 * n2 * usize);
    if(p1 == NULL)
       return p1;
    p2 = (void **)p1;
    p3 = (char *)p1 + nm * n1 * sizeof(void *);
    offset = 0;
    for(i = 0; i < nm; i++)
    {
        for(j = 0; j < n1; j++)
         p2[j] = (void *)(p3 + (j * n2 + offset) * usize);
     mt[i] = p2;
     p2 = p2 + n1;
     offset = offset + ncol[i];
    }
    *dp = (void *)p3;
    return p1;
}
```

B.2 Basis Generation

```
/* BasePwr, pointer to the exponents of the basis.
   D1BasePwr, pointer to the exponents of the first order
              partial derivative of the bases.
   D2BasePwr, pointer to the exponents of the second order
              partial derivative of the bases.
   C1BasePwr, pointer to the coefficients of the first order
              partial derivative of the bases.
   C2BasePwr, pointer to the coefficients of the second order
              partial derivative of the bases.  */
static int **BasePwr;
static int ***D1BasePwr, ***D2BasePwr;
static double **C1BasePwr, **C2BasePwr;
/* Calculate the exponents and the coefficients of the bases,
   the first order partial derivative and the second order partial
   derivative.
   Input: nbasf, the number of the bases.
   nDim, the dimension of the bases  */
int CreateBasisND(int nbasf, int nDim)
{
    register int i;
    int degree;
    int *degInds;
    int cbasf;
```

```
void *dp;
int nbasf, nDim;
BasePwr = NULL;
C1BasePwr = NULL;
C2BasePwr = NULL;
D1BasePwr = NULL;
D2BasePwr = NULL;
cbasf = 0;
BasePwr = (int **)Alloc2D(nbasf, nDim, sizeof(int), &dp);
if(BasePwr == NULL)
    return False;
C1BasePwr = (double **)Alloc2D(nbasf, nDim, sizeof(double), &dp);
if(C1BasePwr == NULL)
    return False;
C2BasePwr = (double **)Alloc2D(nbasf, nDim, sizeof(double), &dp);
if(C1BasePwr == NULL)
    return False;
D1BasePwr = (int ***)Alloc3D(nbasf, nDim, nDim, sizeof(int), &dp);
if(D1BasePwr == NULL)
    return False;
D2BasePwr = (int ***)Alloc3D(nbasf, nDim, nDim, sizeof(int), &dp);
if(D2BasePwr == NULL)
    return False;
degInds = (int *)malloc(nDim * sizeof(int));
if(degInds == NULL)
    return False;
for(i = 0; i < nDim; i++)
    degInds[i] = 0;
degree = 0;
while(1)
```

```
    {
        MakeBasis(degInds, 0, nDim, degree, &cbasf, nbasf);
        if(cbasf == nbasf)
            break;
        degree++;
    }
    free(degInds);
    return True;
}
/* Intermediate recursive function to calculate the exponents
   and the coefficients of the basis and partial derivatives */
static void MakeBasis(int *degInds, int allocatedDim, int nDim,
                      int allocDeg, int *curBas, int numBas)
{
    register int i, j, k, cur;
    int remainDeg, remainDim;
    remainDim = nDim - allocatedDim;
    for(i = 0; i <= allocDeg; i++)
    {
        remainDeg = i;
        degInds[allocatedDim] = allocDeg - i;
        if(remainDim == 2 || remainDeg == 0)
        {
            if(remainDim == 2)
                degInds[allocatedDim + 1] = i;
            else if(remainDeg == 0)
            {
                for(j = allocatedDim + 1; j < nDim; j++)
                    degInds[j] = 0;
            }
    cur = *curBas;
    for(j = 0; j < nDim; j++)
    {
        BasePwr[cur][j] = degInds[j];
        C1BasePwr[cur][j] = degInds[j];
        if(degInds[j] == 0 || degInds[j] == 1)
            C2BasePwr[cur][j] = 0.0;
        else
            C2BasePwr[cur][j] = degInds[j] *
                                (degInds[j] - 1);
        for(k = 0; k < nDim; k++)
        {
            if(k == j)
            {
                D1BasePwr[cur][j][k] = degInds[k] - 1;
                if(D1BasePwr[cur][j][k] < 0)
                    D1BasePwr[cur][j][k] = 0;
                D2BasePwr[cur][j][k] = degInds[k] - 2;
                if(D2BasePwr[cur][j][k] < 0)
                    D2BasePwr[cur][j][k] = 0;
            }
            else
            {
                D1BasePwr[cur][j][k] = degInds[k];
                D2BasePwr[cur][j][k] = degInds[k];
            }
        }
    }
    (*curBas)++;
}
```

```
        else
            MakeBasis(degInds, allocatedDim + 1, nDim, remainDeg,
                      curBas, numBas);
        if(*curBas == numBas)
            break;
    }
}
```