Scattered surface data estimation and visualization

Weibao Wu

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/251

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Scattered surface data estimation and visualization

Wu, Weibao, M.S.
University of Nevada, Las Vegas, 1992
Scattered Surface Data Estimation and Visualization

by

Weibao Wu

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Department of Computer Science
University of Nevada, Las Vegas
December, 1992
The thesis of Weibao Wu for the degree of Master of Science in Computer Science is approved.

Chairperson, Evangelos A. Yfantis Ph. D.

Examinig Committee Member, Kia Makki Ph. D.

Examinig Committee Member, Laxmi P. Gewali Ph. D.

Graduate Representative, Peter Shiue Ph. D.

Graduate Dean, R. W. Smith Ph. D.

University of Nevada, Las Vegas

November., 1992
ABSTRACT

Scattered surface data visualization and estimation is an important part of Scientific data visualization. Many efforts have been made on the new methods for scattered data interpolation, estimation and visualization in recent years.

In the thesis, two new interpolation methods, i.e., extended piecewise Hardy's method and Bezier triangle method, are developed for scattered surface data and the biquadratic method is developed for regular data set. A new contour method which can generate arbitrarily smooth contour line is also presented. The new methods are compared with the existing ones with real scattered data set, supplied by the DOE (Department of Energy), via DRI (Desert Research Institute) and computer simulated data. The results show that the extended piecewise Hardy's method is the most accurate method among all the methods compared, and it is much faster than the original Hardy's method. The speedup is approximately of the order $O(n^2)$. The biquadratic method is approximately three times faster than the bicubic splines, and both generate comparable result for the tested data.

A graphics package for scattered data interpolation and visualization with the above methods is implemented on Sun SPARC station under UNIX running X windows as well as IBMPC under the MSDOS. The package is written in C.
Acknowledgment

I would like to thank Dr. R. Jacobson and J. Hall, scientists of DOE, for giving the opportunity to do the project, and thank many scientists in DOE and DRI for their guidance. I would like to express my gratitude to my adviser, Dr. Evangelos A. Yfantis, for his advice on my thesis work and the support, and my committee members, Dr. Kia Makki, Laxmi Gewali and Peter Shiue for taking time to read my thesis and giving valuable suggestions.

The project is supported by the DOE, Las Vegas Laboratory.
## Contents

1 Introduction ................................................................. 1

2 Rectangular Grad Surface Data Interpolation ................. 5
   2.1 The Problem ......................................................... 5
   2.2 The Existing Method ............................................ 6
      2.2.1 Deterministic Method: Cubic Splines and Surfaces ...... 6
      2.2.2 Stochastic Method: Kriging ............................... 11
   2.3 The Quadratic Method and Its Extensions ................. 13
      2.3.1 The Quadratic Method ..................................... 13
      2.3.2 Extensions ..................................................... 15
   2.4 Comparisons ....................................................... 17
      2.4.1 Practical Considerations ................................. 17
      2.4.2 Experimental Results .................................... 18

3 Scattered Surface Data Interpolation ............................ 23
   3.1 The problem ....................................................... 23
   3.2 Triangulation of Scattered Data ............................ 24
      3.2.1 Delaunay Triangulation ................................. 24
      3.2.2 The Delaunay Triangulation ............................. 26
      3.2.3 Inserting Point To The Triangle Graph ............. 31
   3.3 The Existing Interpolation Methods ....................... 32
      3.3.1 Gauss Interpolation ....................................... 32
      3.3.2 Hardy Interpolation ...................................... 33
   3.4 The New methods ................................................. 33
      3.4.1 Extended Piecewise Hardy Interpolation ............. 33
      3.4.2 Bezier Triangle Interpolation .......................... 35
Chapter 1

Introduction

There are basically two kinds of data in surface and volume estimation according to the topology of the data grid. The most widely used data set are topologically rectangular grid. This kind of data can also be subdivided into rectangle grid and irregular grid data. The rectangle grid data set has the same length between two adjacent data points along \( x \), and\( y \) directions for surface data and \( x \), \( y \), and \( z \) for volume data. The irregular data set has the same rectangular topological structure but the length between two adjacent data points may not be the same. The second kind of data which is topologically different from the first is the scattered data. In scattered data set, there is no obvious relation between data points, it may be randomly distributed over surface or volume[23]. The diagram for the rectangle, irregular and scattered data set is shown in Figure 1.1.

The topologically rectangle data is well understood and widely used in data estimation and computer aided geometric design. By the nature of scattered data, it is more difficult to do surface estimation and interpolation. However, the scattered data set is encountered in various areas, scientific research, industry, medical diagnosis, etc., for interpolating experimental data, estimating the ore reservation and pollution concentration. In many practical occasions, some data points may not be

![Diagram showing rectangular grid, irregular grid, and scattered data](image-url)

Figure 1.1: The rectangular and scattered data.
available for various reasons, they may be too expensive, or may be impossible to get.

The study of scattered data estimation and interpolation was stimulated by the advent of faster computers in the 70's, and the needs to solve many real problems with computer graphics. New methods are still being developed in recent years[12, 20]. Usually, the algorithm for scattered data estimation is more complicated than the regular ones.

By the scattered nature, it is hard to apply tensor product methods as we usually do in the rectangular data set in the surface or volume estimation for the scattered data. New method must be developed to handle the scattered set. One approach is to treat the data as a whole, fit the data into one function by solving a system of equations. One of the best in such approach is the Hardy method developed by Hardy[11]. It is widely used in small data set estimation. Another approach is the piecewise method, the primary concern with this method is how to find the relation between data points, the second is how to combine different pieces together to form one surface with certain degree of smoothness. The most widely used approach in finding the nature relation among the data points is to use the Delaunay triangle or Vororoid diagram [14]. The triangle network builds up the connection among the points which makes data searching, inserting become much easier. Minimum Norm Network(MNN) method is used by Nielson[18], and an extended Gauss interpolation method is used by Agishtein [1] to blend the pieces together. Surface subdivision method is used by Dyn et al[6], which is advantageous in surface design. New mathematical methods are also developed to handle the scattered data estimation in recent years, which generalized the Spline method in the rectangle data set to the scattered one, among them are Multivariate Splines and Box Splines [18, 12].

Contouring of surface data and isovolume visualization of volume data are widely used for viewing 2 and 3 dimensional data set. There are two problems in the surface contouring as well as in the isosurface generation with Marching Cube algorithm[15]. The first one is the ambiguity of the contour line or isosurface in certain conditions[26], the second is the smoothness of the final result[23].
In this thesis, we proposed the quadratic method for regular data, extended piecewise Hardy and Bezier triangle method for scattered surface data estimation. A subdivision approach which eliminates the ambiguity problem and can also generate arbitrarily smooth contour lines for both the regular and scattered data surface is also presented.

A graphics package based on the approaches described in the thesis for scattered data estimation and visualization is implemented with a Graphics User Interface (GUI) on Sparc Station running UNIX and X windows and on IBMPC running MSDOS.

In Chapter 2, we review commonly used deterministic and stochastic approaches in regular data estimation, and proposed the quadratic method for surface estimation and compared it with the other two methods. The quadratic approach is faster than the other two methods and may be used in real time application such as robotic motion planning.

In Chapter 3, we introduce the method to triangulate the scattered surface data set and reviewed methods in scattered data estimation, the Hardy, Gauss and Kriging method. Then we discuss the two new methods, namely, the piecewise Hardy method and Bezier triangle method. The extended piecewise Hardy method is an extension of the original Hardy method over the triangle network. The Bezier triangle method is based on the regular Bezier patch estimation in Computer Aided Geometric Design. The two methods generally give satisfactory results in our experiments with both computer generated and practical data.

Chapter 4 is about the contouring of regular and scattered surface data. The new subdivision approach is discussed in detail along with the existing ones. One problem in surface contouring is the ambiguity and the smoothness of the contour line for the rectangular data set. The new approach solves this problem by dividing the rectangle grid into triangles or use the triangles from the Delaunay triangulation to construct the surface contour for scattered or regular data set. By considering triangles only, the algorithm becomes very simple and arbitrarily smooth contour lines can be obtained by increasing the number of subdivisions.
In chapter 5, we give a brief introduction to the Scattered Data Interpolation and Visualization package (SDIV) we implemented with the above methods, the Graphics User Interface (GUI) and the functions to handle the scattered data under X11. The source code for the package is listed in the appendix.
Chapter 2

Rectangular Grad Surface Data Interpolation

2.1 The Problem

Rectangular grid data is the most widely used data format in surface estimation. The problem with the grid surface data interpolation is:

given data points \((x, y, z)\) in rectangular grids on a domain \(D\), to estimate the value at certain point within the domain.

The estimation and interpolation for grid data is well studied and the methods can be divided into two categories: deterministic and stochastic. The frequently used deterministic method is the nature bicubic spline and the widely used stochastic method is the Kriging method. The bicubic spline method assumes the continuity of the second derivative of the estimated curve or surface, \(i.e.,\) it is \(C^2\) continuous. Furthermore, the cubic spline method need to assume certain boundary condition to solve the system of equations. The Kriging method assumes that the data to be estimated is from a stationary process. It also needs to estimate the autocorrelation of the process. In many cases, the original data is not enough for a reliable estimation. The Kriging method is computationally expensive, one needs to solve a system of equations for almost every estimated point. While the two methods give quite good results in some applications, they are not fast enough for others, such as real time applications. In this chapter, we first describe the cubic spline and the Kriging method, and then discussed the quadratic method, which is proposed by Yfantis [27] [28] [29], and its extensions for fast curve and surface estimation. The quadratic method is a local method in that one point only influences a few neighbor points, as opposed to the cubic spline method, which influences all.
2.2 The Existing Method

2.2.1 Deterministic Method: Cubic Splines and Surfaces

The Cubic Splines

Suppose we have \((x_i, y_i, z_i), i = 0, 1, \ldots, n\) points in three dimensional space. We want to interpolate the points with cubic polynomials, i.e., cubic splines. To construct cubic spline, we assume the continuity of the first and the second derivative of the curve to be estimated. The \(i\)th curve segment can be generally express by:

\[
\begin{align*}
X_i(t) &= a_i + b_i t + c_i t^2 + d_i t^3 \\
X_i^{(1)}(t) &= b_i + 2c_i t + 3d_i t^2 \\
X_i^{(2)}(t) &= 2c_i + 6d_i t
\end{align*}
\] (2.1)

as shown in Figure 2.1. The expression is parametric. We assume that the interpolation in the \(x\) coordinates is determined by the \(x\) coordinates only, and so are the \(y\) and \(z\) coordinates. For simplicity, we develop only the equation for the \(x\) coordinate. The derivation for the other two is similar to that of \(x\).

With the boundary conditions, we have:

\[
\begin{align*}
X_i(0) &= x_i &= a_i \\
X_i(1) &= x_{i+1} &= a_i + b_i + c_i + d_i \\
X_i^{(1)}(0) &= r_i &= b_i \\
X_i^{(1)}(1) &= r_{i+1} &= b_i + 2c_i t + 3d_i t^2 
\end{align*}
\] (2.2)
Where \( r_i \) is the first derivative at point \( x_i \). Solving the four equations, we get:

\[
\begin{align*}
    a_i &= x_i \\
    b_i &= r_i \\
    c_i &= 3(x_{i+1} - x_i) - 2r_i - r_{i+1} \\
    d_i &= 2(x_i - x_{i+1}) + r_i + r_{i+1}
\end{align*}
\]

Now we need to solve the first derivative from the original data, i.e. from the \( x_0, x_1, \ldots, x_n \), where we have \( n + 1 \) data points.

By the continuity of the first and second derivatives, we have the equation:

\[
\begin{align*}
    X_{i-1}(1) &= x_i \\
    X_{i-1}^{(1)}(1) &= X_{i}^{(1)}(0) \\
    X_{i}(0) &= x_i \\
    X_{i-1}^{(2)}(1) &= X_{i}^{(2)}(0)
\end{align*}
\]

and also the conditions at the two ends:

\[
X_0(0) = x_0, \quad X_{m-1}(1) = x_m
\]

Now we have \( 4(m - 1) + 2 = 4m - 2 \) for \( 4m \) variables. We still need two more equations. These two equations can be obtained by specifying the two end conditions. One of such choices is to assume the second derivatives at the two ends be zero, which is the so called nature condition.

From the second derivative continuity, \( X_{i-1}^{(2)}(1) = X_i^{(2)}(0) \), we have:

\[
2c_{i-1} + 6d_{i-1} = 2c_i
\]

Substituting into 2.3, we get:

\[
r_{i-1} + 4r_i + r_{i+1} = 3(y_{i+1} - y_{i-1})
\]

with the two end conditions, we can also get:

\[
2r_0 + r_1 = 3(x_1 - x_0)
\]

\[
r_{m-1} + 2r_m = 3(x_m - x_{m-1})
\]
We now get the system of equations to solve the first derivatives:

\[
\begin{bmatrix}
2 & 1 \\
1 & 4 & 1 \\
1 & 4 & 1 \\
1 & 4 & 1 \\
\vdots \\
1 & 4 & 1 \\
1 & 2
\end{bmatrix}
\begin{bmatrix}
r_0 \\
r_1 \\
r_2 \\
r_3 \\
\vdots \\
r_{m-1} \\
r_m
\end{bmatrix}
= 
\begin{bmatrix}
3(x_1 - x_0) \\
3(x_2 - x_0) \\
3(x_3 - x_1) \\
3(x_4 - x_2) \\
\vdots \\
3(x_m - x_{m-2}) \\
3(x_m - x_{m-1})
\end{bmatrix}
\] (2.4)

By solving the equations for \( r_i \), we can use equation 2.3 to get the necessary parameters to estimate the points between points \( x_i \) and \( x_{i+1} \).

The system of equation can be easily solved by the following steps [3]:

step 1: eliminate the first 1 in each row.

\[
\gamma_0 = 1/2 \\
\delta_0 = 3(x_1 - x_0)\gamma_0 \\
\text{for } i = 1 \text{ to } m - 1 \text{ do} \\
\gamma_i = 1/(4 - \gamma_{i-1}) \\
\delta_i = (3(x_m - x_i) - \delta_{i-1})\gamma_i \\
\text{end for} \\
\gamma_m = 1/(2 - \gamma_{m-1}) \\
\delta_m = (3(x_m - x_{m-1}) - \delta_{m-1})\gamma_m
\]

Now the system of equations becomes:

\[
\begin{bmatrix}
1 & \gamma_0 \\
1 & \gamma_1 \\
1 & \gamma_2 \\
\vdots \\
1 & \gamma_{m-1} \\
1 & 1
\end{bmatrix}
\begin{bmatrix}
r_0 \\
r_1 \\
r_2 \\
\vdots \\
r_{m-1} \\
r_m
\end{bmatrix}
= 
\begin{bmatrix}
\delta_0 \\
\delta_1 \\
\delta_2 \\
\vdots \\
\delta_{m-1} \\
\delta_m
\end{bmatrix}
\] (2.5)

Step 2: solve the equation by backward substitution:

\[
r_m = \delta_m \\
\text{for } i = m - 1 \text{ to } 0 \text{ do}
\]
Cubic Surface

The surface data can be estimated by the cubic surface in a similar way. We also suppose that the surface under consideration is $C^2$ continuous. The surface in $X_{i,j}$ patch as shown in Figure 2.2 can be expressed as:

$$X_{i,j}(u, v) = a_{00} + a_{01}v + a_{02}v^2 + a_{03}v^3 +$$
$$a_{10}u + a_{11}uv + a_{12}uv^2 + a_{13}uv^3 +$$
$$a_{20}u^2 + a_{21}u^2v + a_{22}u^2v^2 + a_{23}u^2v^3 +$$
$$a_{30}u^3 + a_{31}u^3v + a_{32}u^3v^2 + a_{33}u^3v^3$$

The first partial derivatives and the mixed partial derivatives along $u$ and $v$ can be expressed:

$$\frac{\partial X_{i,j}(u, v)}{\partial u} = a_{10} + a_{11}v + a_{12}v^2 + a_{13}v^3 +$$
$$2u(a_{20} + a_{21}v + a_{22}v^2 + a_{23}v^3) +$$
$$3u^2(a_{30} + a_{31}v + a_{32}v^2 + a_{33}v^3)$$
\[ \frac{\partial X_{ij}(u,v)}{\partial v} = a_{01} + 2a_{02}v + 3a_{03}v^2 + \\
\quad u(a_{11} + 2a_{12}v + 3a_{13}v^2) + \\
\quad v^2(a_{21} + 2a_{22}v + 3a_{23}v^2) + \\
\quad v^3(a_{31} + 2a_{32}v + 3a_{33}v^2) \tag{2.7} \]

\[ \frac{\partial^2 X_{ij}(u,v)}{\partial u \partial v} = a_{11} + 2a_{12}v + 3a_{13}v^2 + \\
\quad 2u(a_{21} + 2a_{22}v + 3a_{23}v^2) + \\
\quad 3u^2(a_{31} + 2a_{32}v + 3a_{33}v^2) \]

We have 16 unknowns in equation 2.7, our goal is to find the unknowns from the data points we have. Suppose we know the first and the mixed derivatives as well as the original data for the time being.

From \( C^0 \) continuity, we have:

\[
\begin{align*}
X_{i,j}(0,0) &= x_{ij} = a_{00} \\
X_{i,j}(0,1) &= x_{i,j+1} = a_{00} + a_{01} + a_{02} + a_{03} \\
X_{i,j}(1,0) &= x_{i+1,j} = a_{00} + a_{10} + a_{20} + a_{30} \\
X_{i,j}(1,1) &= x_{i+1,j+1} = a_{00} + a_{01} + a_{02} + a_{03} + \\
& \quad a_{10} + a_{11} + a_{12} + a_{13} + \\
& \quad a_{20} + a_{21} + a_{22} + a_{23} + \\
& \quad a_{30} + a_{31} + a_{32} + a_{33}
\end{align*}
\]

From the \( C^1 \) continuity along the \( u \) and \( v \) direction and denote \( \frac{\partial X_{i,j}}{\partial u} = p_{i,j} \), and \( \frac{\partial X_{i,j}}{\partial v} = q_{i,j} \), we have:

\[
\begin{align*}
p_{i,j} &= a_{10} \\
p_{i,j+1} &= a_{10} + a_{11} + a_{12} + a_{13} \\
p_{i+1,j} &= a_{10} + 2a_{20} + 3a_{30} \\
p_{i+1,j+1} &= a_{10} + a_{11} + a_{12} + a_{13} + \\
& \quad 2(a_{20} + a_{21} + a_{22} + a_{23}) + \\
& \quad 3(a_{30} + a_{31} + a_{32} + a_{33})
\end{align*}
\]

and

\[
\begin{align*}
q_{i,j} &= a_{01} \\
q_{i,j+1} &= a_{00} + 2a_{02} + 3a_{03} \\
q_{i+1,j} &= a_{01} + a_{11} + a_{21} + a_{31} \\
q_{i+1,j+1} &= a_{01} + a_{11} + a_{21} + a_{31} + \\
& \quad 2(a_{02} + a_{12} + a_{22} + a_{32}) + \\
& \quad 3(a_{03} + a_{13} + a_{23} + a_{33})
\end{align*}
\]
From the continuity of the mixed partial derivative, and denote \( \frac{\partial^2 X_{ij}}{\partial u \partial v} = r_{ij} \), we get another set of equation:

\[
\begin{align*}
\tau_{i,j} &= a_{11} \\
\tau_{i,j+1} &= a_{11} + 2a_{12} + 3a_{13} \\
\tau_{i+1,j} &= a_{11} + 2a_{21} + 3a_{31} \\
\tau_{i+1,j+1} &= a_{11} + 2a_{12} + 3a_{13} \\
&\quad + 2a_{21} + 4a_{22} + 6a_{23} \\
&\quad + 3a_{31} + 6a_{32} + 9a_{33}
\end{align*}
\]

With the above 16 equations, we can solve the 16 unknowns and hence can estimate the points in the patch \( X_{ij} \). The solution can be expressed in the matrix form as:

\[
X_{ij}(u, v) = HBH'
\]

where

\[
H = \begin{bmatrix} (2t + 1)(1 - t^2) & t(1 - t^2) & t^2(t - 1) & t^2(3 - 2t) \end{bmatrix}
\]

\[
B = \begin{bmatrix} x_{0,0} & q_{0,0} & q_{0,1} & x_{0,1} \\ p_{0,0} & r_{0,0} & r_{0,1} & p_{0,1} \\ p_{1,0} & r_{1,0} & r_{1,1} & p_{1,1} \\ x_{1,0} & q_{1,0} & q_{1,1} & x_{1,1} \end{bmatrix}
\]

Our problem now is to get the partial and the mixed partial derivatives. The partial derivatives along \( u \) and \( v \) can be obtained by the method as discussed in the cubic splines. The mixed partial can be obtained from the already known partials, say \( p_{i,j} \), along one direction to estimate the partial derivative along the other direction by the method described above.

2.2.2 Stochastic Method: Kriging

Kriging is a widely used stochastic method for surface estimation. Especially in ore reserve estimation, meteorology and environmental problems. Kriging method assumes that the surface under consideration is a stationary process with the fixed mean \( \mu \), variance \( \sigma^2 \) and the autocorrelation is the function of the space leg only.
Suppose we have the sample points at \( x_i (i = 1, 2, \ldots, n) \) with their \( Z \) value \( Z(x_i) \), we want to find a set of weights \( a_i (i = 1, 2, \ldots, n) \) to make the weighted average

\[
Z^*(x_0) = \sum_{i=1}^{n} a_i Z(x_i)
\]  

(2.9)

be the best estimation of the true value of \( Z \) at point \( x_0 \) in the mean square error sense, as shown in Figure 2.3.

By definition of stationary process, we have \( E(Z(x_i)) = \mu \) and \( Var(Z(x_i)) = \sigma^2 \) for all points.

Imposing the unbiased condition on \( Z^*(x_0) \), we get:

\[
\mu = E(Z^*(x_0)) = \sum_{i=1}^{n} a_i E(Z(x_i))
\]

or

\[
\sum_{i=1}^{n} a_i = 1
\]

Our goal is to minimize the square error of the estimation, and the square error can be expressed as [4]:

\[
E(Z^*(x_0) - Z(x_0))^2 = \sum_{i} \sum_{j} a_i a_j R(x_i - x_j) - 2 \sum_{i} a_i R(x_i - x_0) + \sigma^2
\]

(2.10)

where \( R(x_i - x_j) \) is the autocorrelation function at lag \( x_i - x_j \). For statistically stationary process, the autocorrelation function depends only on the distance between
two points, not the absolute position of the points. Now, the square error is a function of \( a_i \). To minimize the error, we use the Lagrange multiplier with the constrain of \( \sum a_i = 1 \):

\[
f(a_1, a_2, \ldots, a_n) = \sum_i \sum_j a_i a_j R(x_i - x_j) - 2 \sum_i a_i R(x_i - x_0) + \sigma^2 + 2\lambda(\sum a_i - 1)
\]

and the derivatives

\[
\frac{\partial f}{\partial a_k} = 2 \sum_i a_i R(x_k - x_i) - 2 R(x_k - x_0) + 2\lambda = 0
\]

\[
\frac{\partial f}{\partial \lambda} = \sum_i a_i - 1 = 0 \quad k = 1, 2, \ldots n
\]

So we get a linear system of equations with \( n+1 \) unknowns.

\[
\begin{cases}
\sum_i a_i R(x_k - x_i) + \lambda = R(x_k - x_0) & i = 1, 2, \ldots n \\
\sum_i a_i = 1
\end{cases}
\]

(2.11)

Solving equation 2.11 for all the \( a_i \)-s and substitute back to equation 2.9, we can get the best linear unbiased estimation at point \( x_0 \). With the result, we can also get certain knowledge about the error of our estimation by equation 2.10.

2.3 The Quadratic Method and Its Extensions

2.3.1 The Quadratic Method

The quadratic method for curve and surface estimation was proposed by Yfantis[27]. The following two sections is a brief description with some extensions.

The Quadratic Curve

Let \((x_0, y_0, z_0), (x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\) be \( n+1 \) points in the three dimensional space, our goal is for finding a curve passing through these points. The strategy to find the curve passing through points \((x_i, y_i, z_i)\) and \((x_{i+1}, y_{i+1}, z_{i+1})\) is the following:

1. Find the quadratic parametric curve passing through \((x_i, y_i, z_i), (x_{i+1}, y_{i+1}, z_{i+1})\) and \((x_{i+2}, y_{i+2}, z_{i+2})\).

Suppose that the curve has the form

\[
X_i(t) = a_0 + a_1 t + a_2 t^2 \quad 0 \leq t \leq 2, 1 \leq i \leq n - 2
\]

(2.12)
with the condition: \( X_i(0) = x_i, X_i(1) = x_{i+1} \) and \( X_i(2) = x_{i+2} \)

By solving the equation, we get:

\[
X_i(t) = 0.5(x_{i+2} - 2x_{i+1} + x_i)t^2 + 0.5(4x_{i+1} - 3x_i - x_{i+2})t + x_i \\
0 \leq i \leq n - 2, \quad 0 \leq t \leq 2
\]

or

\[
X_i(t) = 0.5(2 - 3t + t^2)x_i + (2t - t^2)x_{i+1} + 0.5(t^2 - t)x_{i+2} \tag{2.13}
\]

2. Similarly find the quadratic parametric curve passing through points \((x_{i-1}, y_{i-1}, z_{i-1}), (x_i, y_i, z_i)\) and \((x_{i+1}, y_{i+1}, z_{i+1})\) with the condition \(X_i(-1) = x_{i-1}, X_i(0) = x_i\) and \(X_i(1) = x_1\).

We have

\[
X_{i-1} = 0.5(t^2 - t)x_{i-1} + (1 - t^2)x_i + 0.5(t^2 + t)x_{i+1} \tag{2.14}
\]

Combining equation 2.13 and 2.14 and weighted with \(w_i\) and \((1 - w_i)\), we get:

\[
X_i = w_i[0.5(2 - 3t + t^2)x_i + (2t - t^2)x_{i+1} + 0.5(t^2 - t)x_{i+2}] + \\
(1 - w_i)[(1 - t^2)x_i + 0.5(t^2 + t)x_{i+1} + 0.5(t^2 - t)x_{i-1}]
\]

For the first and the last patch:

\[
X_0 = 0.5(2 - 3t + t^2)x_0 + (2t - t^2)x_1 + 0.5(t^2 - t)x_2 \\
X_{n-1} = 0.5(t^2 - t)x_{n-2} + (1 - t^2)x_{n-1} + 0.5(t^2 + t)x_n
\]

**The Quadratic Surface**

The surface estimation is based on the curve estimation and the tensor production method. Intuitively, we can first estimate all the points row by row with the quadratic splines and then estimate the points column by column including the original and the estimated points from the row estimation.

There are 4 cases in horizontal and vertical relations among data points as shown in Figure 2.4. Case 0 is for interior segments, case 1 is for the left end segments,
case 2 is for the right end segments. Case 4 may not happen in most of occasions, in which case there are only two points in a row or column and we may use linear interpolation.

We define the matrix for surface estimation:

\[
B = \begin{bmatrix}
  x_{i-1,j-1} & x_{i-1,j} & x_{i-1,j+1} & x_{i-1,j+2} \\
  x_{i,j-1} & x_{i,j} & x_{i,j+1} & x_{i,j+2} \\
  x_{i+1,j-1} & x_{i+1,j} & x_{i+1,j+1} & x_{i+1,j+2} \\
  x_{i+2,j-1} & x_{i+2,j} & x_{i+2,j+1} & x_{i+2,j+2}
\end{bmatrix}
\]

and array of vectors

\[
T[0](u) = (0.25(t^2 - t), 0.25(4 - 3t - t^2), 0.5(3t - t^2), 0.5(t^2 - t))
\]

\[
T[1](u) = 0.5(0, 2 - 3u + u^2, 4u - 2u^2, u^2 - u)
\]

\[
T[2](u) = 0.5(u^2 - u, 1 - u^2, u^2 + u, 0)
\]

\[
T[3](u) = (0, 1 - u, u, 0)
\]

where \(T[0] \ldots T[3]\) are the blending functions for the four horizontal or vertical cases in the interpolation. For patch \(X_{i,j}\) with the horizontal case \(l\) and vertical case \(m\) we have

\[
X_{i,j}(v, u) = T[m](v)BT'[l](u)
\]

2.3.2 Extensions

Although simple, the quadratic method described in the last section does not ensure the \(C^1\) or \(C^2\) or higher degree of continuity. The desired continuity can be achieved by a proper weight.

Suppose we have two functions \(f_1(t)\) and \(f_2(t)\), they satisfy the property:

\[
\begin{align*}
  f_1^{(i)}(0) &= f_2^{(i)}(0) \\
  f_1^{(i)}(1) &= f_2^{(i)}(1), \quad i = 0, 1, \ldots k
\end{align*}
\]

i.e., the two functions are the same in \(0 \leq t \leq 1\) up to the \(k\)th derivatives. We take the form

\[
f(t) = w(t)f_1(t) + (1 - w(t))f_2(t)
\]
to combine the two functions to form a new function \( f(t) \) with the property that it satisfies the \( C^n \) continuity with the condition:

\[
\begin{align*}
  f^{(i)}(0) &= f_1^{(i)}(0) \\
  f^{(i)}(1) &= f_2^{(i)}(1) & i = 0, 1, \ldots n
\end{align*}
\]

Because

\[
\begin{align*}
  f^{(n)}(t) &= \sum_{i=0}^{n} C^n_i((1 - w(t))^{(n-i)} f_1^{(i)}(t) + w^{(n-i)}(t)f_2^{(i)}(t)) \\
  &= (1 - w(t))f_1^{(n)}(t) + w(t)f_2^{(n)}(t) + \\
  &\quad \sum_{i=0}^{n-1} C^n_i w^{(n-i)}(t)(f_2^{(i)}(t) - f_1^{(i)}(t)) \\
  &= (1 - w(t))f_1^{(n)}(t) + w(t)f_2^{(n)}(t) + \\
  &\quad \sum_{i=k+1}^{n-1} C^n_i w^{(n-i)}(t)(f_2^{(i)}(t) - f_1^{(i)}(t))
\end{align*}
\]

we get:

\[
\begin{align*}
  f^{(n)}(0) &= (1 - w(0))f_1^{(n)}(0) + w(0)f_2^{(n)}(0) + \\
  &\quad \sum_{i=k+1}^{n-1} C^n_i w^{(n-i)}(0)(f_2^{(i)}(0) - f_1^{(i)}(0))
\end{align*}
\]
If we set
\[ w(0) = 0, \quad w(1) = 1, \]
\[ w^{(i)}(0) = w^{(i)}(1) = 0, \quad i = 1, \ldots, n - k - 1, \]
then, the function \( f(t) \) satisfies the conditions. Further more, if we suppose \( w(t) \) is a polynomial, we can solve a differential equation with the above condition to get the \( w(t) \).

For \( n = k, k + 1, k + 2 \), we have:

\[
 w(t) = \begin{cases} 
  \text{constant} & n = k \\
  1 - t & n = k + 1 \\
  1 - 3t^2 + 2t^3 & n = k + 2 
\end{cases}
\]

In the quadratic splines, the two functions to be blended are quadratic and \( C^0 \) continuous over the interval \( 0 \leq t \leq 1 \), i.e., \( k=0 \). We can achieve \( C^1 \) and \( C^2 \) continuity by selecting \( 1 - t \) or \( 1 - 3t^2 + 2t^3 \) as the weight in the blending functions.

**2.4 Comparisons**

**2.4.1 Practical Considerations**

The cubic splines are widely used and usually give satisfactory result. To estimate the data with cubic splines, we first need to solve the system of equations to get the first derivative of the curve, with time complexity of \( O(n) \), where \( n \) is the number of data points.

For bicubic surface, we need to solve two partial derivatives and the mixed partial derivatives with the method similar to the one of cubic splines. For the data set of \( n \times n \), the we need solve the system of equations \( 3n \) times. The time complexity is the order of \( O(n^2) \).
The Kriging method is the statistically best method in theory provided that the data satisfies the assumptions. To estimate with Kriging method, one need to solve a system of equations. The size of the unknowns depends on the original points in the zone of influence, i.e., the autocorrelation function. In most practical situations, the autocorrelation function in a stationary process will becomes zero at certain distance as shown in Figure 2.5. The points beyond the distance from the estimated point will not be included in the estimation. Usually the Kriging method take more time for the same estimation than other methods because the time complexity to solve a equation of $n$ variables. But for data in rectangular grid, some technique may be used to make the estimation faster by noticing that the matrix in the system of equation is neither related to the absolute position nor the surface value. One problem with Kriging is that it assumes that the surface to be estimated is an ensemble of a stationary process with the fixed mean and variance. This assumption may not be satisfied for the actual data or it is difficult to verify that the data satisfies. Another problem is how to estimate the autocorrelation function or semi-varigram used in the equations giving only the original data. But, The problems can be solved in some practical applications. The method is widely used in ore, meteorology and environmental problems with much success and some empirical autocorrelation functions have been established with only a few parameters needed to be adjusted by the real data.

In the quadratic estimation, one does not need to solve system of equations, which makes the approach faster than the other two, nor does it ensure the smoothness of the estimation. To raise the quadratic estimation to cubic or higher with some blending function one can solve this problem and make the estimation to be $C^1$ or $C^2$, while still maintaining the simplicity of the approach.

2.4.2 Experimental Results

In this section, we give some experimental result on surface estimation with the cubic and quadratic splines. we use the known function to test the methods with a sampling interval $\delta = 5$ and range from $-20$ to 20 for the $x$ and $y$ directions. We give in the figure the original, estimated and error results for different methods.
The functions are defined as following:

\[
F_1(x, y) = 0.5 + 0.00001526(x^3 + y^3 + x^2y + y^2x)
\]

\[
F_2(x, y) = \exp(-\sin^2(0.2\sqrt{x^2 + y^2}))
\]

The errors and the time for different method are shown in table 2.1 and 2.2.

From the tables, we have the conclusion that the quadratic or its extension is approximately three times faster than the cubic spline method, but the cubic spline has less error for complex functions.
Figure 2.6: The original function. Left: function 1. Right: function 2.

Figure 2.7: The function 1 estimation using the Cubic Splines. Left: the estimated function. Right: error surface. Note the scale in the error surface is from 0.1 to 0.1

Figure 2.8: The function 1 estimation using the Quadratic method. Left: the estimated function. Right: error surface. Note the scale in the error surface is from 0.1 to 0.1
Figure 2.9: The function 1 estimation using the extended $C^1$ Quadratic method. Left: the estimated function. Right: error surface. Note the scale in the error surface is from 0.1 to 0.1

Figure 2.10: The function 2 estimation using the Cubic Splines. Left: the estimated function. Right: error surface. Note the scale in the error surface is from 0.1 to 0.1

Figure 2.11: The function 2 estimation using the Quadratic method. Left: the estimated function. Right: error surface. Note the scale in the error surface is from 0.1 to 0.1
Figure 2.12: The function 2 estimation using the extended $C^1$ Quadratic method. Left: the estimated function. Right: error surface. Note the scale in the error surface is from 0.1 to 0.1
Chapter 3

Scattered Surface Data Interpolation

3.1 The problem

Surface interpolation or estimation from scattered data is an important problem in computer aided geometric design, scientific data visualization, computer graphics and image processing. The scattered data estimation has been extensively studied in the past decades. The problem may be stated as following:

Given data set \((x_i, y_i, z_i), \ i = 0, 1, \ldots, n - 1\) of size \(n\), reconstruct a function of \(F(x, y)\) such that \(F(x_i, y_i) = z_i\) for all \(i\), provided that all \((x_i, y_i)\) pairs are distinct and there is no assumption about the position of \((x_i, y_i)\) on the data set.

Generally, there are two approaches for scattered data interpolation; the global and local approaches, global and local. Among all the local methods dealing with the scattered data surface estimation, most of them are based on Delaunay triangulation [1]. Delaunay triangulation of the scattered data is a fundamental way to deal with the irregular data set. With the triangulation method, we find the nature relation between the seemingly unrelated data set. Hardy’s method, on the other hand, is a global approach to solve the surface estimation from the scattered data set, it can be easily generalized to the three dimensional data estimation and generally gives rather satisfactory results [17]. However, one needs to solve a large system of equations for large data set which may become very difficult and time consuming, because the time complexity to solve a system of equation of \(n\) variables is of the order \(O(n^3)\). As \(n\) becomes large, the equation may become ill conditioned [17]. The recently developed Multivariate and the Box splines methods are nontensor extension of the B-Spline method and preserve the properties of the splines in the regular grids [2].

Triangulation is the first step of surface estimation for most of the piecewise surface
estimation methods. After that, we still need to find ways to interpolate the data in the triangles. The problem becomes not so straightforward because there may be indefinite numbers of points connecting to any triangle vertex.

In this chapter, we first give the algorithm for Delaunay triangulation, then review some of the methods dealing with the scattered data interpolation, and introduce two new methods for the scattered data interpolation. In one of our method, we extended the Hardy's method to the piecewise situation, and use only the triangle enclosing the data point and its neighboring triangles to do the interpolation. The point in a triangle can be obtained by first getting three values from the three vertices and then sum them up with certain weight in accordance with normals of the vertices and the barycentric position of the data point in the triangle. The interpolated surface is $C^1$ and is determined by the weight function used. We also extend the Bezier method from regular triangle surface estimation to the irregular case. In this approach, we use the triangle and its neighbors to find the control points in the triangle, and then estimate the points in it. This scheme uses the parametric estimation, and thus preserves much of the desirable features of the original Bezier method. It can also be extended to nonfunctional surface estimation and surface design once we get the triangulation of the original data.

3.2 Triangulation of Scattered Data

3.2.1 Delaunay Triangulation

Given a set of data points of the form $(x_i, y_i, z_i)$, our goal is to estimate a function $F(x, y)$ such that $F(x_i, y_i) = z_i$ for all the given data as stated above. The problem becomes not so easy in that we can not find relations between data points at first. Before we can estimate the surface, we have to know the relation between point $(x_i, y_i)$ and his neighbors as we do in the rectangular data set estimation. It may be hard to find a rectangle for every data point in the scattered data set and use the method of tensor product. But it is possible to define a triangular relation for the data points of the $x - y$ plane. Obviously, we have many ways to define the triangular relation for
the data set. One of the widely used method to solve the problem is the Delaunay triangulation. The method is optimal in the sense that it makes the triangle in the triangle net as equilateral as possible just as we would like to draw when we build the triangle net by hand.

Triangulation for scattered surface estimation can be regarded as a planar graph in which all the edges intersect only at the vertices. Obviously, the triangulation is not unique, and for a convex hull, there are $3(n_i - 1) - n_b$ edges and $2(n_i - 1) - n_b$ triangles for triangulation of data set of size $n$ with $n_b$ boundary points and $n_i$ interior points [14].

There are three criteria to get the optimal triangulation, i.e., Max-Min angle criterion, Circle criterion and the Thiessen region criterion, and were proved to be equivalent [14]. For the convex quadrilateral formed by two triangles, the Max-Min angle means to change the diagonal of the quadrilateral to make the minimum angle in the two triangles maximum. Thiessen region criterion is to subdivide the plane into polygon each of which associates with a point in the data set. The points inside the polygon are the points closer to this polygon than to any other ones. In this paper, we use the circle criterion. It may be stated as:

Let $K$ denote a circle passing through three of the four vertices of a strictly convex quadrilateral $Q$. if the fourth vertex is within the circle $K$, insert the diagonal from this fourth vertex to the opposite vertex. If the fourth vertex is exterior to $K$, insert the diagonal of the other two vertices. If the fourth vertex is on the circle, we can insert in either way, see Figure 3.1. By making all the triangles local optimal with this criterion, we get a triangle net which is global optimal [14].

The algorithms developed for Delaunay triangulation of scattered data set can be divided into roughly two classes, i.e., incremental algorithms and divide-and-conquer algorithms. The incremental algorithms [5] construct the Delaunay triangulation by adding points to the already existing triangle net, treats the newly inserted point as a perturbation to the original triangle net, then adjusts the triangle net and makes it optimal again in a number of steps. The divide-and-conquer algorithms recursively
a. The fourth point is in the circle  

b. The fourth point is exterior to the circle

Figure 3.1: Circle and Thiesson criterion for triangulation

Figure 3.2: Delaunay triangulation

divided the region into two subregions until the final triangulation is obtained. Incremental algorithms have the time complexity of $O(n^2)$ in the worst case, but they are easier to code and require less memory space than the divide-and-conquer method. In [10], a novel approach is used to make the incremental algorithm with the time complexity of $(n \log n)$ on the average.

### 3.2.2 The Delaunay Triangulation

We adopt the incremental method in the following. The method that follows is based on [5] with some modifications. In our approach, the overall time complexity for triangulation is of the order $O(n^{1.5})$ for the worst case and $O(n)$ for the best case, which is better than the standard incremental methods.

Suppose all the data are in the array `DataPoint`, which is a structure of the form

```c
struct { float x; float y; float z; } DataPoint[];
```
We define structure **TRIANGLE** to keep track of the triangles of the triangle net and use the variable **Triangle** as the header of the data structure.

```c
struct {P[3]; struct TRIANGLE *E[3]; int visit;} TRIANGLE;
```

Here, we use `P[3]` as the index to the DataPoint to keep track of the three vertices of the triangle, `E[3]` as three pointers to keep track of the three neighbors of the triangle, it is **NULL** if none. `visit` is used by the traversal of the triangle and is initially set to 0. We use a circular linked list to keep track of the boundary of the triangle net. Each node in the link list has a pointer to the triangle that shares the same edge, and a pointer to the next node. We adopt the convention that we label all the data in counter clockwise order, for instance, `P[0],P[1],P[2]` are the three vertices of a triangle in the counter clockwise order, and so is the `E[3]`, the order of the three neighbors.

There are special cases for the triangulation. The most severe problem is the boundary condition. A newly inserted point may fall beyond the existing triangle network or lie just on an edge of the boundary. These conditions can be circumvent by first building a triangle large enough to enclose all the scattered data points. This way, we may consider only a few special cases and the boundary of the triangle network keeps the same during the point insertion.

The following are pseudo code for triangulation and **InitialStack**, **EmptyStack**, **Push** and **Pop** are operations to manipulate the triangles in the stack.

**Procedure Triangulation**

step 1:

Build the first triangle large enough to enclose all the data points.

Set **Triangle** as the first one.

**InitialStack**

step 2:

```c
for point = 1 to n
    InsertPoint(i)
end for
```
End Triangulation

The procedure InsertPoint is used to insert a new point to the existing triangle network. The code is shown in the following.

Procedure InsertPoint(n)
{n is the point index in the data set }

i=InTriangle(&t,n);
{Intriangle tests if point n is in the triangle net, if yes it sets the triangle t which encloses the point and return 1, if the point is on an edge of a triangle it sets the triangle t and return 0. }

if(i == 0) { the point is on one edge of triangle t }

SplitTriangle0(t, &t0, &t1, &t2, &t3, n);
{Split the triangle t into four subtriangles, or two if t has no neighbor on this edge.}

Edge = 1; Triangle = t0;

if(t0->E[Edge]! = NULL)
{Push all the exchangeable edges into stack if this edge has neighbor }

Push(Edge,t0, t0->E[Edge]);

if(t1->E[Edge]! = NULL)
Push(Edge,t1, t1->E[Edge]);

if(t2->E[Edge]! = NULL)
Push(Edge,t2, t2->E[Edge]);

if(t3->E[Edge]! = NULL)
Push(Edge,t3, t3->E[Edge]);

end if(i==0)

if(i == 1) { the point is in the triangle }

SplitTriangle1(t, &t0, &t1, &t2, n);
{Split the triangle \( t \) into 3 subtriangles with point \( n \) as the first point in the new triangles.}

\[ \text{Edge} = 1; \text{Triangle} = t0; \]

if\( (t0 - > E[\text{Edge}]! = NULL) \)

{Push all the exchangeable edges into stack if this edge has neighbor}

\[ \text{Push}(\text{Edge}, t0, t0 - > E[\text{Edge}]); \]

if\( (t1 - > E[\text{Edge}]! = NULL) \)

\[ \text{Push}(\text{Edge}, t1, t1 - > E[\text{Edge}]); \]

if\( (t2 - > E[\text{Edge}]! = NULL) \)

\[ \text{Push}(\text{Edge}, t2, t2 - > E[\text{Edge}]); \]

end if (i == 1)

while\( (\text{not EmptyStack}()) \)

{ if the test edges are not over }

\[ \text{Pop}(\&\text{Edge}, \&t1, \&t2); \]

if\( (\text{Swap}(\text{Edge}, t1, t2)) \) {if swap the edge}

\[ \text{Triangle} = t1; \]

if\( (t1 - > E(0)! = NULL) \)

{push all the possible exchangeable edge into stack}

\[ \text{Push}(0, t1, t1 - > E[0]); \]

if\( (t1 - > E(1)! = NULL) \)

\[ \text{Push}(2, t1, t1 - > E[2]); \]

if\( (t2 - > E(0)! = NULL) \)

\[ \text{Push}(0, t2, t2 - > E[0]); \]

if\( (t1 - > E(2)! = NULL) \)

\[ \text{Push}(2, t2, t2 - > E[2]); \]

end while

End InsertPoint

In typical situation, an inserting point is enclosed by a triangle of the exist triangle
net, but special attention must be paid when the inserting data point is on a triangle edge. In this case, we need to split the two triangles into four and push all the four outside edges into stack for possible edge exchange.

The following is the code for the case that the data point is on an edge of the triangle.

**Procedure SplitTriangle0**($t, t0, t1, t2, t3, n$)

{Point $n$ is now on one edge of triangle $t$, the triangle $t$

and its neighbor may be split into four triangles $t0, t1, t2, t3$

if $t$ has neighbor sharing the edge, otherwise, $t$ will be

split into two triangles $t0, t1$.}

$i = \text{FindOnEdge}(t, n);$  

{Find on which edge the point $n$ is on the triangle $t$.}

$\text{AdjTriangle} = t \rightarrow E[i]$;  

{get adjacent triangle sharing the edge}

if$(\text{AdjTriangle} = NULL)$

{Split the two triangle into four subtriangles with point $n$

as the first point in each triangle.}

else  

{$t$ has no neighbor on the side.}

{split $t$ into two subtriangles with point $n$ as the first

point in the triangles}

end if

End SplitTriangle0
3.2.3 Inserting Point To The Triangle Graph

One of the important operations in our algorithm is the procedure InTriangle(), to find if a point is in the triangle net. It may take $O(n)$ steps to find the result, where $n$ is the number of points in the triangle net. This makes the overall complexity of the algorithm $O(n^2)$. In our algorithm, we use a heuristic method to find the triangle enclosing the point. We start with the triangle $Triangle$, if the point to be inserted is not in this triangle, the point must be in the left side of an edge when we go around the triangle with the counter clockwise order. We select the next triangle to be tested as the neighbor of this edge, as shown in Figure 3.4. By this way, we can find a point in about $\sqrt{n}$ steps. Because the data to be inserted are usually related to each other, the next data point will be the neighbor of the last one in most of the situations. It is especially true when the triangle network is used for grid point estimation. By changing $Triangle$ every time to point to the newly found triangle, we can find the place for the new point in a few steps in most of the cases, which makes the complexity of the algorithm approximately the order of $O(n)$ if the data are closely related to each other, or $O(n^{1.5})$, if the data has no relation, which is faster than $O(n^2)$ while still preserves the simplicity of the code.
3.3 The Existing Interpolation Methods

In this section, we will give an introduction to some of the existing algorithms in scattered data interpolation.

3.3.1 Gauss Interpolation

In [1] a slight modified Gauss interpolation method is used in the scattered data interpolation. In order to estimate a data point in a given triangle, the triangle is divided into three subareas by the estimated data point as shown in Figure 3.5. The data point is estimated by the following formula:

\[ f(S_1, S_2) = \frac{\sum_{k=1}^{3} w_k f_k(S_1, S_2)}{\sum_{k=1}^{3} w_k(S_1, S_2)} \]

where \( f_k(.) \) is a function passing exactly through the corresponding vertex of the triangle. \( w_k \) is the weight of the vertex of the triangle. For \( w_3 \) we have

\[ w_3 = \frac{(S_1^2 + S_2^2) * S_3^2}{(\zeta_3 - x)^2 + (\eta_3 - y)^2} \]

where \((\zeta_3, \eta_3)\) is the coordinate of a vertex. We can get other weights by changing the indices. By selecting weights this way, we can get a \( C^1 \) estimation of the surface, provided the functions \( f_k \) passes through the three corresponding triangle vertex[1].

To find the function \( f_k \) passing through the three vertices of the triangle, a simple choice is to find a plane passing through the corresponding vertex with the surface normal as the average of the plane normals of all the triangles sharing the same vertex, i.e. the best fit plane on a vertex. It is easy to see that this method is a local interpolation, which uses only the triangle around the data point and triangles adjacent to it. The estimation is dependent on the method of selecting the function \( f_k \).
as well as the method of selecting the weight $w_k$, a better selection of the $f_k$ function can get a better estimation.

### 3.3.2 Hardy Interpolation

Hardy's method for irregular data estimation uses a global approach. Instead of using only local data to estimate a given point, this method uses all the data points to estimate a value $z$ in position $(x, y)$.

$$ z = \sum_{i=1}^{n} c_i q(x_i, y_i; x, y) \quad (3.1) $$

where $c_i$ is a coefficient and $q(x_i, y_i; x, y)$ is a function of the known point and the point to be estimated, it takes the form

$$ q(x_i, y_i; x, y) = ((x_i - x)^2 + (y_i - y)^2 + R^2)^{1/2} $$

where $R^2 > 0$ is a constant.

By substituting the $n$ known data points into the above equation, we get a system of equation of the form of $n$ unknowns $c_i, i = 1, 2, \ldots n$.

$$ \sum_{j=1}^{n} c_j ((x_j - x_i)^2 + (y_j - y_i)^2 + R^2)^{1/2} = z_i \quad (3.2) $$

By solving the system of equation, we can use equation 3.1 to estimate the data value in position $(x, y)$.

Apparently, this approach is a global method and can be applied in the irregular data estimation. It is shown that the result is generally satisfactory for small data size. The only difficulty here is the selection of the value $R^2$ which is highly dependent on the given data [17].

### 3.4 The New methods

#### 3.4.1 Extended Piecewise Hardy Interpolation

One difficulty with the Hardy's method is that we have to solve a large number of system of equations if the data set is large, say 1000 data points, which may be the
case in many practical applications. Solving such large system of equations is rather time consuming. An alternative is to combine the local method with this method by first triangulating the data with Delaunay triangulation, then estimating the surface passing through all the triangles sharing one common point with the Hardy's method, as shown in Figure 3.6. We can get three such surfaces for every triangle and by combining them as in the Gauss interpolation in the last session, we can get a \( C^1 \) surface estimation.

The extended piecewise Hardy's method could be better than the Gauss interpolation, because the three surfaces passing through all the vertices of the triangle is a better estimation than the best fit plane. Furthermore this method can also be used in the three dimensional data estimation by three dimensional triangulation of the data set[13], since Hardy's method, unlike most of the method in the surface estimation, can be generalized in the three dimensions. Because triangles sharing one common point are usually less than 10, we have about 10 data points, therefore about 10 equations to be solved in the extended piecewise Hardy surface estimation for each triangle vertex, \textit{i.e.} for each data point. Instead of solving one large system of equations, we need to solve \( n \) such system of equations in order to estimate the entire domain. The time complexity for solving a system of equations of \( n \) variables is \( O(n^3) \). In this method, suppose we solve a system of equations of \( m \) variables for every vertex, then the time needed to solve the whole system is the order of \( O(nm^3) \), where \( m \) is around 10 and not related to \( n \). So the time to solve the system of equations for the whole system in this method is linear to the total number of data points, which is a high speed up for the global Hardy's method. When considering the overhead incurred in the piecewise method to triangle the data points, to traverse the triangle in searching
the neighbor points in order to get the system of equations of the vertex, the speedup may less than quadratic. Our experimental results show that for problems with 1000 points, the speed up is about 80. The results on the computer generated and real data show that this method generates visually satisfiable estimation.

The difficulty with both the global and extended piecewise Hardy's method is to choose the Hardy constant \( R \). In the extended case, each system of equations, i.e., each data points, can associate with its own Hardy constant according to its surrounding points. In our algorithm, we choose the Hardy constant for each point proportional to the shortest distance between two points in the neighbor. The proportional constant is the same for all the points and is given by the user.

### 3.4.2 Bezier Triangle Interpolation

The method is based on the De Casteljau's work on the regular triangle for surface estimation. Like De Casteljau algorithm for the three dimensional curve, the surface estimation can also be expressed by the Bernstein polynomial in barycentric form with \( u, v \) and \( w \) as the barycentric coordinate of a point in the triangle. For a regular triangle of subdivision 4, we have the Bernstein coefficient as the following:

\[
\begin{align*}
  v^5 & \\
  5v^4w & 5vw^4 \\
  10v^3w^2 & 20uv^3w & 10u^2v^3 \\
  10v^2w^3 & 30uv^2w^2 & 30u^2v^2w & 10u^3v^2 \\
  5vw^4 & 20uvw^3 & 30u^2vw^2 & 20u^3vw & 5u^4v \\
  w^5 & 5w^4u & 10w^3u^2 & 10w^2u^3 & 5u^4w & u^5
\end{align*}
\]

The regular Bezier triangle surface estimation is parametric which preserves the properties of affine invariance as in the three dimensional curve. Two Bezier triangle patches can be combined together along a common edge to form a \( C^1 \) surface if and only if the subtriangles along the common edge of the two are coplanar and each pair is an affine map of the two triangles [7]. That is, if the two triangles are regularly subdivided triangles and all pairs of subtriangles sharing the common edge
are coplanar, then the surface formed by these two triangles is $C^1$, as shown in Figure 3.7.

In our approach, all the subtriangles in $T_1$ or $T_2$ are the same and are similar to the corresponding large one. If the barycentric coordinate is denoted by $(u, v, w)$ as show in the figure, the barycentric coordinate for the interior points can be easily obtained. Assume the triangle $T_1$ and $T_2$ are sharing an edge as shown in Figure 3.7, the value for the vertices are $(x_a, y_a, z_a), (x_u, y_u, z_u), (x_j, y_j, z_j)$ and $(x_{ui}, y_{ui}, z_{ui})$, we can obtain the Bezier control points for the triangle $T_1$ in the following way in order to form a $C^1$ surface along the border of the two triangles:

1. Find the best fit planes $L_a(u, v, w), L_j(u, v, w), L_{ui}(u, v, w)$ passing through point $a, j, u'$ respectively, where $u, v, w$ are barycentric coordinate of a point in the triangle.
2. Points $a, b, c, g, h$ and $l$ are obtained from plane $L_a(u, v, w)$, where $(u, v, w)$ are the coordinates for the points respectively. Similarly, The other two set of six points $d, e, f, j, k, o$ and $p, s, t, r, q$ can be obtained from the $L_j(u, v, w)$ and $L_{ui}$ planes.
3. Now the only undecided points in the triangle $T_1$ are the three points in the middle triangle, i.e. point $i, m$ and $n$. The triangles on $T_1$ other than $\Delta cdi$ on the border $af$ are all coplanar with the subtriangles of $T_2$. In order to make subtriangle $\Delta cdi$ coplanar with $\Delta cdi'$, we may select point $i$ and $i'$ as two points of the plane passing
through point \(c, d\) and parallel to the line \(u'u''\). Similarly, select point \(m, n\) this way according to the triangle sharing edge with \(T_1\) on the other two sides.

By the above three steps, all the points in \(T_1\) are determined by the triangle \(T_1\) and its neighbors, furthermore, all the subtriangles on the edge of \(T_1\) are coplanar with subtriangles sharing the same edge. So the resultant surface formed in such way is a \(C^1\). It is easy to show that the above method is invariant under Euclidean transformation.

The point to be estimated can be obtained by the regular Bezier triangle estimation method with the subdivision of 4 by the following formula:

\[
X(u, v, w) = v^5 x_u + 5v^4 wx_z + 5uv^4 x_t + 10v^3 w^2 x_p + \\
20uv^3 wx_q + 10u^2 v^3 x_r + 10v^2 w^3 xl + 30uv^2 w^2 x_m + \\
30u^2 v^2 wx_n + 10u^3 v^2 x_k + w^5 x_a + 5w^4ux_b + \\
10w^3 u^2 x_c + 10w^2 u^3 x_d + 5u^4 wxe + u^5 xf
\]

where \(u, v, w\) is the barycentric coordinate of the point.

Similarly, we can get the \(Y(u, v, w)\) and \(Z(u, v, w)\) by replacing the \(x\) coordinate with the \(y\) and \(z\) coordinate in the above formula.
Table 3.1: error statistics for computer generated scattered data interpolation.

### 3.5 Experimental Results

We tested the three methods with both computer generated data and a practical data set for the environmental problem.

#### 3.5.1 Results with Computer Generated Data

The computer generated data are obtained from the following three functions by random sampling of the coordinates over the range of \((-1,1)\). We use 100 points for each data set. The frames are displayed in 50 by 50 grid.

The following three functions are used in our test.

\[
F_1 = \exp(-1.0(x^2 + y^2))
\]

\[
F_2 = \frac{\tanh(4.5(y - x) + 1)}{9.0}
\]

\[
F_3 = \sin(2\pi(x + 1))\exp(-y - 2.5)
\]

Table 3.1 lists the average error (AveError), the absolute error (AbsError) and their correspondent standard deviations of the estimated surface functions. From the table, we can see that the three methods perform not much difference with the piecewise Hardy’s method slight better than the other two. This is also true for other functions.

#### 3.5.2 Results with Practical Data Set

The practical data set has 72 randomly distributed data points. Because we do not know the exact value for the entire domain, we evaluate the different methods by
Figure 3.8: Delaunay triangulation for 100 data points

Figure 3.9: Left: Gauss for F1. Right: Bezier for F1.

Figure 3.10: Left: Hardy for F1. Right: Gauss for F2.
Figure 3.11: Left: Bezier for F2. Right: Hardy for F2.

Figure 3.12: Left: Gauss for F3. Right: Bezier for F3.

Figure 3.13: Hardy for F3
<table>
<thead>
<tr>
<th>method</th>
<th>AveError</th>
<th>AveStd</th>
<th>AbsError</th>
<th>AbsStd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss</td>
<td>0.0166</td>
<td>2.194</td>
<td>1.520</td>
<td>2.188</td>
</tr>
<tr>
<td>Bezier</td>
<td>0.0157</td>
<td>2.169</td>
<td>1.495</td>
<td>2.164</td>
</tr>
<tr>
<td>Hardy</td>
<td>-0.0095</td>
<td>1.913</td>
<td>1.308</td>
<td>1.916</td>
</tr>
</tbody>
</table>

Table 3.2: error statistics for computer generated scattered data interpolation.

Figure 3.14: Original data.

omitting one point and estimating it with the different methods. The method is better for this kind of data if the estimated value is closer to the original value. We take the square root of the differences as a measure of the accuracy of the different methods. The wire frame functions with different methods are shown in the following figures. The error statistics is shown in table3.2.

3.5.3 Piecewise versus Global Hardy's Method

Both the speed and the accuracy are compared for the piecewise and global Hardy's methods. Table3.3 is the time vs data points for global and piecewise Hardy's methods. The speedup for each group are shown in Table 3.4. The data are also shown in Figure 3.18, Figure 3.19 and Figure 3.20. The interpolation is done with 40 x 40 grids.
Figure 3.15: Gauss estimation data.

Figure 3.16: Bezier estimation.

Figure 3.17: Hardy estimation.
Table 3.3: time for global and piecewise Hardy’s method

<table>
<thead>
<tr>
<th>#points</th>
<th>method</th>
<th>SlvEqu(sec)</th>
<th>Interp(sec)</th>
<th>Total(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>piecewise</td>
<td>2.27</td>
<td>3.91</td>
<td>6.21</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>17.80</td>
<td>12.68</td>
<td>30.49</td>
</tr>
<tr>
<td>400</td>
<td>piecewise</td>
<td>5.16</td>
<td>4.55</td>
<td>9.71</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>123.55</td>
<td>34.89</td>
<td>148.44</td>
</tr>
<tr>
<td>600</td>
<td>piecewise</td>
<td>10.46</td>
<td>5.85</td>
<td>16.32</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>390.16</td>
<td>37.60</td>
<td>427.76</td>
</tr>
<tr>
<td>800</td>
<td>piecewise</td>
<td>12.76</td>
<td>6.05</td>
<td>18.80</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>900.73</td>
<td>49.37</td>
<td>950.09</td>
</tr>
<tr>
<td>1000</td>
<td>piecewise</td>
<td>21.50</td>
<td>9.03</td>
<td>30.53</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>1749.36</td>
<td>59.46</td>
<td>1808.83</td>
</tr>
</tbody>
</table>

Table 3.4: speedup of piecewise over global Hardy’s method

<table>
<thead>
<tr>
<th>#points</th>
<th>SlvEqu(sec)</th>
<th>Interp(sec)</th>
<th>Total(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>7.8</td>
<td>3.2</td>
<td>4.9</td>
</tr>
<tr>
<td>400</td>
<td>24.0</td>
<td>5.5</td>
<td>15.3</td>
</tr>
<tr>
<td>600</td>
<td>37.3</td>
<td>6.4</td>
<td>26.2</td>
</tr>
<tr>
<td>800</td>
<td>70.6</td>
<td>8.2</td>
<td>50.5</td>
</tr>
<tr>
<td>1000</td>
<td>81.4</td>
<td>6.6</td>
<td>59.2</td>
</tr>
</tbody>
</table>

Figure 3.18: time versus number of data points for global Hardy’s method.
Figure 3.19: time versus number of points for piecewise Hardy's method.

Figure 3.20: speedup of piecewise over global Hardy's method.
From the figures and the table we can see that the time complexity for global Hardy’s method is $O(n^3)$ and the time complexity for the piecewise Hardy’s method is near linear, *i.e.*, $O(n)$. The reason that the speedup is little less than the order of $O(n^2)$ is because the overhead incurred in the piecewise method in searching for neighboring points in order to solve the system of equations associated with each point. One advantage of the piecewise method over the global is that different Hardy constant $R$ can be selected in a reasonable time period to make to interpolation near optimum. For a problem with data points of one thousand, the gain is obvious.

The Hardy’s method is know to be one of the best methods in dealing with the scattered data interpolation. We compared the global and piecewise Hardy’s method regarding the accuracy of the interpolation. The comparison is done with the above three function, *i.e.*, $F_1$, $F_2$ and $F_3$. Two hundred points are used in the evaluation. Different Hardy constant are tested for each function and the statistical data in the table are the one with best statistical result among different constant selections. The result are shown in Table 3.5.

From the table, we see that the piecewise Hardy’s method performs better than the global method in all the cases. The reason is partly because that in the global method there is only one Hardy constant for each run, while in the piecewise method, each system of equations associated with a data point has its own Hardy constant in accordance with its own environment as described in the above section, thus allows more accurate surface fitting.

<table>
<thead>
<tr>
<th>function</th>
<th>method</th>
<th>AveError</th>
<th>AveStd</th>
<th>AbsError</th>
<th>AbsStd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>piecewise</td>
<td>0.000</td>
<td>0.0007</td>
<td>0.0004</td>
<td>0.0006</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>-0.0087</td>
<td>0.0212</td>
<td>0.0190</td>
<td>0.0128</td>
</tr>
<tr>
<td>2</td>
<td>piecewise</td>
<td>0.002</td>
<td>0.0027</td>
<td>0.0012</td>
<td>0.0024</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>0.003</td>
<td>0.0033</td>
<td>0.0017</td>
<td>0.0029</td>
</tr>
<tr>
<td>3</td>
<td>piecewise</td>
<td>-0.0001</td>
<td>0.0055</td>
<td>0.0025</td>
<td>0.0049</td>
</tr>
<tr>
<td></td>
<td>global</td>
<td>-0.0016</td>
<td>0.0166</td>
<td>0.0114</td>
<td>0.0121</td>
</tr>
</tbody>
</table>

Table 3.5: error statistics for piecewise and global Hardy’s method
Chapter 4

Contouring of Surface Data

4.1 The Problem

The surface contour problem can be stated as following:

*Giving a set of bivariate data points in the form $Z(x, y)$, find all the points $(x, y)$ where the $Z(x, y)$ has the same value $v$, i.e. $Z(x, y) = v$.*

Contouring of surface data is widely used in surface data visualization. It gives better understanding of the surface by labeling the iso-valued lines of the surface and displaying them on a plane. Many contour algorithms and programs are available. The underlying assumption about the contouring of the surfaces is the continuity, i.e., all contour lines are closed around peaks or valleys of the surface except at the boundary.

In contouring the surface, one needs some methods to interpolate the data at certain points. There are two general approaches[25]; fitted functions and weighted averages or global and piecewise interpolation. Fitted function methods determine the parameters of an analytic function. Then, using these parameters, the function is evaluated at a given location to obtain the height of the representative surface. The piecewise methods use only the neighbors of data within range of the interpolation point.

The surface contouring can also be divided into regular surface data and scattered surface data contouring according to the input data points. As in the interpolation case, regular data is somewhat easier, but in contouring, special cases must be taken care of, which make the algorithm difficult to implement[30] or ambiguous.

One problem in contouring is the smoothness of the contour line. Most of the algorithms in the regular grid data contouring assume that the grid is fine enough in
order to generate reasonable results. But in some cases, the original data may not satisfy the assumption.

In the following sections, we first give a description of the general methods in contouring regular grid surfaces, then discuss the triangle subdivision method which can both generate arbitrarily smooth contour lines and avoid the ambiguity that some of the known algorithms may have in certain situations.

4.2 Grid Surface Data Contouring

Most of the algorithms in grid surface contouring assume that the grid point is fine enough and linear interpolation is used. The contouring lines can be found in the rectangular grid by linear interpolation as show in Figure 4.1. The problem with this approach is shown in case c and d, which may generate ambiguity. To avoid the problem, a middle point by averaging the four points may be used which results in the following complete 12 cases[30]. With the 12 cases, the programming may become very complex, although some data structures can be used. Another approach is first to interpolate the rectangular data with some methods, say the cubic splines as we described in the previous chapter. Then we get the equation for the patch passing through the four points:

\[
X(u, v) = \sum_{i=0}^{3} a_i x^i \\
Y(u, v) = \sum_{i=0}^{3} b_i y^i
\]
With the cubic equation, we can compute all the points satisfying the condition of \( Z(u, v) = z \). This approach can generate smooth contour lines, because the interpolated surface is \( C^2 \) continuous. The disadvantage with this method is that we need to find the root of the equation, which is time consuming.

### 4.3 The Triangle Subdivision method

#### 4.3.1 Scattered data

The scattered data surface contouring is different from the regular data because of the topology. The contouring can also be implemented in two ways, by a general fitting method and by the piecewise interpolation.

In the piecewise contouring approach, as in the piecewise interpolation, triangulation may be used. In [22], Sawkar et al, uses three steps contouring the scattered surface:

1. triangulation: to triangulate the original data to get a triangle net.
2. subtriangulation: each triangle can form a quadrilateral with one of its three neighbors. The intersection point of the diagonals is common to the two triangles. The value at the intersection point is calculated by the known point in the vertices of the quadrilateral. A triangle may have none, one or two such intersections, which is also called pseudo points, as shown in the figure. Three pseudo points can subdivide the triangle into four triangles while one pseudo point can only subdivide the triangle into two. The subdivision process can be continued by using the new triangles.

3. interpolation: linear interpolation is used after the triangle subdivision.

This approach may have the following three disadvantages:

1. the subdivision is not uniform, some triangles may be split into more triangles than others.

2. by subdivision, the memory to hold the data and the triangle mesh grows exponentially. The triangle relations also need to be updated to hold the new relation.
3. the contour lines may not be accurate for some data set, since the interpolation to obtain the pseudo points uses only four points.

Based on the above observations, we propose a new triangle subdivision approach. The method is also based on the triangle net for the scattered data and can be described as follow:

1. in contrast to subdivide all the triangles at the same time, we subdivide one triangle recursively up to the desired level, and calculate the contour lines at the same time.

2. each triangle is evenly subdivided into four triangles by interpolating the middle points on the three edges.

3. the interpolation is an input parameter to the contouring program, not determined by the program itself. The interpolation may use one of the methods described in the above chapter.

The triangle subdivision is also shown in Figure 4.4. With the triangle itself, we have no ambiguity problem, but when considering the process of triangulation, we still have two choices when four points are on a rectangular grid, but the ambiguity will be resolved with subdivision unless there is an extremely narrow ridge along the diagonal. As shown in Figure 4.5, the ambiguity problem exists only if there is always at least one rectangle with the ambiguity structure after a number of subdivisions.

In the algorithm, we assume that we have the queue operation functions, initial queue (InitQueue), enter queue (Enqueue) out queue (OutQueue) and empty queue (EmptyQueue) for triangle net traverse. The function FindMiddlePoints finds the three middle points of the input triangle with the supplied function.

The pseudo algorithm is illustrated as following:

/* Contour the scattered data: by triangle subdivision */
/* inputs : */
level[] : array of length nlevels.
Figure 4.4: Triangle subdivision and contouring

Figure 4.5: Subdivision and resolving ambiguity
Depth : subdivision level.

OutPutLine : function to output contour line.

InterpPoint : function to interpolate the point.

Contour(level, nlevels, Depth, OutPutLine, InterpPoint)* /

float level[ ];

int nlevels, Depth;

void (*OutPutLine)(POINT *, POINT *);

int (*InterpPoint)(float x, float y, float *z);

{ int i; TRIANGLE *t; POINT P[3];

/* initial the triangle queue */

t=Triangle; InitQueue(); EnQueue(t);

/* traverse the triangle net */
do {

OutQueue(& t); t->visit=1;

for(i=0;i<3;i++) {
/* if edge has neighbor and has not been visited */

if((t->E[i]!=NULL) && (t->E[i])->visit!=1 )

EnQueue(t->E[i]);
}

for(i=0; i<3; i++) /* get triangle vertices */

P[i]=Point[t->P[i]];

/* subdivide the triangle */

trace(t, P, level, nlevels, 0, Depth, OutPutLine, InterpPoint);
}

while(EmptyQueue()!=1);

return;
}

/* function:trace the contour line. */
/* the trace is recursively called. */
/* input :P[]: triangle three points, */
/* t: triangle pointer, nlevels in level[] */

trace(t, P, level, nlevels, currddepth, Depth, OutPutLine, InterpPoint)

struct triangle *t;

struct point P[ ];

float level[ ];

int nlevels, currdepth, Depth;

void (*OutPutLine)();

void (*InterpPoint)(struct triangle *t, float x, float y, float *z);

{ POINT Pm[3], P0[3], P1[3], P2[3], P3[3];

  if(currdepth<Depth) { /* subdivide the triangle */
    FindMiddlePoints(t, P, Pm, InterpPoint);
    P0[0]=P[0]; P0[1]=Pm[0]; P0[2]=Pm[2];
    trace(t, P0, level, nlevels, currddepth+1, Depth, OutPutLine, InterpPoint);
    P1[0]=P[1]; P1[1]=Pm[1]; P1[2]=Pm[0];
    trace(t, P1, level, nlevels, currddepth+1, Depth, OutPutLine, InterpPoint);
    trace(t, P2, level, nlevels, currddepth+1, Depth, OutPutLine, InterpPoint);
    trace(t, Pm, level, nlevels, currddepth+1, Depth, OutPutLine, InterpPoint);
  }
  else { /* out put the conotur */
    OutPutContour(P, level, nlevels, OutPutLine);
  }
}

4.3.2 Grid Data

In surface contouring, we have two problems in the rectangular data with the methods discussed in section 4.2. With the triangle subdivision method, these two problems can be solved easily. So, we may also divide the rectangle into triangles and use the triangle subdivision to contour the surface as in the scattered data. A rectangle can be divided into two or four triangles as shown in Figure 4.6. With the two triangle
subdivision, we still have the ambiguity problem to select one of the diagonals, so four triangle subdivision is used in our algorithm. The basic idea is the same except that we do not need the triangle network.

4.4 Experimental Results

Figure 4.9 is the contour by the scattered method for Figure 4.8 with the data distribution in Figure 4.7
Figure 4.7: Data distribution.

Figure 4.8: Wire frame for contouring data.
Figure 4.9: Contour plotting with triangle subdivision method.
Chapter 5

Graphics User Interface (GUI) of the Scattered Data Interpolation and Visualization (SDIV) Package

Graphics User Interface (GUI) is an important part for a user friendly package. Windows and menus are common in the programs developed today. The tools and libraries, such as X11/Motif in UNIX and Microsoft Windows in IBMPC, also facilitate the task for user friendly interface. In this chapter, we give an overview of the Scattered Data Interpolation and Visualization package, the usage of the program as well as some results.

The package was implemented in both Sun Sparc Station running UNIX and X window and IBMPC running MSDOS. We describe only the X11 implementation in the following sections. The MSDOS implementation has the same feature but with different operations.

5.1 The GUI under X11

There are two windows in the display for this package, one is control panel, and the other is the graphics window. The two windows are shown in Figure 5.1 and Figure 5.2 respectively. The control panel is a window consisting of control buttons to operate the display output in the graphics window. The control window has the following functions:

- display method selection.

- interpolation method selection.

- file handling.
Figure 5.1: The control panel

Figure 5.2: The graphics window
• information and help.

• graphics rotation around X, Y, and Z axis.

• graphics translation.

• amplification along X, Y, Z directions.

As shown in Figure5.1, the display method, interpolation method, file handling and control, and information selections are implemented by pull-down menus and buttons. While graphics transformation operations, rotation, translation and amplification, are implemented as graphical objects to make the operation intuitive.

5.2 Command Line Options

The program can be activated by the command:

`xsdv [−f filename] [−d display] [−g geometry] [−fx frameX] [−fy framrY]

The −f option specifies the data file to be read and interpolated. The file may be in one of the three formats described in the following section. If no file name is specified in the command, the program reads `demo.dat` file. The program interpolates the data with the default interpolation method after the data is loaded into memory.

The −d option selects the host display, it can display the results across the network. The default is the current host display.

The −g option specifies the `Graphics` window size. The option 400x500 specifies the width and height of the window as 400 and 500. The default window size is 500x500.

The −fx and −fy options are used to specify the number of points in X or Y direction used in wire frame display.

5.3 Data File Formats

Presently, the program recognizes three file formats, the XYZ.POINTS, IJZ.POINTS and Z.ONLY format. The XYZ.POINTS format data file is a list of x, y, z triples with the first line of the file marked with XYZ.POINTS. This format is common in the scattered data interpolation and estimation. The IJZ.POINTS is a list of i, j, z
triples with the first line marked with IJZ.POINTS and the following four lines with the information:

- **line 1**: IJZ.POINTS interpolated data
- **line 2**: FramePointX FramePointY
- **line 3**: minx maxx
- **line 4**: miny maxy
- **line 5**: minz maxz
- **line 6**: i j z
  
  The i,j in the triple is the x and y index to the array holding the Z values. The FramePointX and FramePointY are the dimension of the array in the two directions.

The Z.ONLY format is almost the same as the IJZ.POINTS with only single z value in the data list. The IJZ.POINTS format is used because in the scattered data, interpolation at certain place may not be available even after interpolation. Only the points indexed by the i,j values can be obtained.

The IJZ.POINTS can be used to display interpolated scattered data, the Z.ONLY data format can be used to display the rectangular data sets.

### 5.4 Display Method Selection

There are four items, Points, Wire Frame, Contour, Frame and Contour as shown in Figure5.3. The Points selection is used to display the original points read from the file. The points are displayed in a box with dashed line on its bottom. Each point is displayed as a 4×4 little rectangle in the projected space. A dashed line is connected from the point to the bottom of the box to make the data points looks 3 dimensional as shown in Figure5.4. This display method can also be used to adjust the the data points and the frame in a proper position by using the graphics rotation, translation, and amplification buttons on the Control Panel. The other display method may take longer time to put the object on the desired place.
The Wire Frame selection displays a wire frame of the interpolated surface with hidden line elimination, Figure 5.2. The wire frame can be rotated, translated and amplified with the correspondent operations.

The Contour selection displays the contour map of the interpolated data set as shown in Figure 5.5. It occupies the whole graphics window with graphics transformations disabled.

The Frame and Contour displays wire frame of the interpolated surface data with the contour map projected underneath it, as shown in Figure 5.6. This method enables the simultaneous view of the contour map and the wireframe in the same window.

The Triangle network selection displays the Delaunay triangulation of the original data points. It can be operated by the graphics transformations to make the display as a wireframe of triangles as shown in Figure 5.7.

5.5 Interpolation Method Selection

Four methods for the scattered data interpolation are implemented at present, Gauss, Bezier, Hardy and Quadratic. The details of the method are described in the following sections. The different interpolation methods can be selected through the
Figure 5.4: Display points selection

Figure 5.5: Display contour selection
Figure 5.6: Display frame and contour selection

Figure 5.7: Display triangle network selection
Interpolation Method menu as shown in Figure 5.8.

5.6 Control Selection

The menu for Control Selection is shown in Figure 5.9.

The Load file selection can be used to load a new file for interpolation and display. When select this item, you need to input the file name through the standard input. If the program can not open the file, it will ask you to input the file name again.

The Save file selection is to save the interpolated data into a file. The data is saved in IJZ.POINTS format, i.e., the first two data are integers representing the \( x \) and \( y \) index of the data point in the array, the third one is the interpolated datum at the point indexed by the first two.

The Reset selection is used to reset the rotation, translation and amplification to their initial value.

The Exit selection exit the program.
5.7 Information Selection

The Information selection is shown in Figure 5.10. It provides the information about the program itself, the graphics transformation, the rotation angle, the amplify factor, the translation in the graphics window and the information about the data file, the interpolation method and display method as shown in the following:

***************
SCATTERED DATA INTERPOLATION AND DISPLAY
***************

COMPUTER GRAPHICS LABORATORY
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF NEVADA, LAS VEGAS
Weibao Wu & E. Yfantis
July, 1992

File Name : demo.dat
Number of Points : 200
5.8 Rotation, Translation and Amplification

The rotation of the graphics data is operated through the three clock like panels, numbered X, Y and Z respectively. The graphics in the graphics window can be rotated around X, Y and Z axis by clicking the mouse button in the clock. The clock hand will move to the cursor position and rotate the graphics. Press the mouse button down and move the cursor around the clock can also make the clock hand...
stick to the cursor and rotate the graphics data. By pressing the two little buttons inside the rotation window increases or decreases the rotation by one degree. The degree rotated around each axis is displayed between the two little increase/decrease buttons. The same degree in the three axis may not result the same graphics position because of the rotation is order dependent. The rotation buttons can be seen in Figure5.1.

The translation is performed by pressing the mouse button in one of the two translation windows for $X$ or $Y$ directions. The black rectangle in the translation button marks the coordinate origin in that direction. Like in the rotation window, the black rectangle in the translation button can also stick to the cursor when the mouse button is pressed while moving in the translation window. The translation is on the image space, i.e. in pixels.

Amplification of the graphics data is controlled by the three little buttons marked \% +10\% or -10\% in the amplification window for $X$, $Y$ and $Z$ axis. Press one of the three buttons in the window will increase or decrease the graphics size by 10\% percent in the direction.

5.9 Viewing Data in the graphics window

Data can be evaluated in the graphics window when all the three rotation angles are zero. The value at certain point in the graphics window is displayed in the upper left corner of the graphics window when the mouse button is pressed in the graphics window. The data are displayed in the $(x, y, z)$ format. If the data point is outside the scope of the interpolation, the $z$ value may be displayed as unknown.
Bibliography


[27] Yfantis E. accepted for publication, “A New Quadratic and BiQuadratic Algorithm For Curve and Surface Estimation”, Computer Aided Geometric Design
Appendix A

Source Code for the XSDV
CFLAGS = -D -I/local/X11R4/include
#include #INCLUDES = -I/local/X11R4/include
LIBS = -L/local/X11R4/lib -lx11 -lm
LFLAGS = -L/local/X11R4/lib -lx11
CC = gcc
OBJ = main.o xgraphics.o xpanel.o xwinutil.o xbutton.o xinit.o 
     xmenu.o misc.o fileio.o drawframe.o setaxis.o interp.o 
     bezier.o phardy.o ftriangle.o drawcontour.o equation.o

xpanel :$(OBJ) Makefile
    $(CC) $(CFLAGS) -o xsvc main.o xgraphics.o xpanel.o xwinutil.o 
        xbutton.o xinit.o xmenu.o misc.o fileio.o drawframe.o setaxis.o 
        interp.o bezier.o phardy.o ftriangle.o drawcontour.o equation.o
    $(LIBS)
main.o : main.c
    $(CC) $(CFLAGS) -c main.c
xgraphics.o: xgraphics.c
    $(CC) $(CFLAGS) -c xgraphics.c
xpanel.o: xpanel.c
    $(CC) $(CFLAGS) -c xpanel.c
xinit.o: xinit.c
    $(CC) $(CFLAGS) -c xinit.c
xwinutil.o: xwinutil.c
    $(CC) $(CFLAGS) -c xwinutil.c
xbutton.o: xbutton.c
    $(CC) $(CFLAGS) -c xbutton.c
xmenu.o: xmenu.c
    $(CC) $(CFLAGS) -c xmenu.c
misc.o: misc.c
    $(CC) $(CFLAGS) -c misc.c
fileio.o: fileio.c
    $(CC) $(CFLAGS) -c fileio.c
drawframe.o: drawframe.c
    $(CC) $(CFLAGS) -c drawframe.c
drawcontour.o: drawcontour.c
    $(CC) $(CFLAGS) -c drawcontour.c
setaxis.o: setaxis.c
    $(CC) $(CFLAGS) -c setaxis.c
interp.o: interp.c
    $(CC) $(CFLAGS) -c interp.c
bezier.o: bezier.c
    $(CC) $(CFLAGS) -c bezier.c
ftriangle.o: ftriangle.c
    $(CC) $(CFLAGS) -c ftriangle.c
phardy.o: phardy.c
    $(CC) $(CFLAGS) -c phardy.c
equation.o: equation.c
    $(CC) $(CFLAGS) -c equation.c
/* main program for interactive scattered data interpolation and display
 * Weibao Wu, Junly 9, 1992
 * Department of Computer Science
 * University of Nevada, Las Vegas
 */
/* function prototypes */
void CommandLineParsing(int, char **);
void GraphicsScreen(void);
void OpenXDisplay(char *);
void ControlPanel(void);
void EventLoop(void);
void ReadData(char *filename);
void error(char *);
void allocFrameData(float **FrameData, int nx, int ny);

#include <stdio.h>
#include <math.h>
#include "globals.h"

main(int argc, char **argv)
{
    CommandLineParsing(argc, argv);
    ReadData(filename);
    /* open connection with the X server */
    OpenXDisplay(argv[0]);
    /* create graphics screen */
    GraphicsScreen();
    /* create X panel */
    ControlPanel();
    /* event handling loop */
    EventLoop();
}

void error(char *s)
{
    printf("Error: %s
", s);
    exit(1);
}

interp.c
/* Weibao Wu Oct.7, 1991
 * Gauss interpolation.
 * change June 11, 92 */

#include <stdio.h>
#include <math.h>
#include "image.h"
#include "globals.h"

#define sqr(x) (x)*(x)
int OnBoundary(struct triangle *t);

float G_Interpolation(struct triangle *t, float x, float y,
                       float *nx, float *ny, float *nz, float *Area);
int Gauss_InterpPoint(float x, float y, float *z);
int Gauss_InterpPoint2(struct triangle *t, float x, float y, float *z);

/*@ Interpolation the data in the array Point[TotalPoints] in ROI */
void Gauss_Interp(point, x1,y1,x2,y2, deltax, deltay, OutPutPoint)
  float x1,y1,x2,y2;
  float deltax,deltay;
  void (*OutPutPoint)(int, int, float);
{
  int i,j;
  int l,m;
  float x,y,z;
  struct triangle *t;
  if(xl==x2 || yl==y2)
    return;
  if(xl>x2) {
    x=xl; xl=x2; x2=x;
  }
  if(yi>y2) {
    y=yl; yl=y2; y2=y;
  }
  Tranverse(Triangle,0); /* Set visit sign=0 */
  for(i=0, y=yl; i<FramePointX; i++, y+=deltay)
    for(j=0, x=xl; j<FramePointY; j++, x+=deltax) {
      if(GaussJnterpPoint( x, y, &z)) /* interpolated */
        (*OutPutPoint)(j, i, z);
    }
}

int GaussJnterpPoint(float x, float y, float *z)
{
  int l;
  float TArea[3];
  struct triangle *t;
  static float nx[3],ny[3],nz[3];
  static struct triangle *ct=NULL;
  l=InTriangle(&t,x,y);
  if(l<0)
    return 0; /* do not interpolate */
  if(ct!=t){ /* if new triangle */
    TriangleNormals(t,nx,ny,nz); /* Three normals at the end point of t*/
    ct=t;
  }
  TriangleAreas(t,x,y,TArea); /* Three areas in triangle t */
*z=G_Interpolation(t,x,y,nx,ny,nz,TArea);
    return(1);
}

/*! interpolation a point for triangle t */
int Gauss_InterpPoint2(struct triangle *t, float x, float y, float *z)
{
    int i;
    float TArea[3];
    static float nx[3],ny[3],nz[3];
    static struct triangle *ct=NULL;
    if(ct^t) { /* if new triangle */
        TriangleNormals(t,nx,ny,nz); /* Three normals at the end point of t*/
        ct=t;
    }
    TriangleAreas(t,x,y,TArea); /* Three areas in triangle t */
    *z=G_Interpolation(t,x,y,nx,ny,nz,TArea);
    return(1);
}

/*! Interpolation: given (x,y) three plane normals, three areas. */
float G_Interpolation(struct triangle *t,float x,float y,
                          float nx, float ny, float nz, float *Area)
{
    int i,j;
    float t0,t1,t2,z;
    float w[3],f[3];
    float tx[3],ty[3],tz[3],ttz[3];
    /* Compute the real value */
    for(i=0;i<3;i++) {
        tx[i]=Point[t-+P[i]].x;
        ty[i]=Point[t-» P[i]].y;
        tz[i]=Point[t— *P[i]].z;
    }
    /* Compute Weight */
    for(i=0;i<3;i++) {
        w[i]=(sqr(Area[(i+2)%3]) +sqr(Area[i]))*sqr(Area[(i+1)%3]);
        t1=sqr(x-tx[(i+1)%3]) +sqr(y-ty[(i+1)%3]);
        t1=tt1*(sqr(x-tx[(i+2)%3]) +sqr(y-ty[(i+2)%3]));
        w[i]=w[i]*t1;
    }
    for(i=0;i<3;i++) {
        if(x==tx[i] & & y==ty[i]) {
            w[i]=1.0; w[(i+1)%3]=0.0; w[(i+2)%3]=0.0;
            break;
        }
    }
    /* w[0]=Area[1]; w[1]=Area[2]; w[2]=Area[0]; */
t1=w[0]+w[1]+w[2];
for(i=0;i<3;i++)
  w[i]=w[i]/t1;

for(i=0;i<3;i++)
{  
t1=(nx[i]*(x-tx[i])+ny[i]*(y-ty[i]))/nz[i];
  tttz[i]=-t1+tz[i];
}

for(i=0;i<3;i++)
{  
t2+=tttz[i]*w[i];
}
return(t2);

% bezier.c

/* bezier.c */
/* interpolation with Bezier triangle method */
/* Weibao Wu */
*/

#include <stdio.h>
#include <math.h>
#include "image.h"
#include "globals.h"

int OnBoundary(struct triangle *t);
float BJnterpolation2();
float PlaneValue();
int Bezier_InterpPoint(float x, float y, float *z);
int Bezier_InterpPoint2(struct triangle *t, float x, float y, float *z);

/* Interpolation the data in the array Point[TotalPoints] in ROI */
void Bezier_Interp( float x1,y1,x2,y2, deltax, deltay, OutPutPoint)
float x1,y1,x2,y2;
float deltax, deltay;
void (*OutPutPoint)(int, int, float);
{
  int i,j;
  int l,m;
  float x,y,z,t1;
  struct triangle *t;

  if(x1==x2 || y1==y2)
    return;
  if(x1>x2) {
    x=x1; x1=x2; x2=x;
  }
  if(y1>y2) {
y=y1; y1=y2; y2=y;
}
Tranverse(Triangle,0); /* Set visit sign=0 */
for(i=0; y=y1; i<FramePointX; i++, y+=deltay)
for(j=0; x=x1; j<FramePointY; j++, x+=deltax) {
    if(Bezier_InterpPoint(x, y, &z)) /* if interpolated */
        (*OutPutPoint)(j, i, z);
}

/* Interpolation2: given (x,y) three plane normals, three areas. */
float BzPoints[6][6];
float u,v,w;
struct triangle *t;
{
    int i,j;
    float x0,y0,x1,y1,x2,y2;
    float t0,tl,t2,z;
    float tx[3],ty[3],tz[3],ttz[3];
    /* Compute the real value */
    for(i=0;i<3;i++) {
        tx[i]=Point[t—>P[i]].x;
        ty[i]= Point[t->P[i]].y;
        tz[i]= Point[t->P[i]].z;
    }
    u2=u*u; u3=u2*u; u4=u3*u;
    v2=v*v; v3=v2*v; v4=v3*v;
    w2=w*w; w3=w2*w; w4=w3*w;
    /* for boarder */
    t2=0; /* u-v */
    t2 =(((BzPoints[0][0]*u+BzPoints[0][1]*5.0*v)*u+10.0*BzPoints[0][2]*v2)*u+
        BzPoints[0][3]*10.0*v3)*u+BzPoints[0][4]*5.0*v4)*u;
    /* v-w */
    t2+= (((BzPoints[0][5]*v+BzPoints[1][4]*5.0*w)*v+10.0*BzPoints[2][3]*w2)*v+
        BzPoints[3][2]*10.0*w3)*v+BzPoints[4][1]*5.0*w4)*v;
    /* w-u */
    t2+= (((BzPoints[5][0]*w+BzPoints[4][0]*5.0*u)*w+10.0*BzPoints[3][0]*u2)*w+
        BzPoints[2][0]*10.0*u3)*w+BzPoints[1][0]*5.0*u4)*w;
    /* middle points */
    t2+= 10.0*u2*v+w*(2.0*u*BzPoints[1][1]+3.0*v*BzPoints[1][2]);
    t2+= 10.0*u*v2+w*(2.0*v*BzPoints[1][3]+3.0*w*BzPoints[2][2]);
    t2+= 10.0*u*v*w*(2.0*w*BzPoints[3][1]+3.0*u*BzPoints[2][1]);
    return(t2);
}

int Bezier_InterpPoint(float x,float y,float *z) {

int l;
float nx[3], ny[3], nz[3];
float u, v, w;
struct triangle *t;
static float BzPoints[6][6];
static struct triangle *ct=NULL;
l=lnTriangle(&t,x,y);
if(l<0)
    return 0; /* do not interpolate */
if(ct!=t) { /* if the two times are not the same triangle */
    TriangleNormals(t,nx,ny,nz); /* Three normals at the end point of t*/
    TriangleBzPoints(t,BzPoints,nx,ny,nz); /* compute Bezier points */
    ct=t;
}
Barycentric(t,x,y,&u,&v,&w);
*z=B_Interpolation2(t,u,v,w,BzPoints);
return 1;
}

/* interpolation a point for triangle t */
int Bezier_InterpPoint2(struct triangle *t, float x,float y,float *z)
{
    int l;
    float nx[3], ny[3], nz[3];
    float u, v, w;
    static float BzPoints[6][6];
    static struct triangle *ct=NULL;
    if(ct!=t) { /* if the two times are not the same triangle */
        TriangleNormals(t,nx,ny,nz); /* Three normals at the end point of t*/
        TriangleBzPoints(t,BzPoints,nx,ny,nz); /* compute Bezier points */
        ct=t;
    }
    Barycentric(t,x,y,&u,&v,&w);
    *z=B_Interpolation2(t,u,v,w,BzPoints);
    return 1;
}

/* Compute the barycentric cordinates */
Barycentric(t,x,y,u,v,w)
struct triangle *t;
float x,y;
float *u,*v,*w;
{
    float x0,y0,x1,y1,x2,y2;
    float tarea,t1;
    float Area[3];

    /* compute the three bycentric triangle cordinates */
x0=Point[t->P[0]].x; y0=Point[t->P[0]].y;
x1=Point[t->P[1]].x; y1=Point[t->P[1]].y;
x2=Point[t->P[2]].x; y2=Point[t->P[2]].y;
u = x \cdot (y_1 - y_2) - x_1 \cdot (y - y_2) + x_2 \cdot (y - y_1);
v = x_0 \cdot (y - y_2) - x \cdot (y_0 - y_2) + x_2 \cdot (y_0 - y);
w = x_0 \cdot (y_1 - y) - x_1 \cdot (y_0 - y) + x \cdot (y_0 - y_1);
tarea = x_0 \cdot (y_1 - y_2) - x_1 \cdot (y_0 - y_2) + x_2 \cdot (y_0 - y_1);
\* \* = u / \text{tarea}; \* v = v / \text{tarea}; \* w = w / \text{tarea};

```c
/* compute Bezier points */
TriangleBzPoints(t, BzPoints, nx, ny, nz)
struct triangle *t;
float BzPoints[6][6], nx[], ny[], nz[];
/* BePoints[u/v/w] */
{
    int i, j;
    float t1, tx[3], ty[3], tz[3], ttt[3];
    float tmp[4], x, y, z;
    struct triangle *curr;
    for (i = 0; i < 3; i++) {
        tx[i] = Point[t->P[i]].x;
        ty[i] = Point[t->P[i]].y;
        tz[i] = Point[t->P[i]].z;
    }
    /* Border line u-v */
    BzPoints[0][0] = tz[0];
    x = tx[0] + 1.0/5.0*(tx[1] - tx[0]);
    y = ty[0] + 1.0/5.0*(ty[1] - ty[0]);
    BzPoints[0][1] = PlaneValue(nx[0], ny[0], nz[0], tx[0], ty[0], tz[0], x, y);

    x = tx[0] + 2.0/5.0*(tx[1] - tx[0]);
    y = ty[0] + 2.0/5.0*(ty[1] - ty[0]);
    BzPoints[0][2] = PlaneValue(nx[0], ny[0], nz[0], tx[0], ty[0], tz[0], x, y);

    x = tx[0] + 3.0/5.0*(tx[1] - tx[0]);
    y = ty[0] + 3.0/5.0*(ty[1] - ty[0]);
    BzPoints[0][3] = PlaneValue(nx[0], ny[0], nz[0], tx[0], ty[0], tz[0], x, y);

    x = tx[0] + 4.0/5.0*(tx[1] - tx[0]);
    y = ty[0] + 4.0/5.0*(ty[1] - ty[0]);
    BzPoints[0][4] = PlaneValue(nx[0], ny[0], nz[0], tx[0], ty[0], tz[0], x, y);
    BzPoints[0][5] = tz[1];
    /* Border line u-w */
    x = tx[0] + 1.0/5.0*(tx[2] - tx[0]);
    y = ty[0] + 1.0/5.0*(ty[2] - ty[0]);
    BzPoints[1][0] = PlaneValue(nx[0], ny[0], nz[0], tx[0], ty[0], tz[0], x, y);

    x = tx[0] + 2.0/5.0*(tx[2] - tx[0]);
    y = ty[0] + 2.0/5.0*(ty[2] - ty[0]);
    BzPoints[1][1] = PlaneValue(nx[0], ny[0], nz[0], tx[0], ty[0], tz[0], x, y);
```
\[x = tx[0] + 3.0/5.0*(tx[2] - tx[0]);
\]
\[y = ty[0] + 3.0/5.0*(ty[2] - ty[0]);
\]
\[BzPoints[3][0] = PlaneValue(nx[2], ny[2], nz[2], tx[2], ty[2], tz[2], x, y);\]

\[x = tx[0] + 4.0/5.0*(tx[2] - tx[0]);
\]
\[y = ty[0] + 4.0/5.0*(ty[2] - ty[0]);
\]
\[BzPoints[5][0] = tx[2];\]

\[/* \text{Border line w-v */}\]
\[x = tx[1] + 1.0/5.0*(tx[2] - tx[1]);
\]
\[y = ty[1] + 1.0/5.0*(ty[2] - ty[1]);
\]
\[BzPoints[1][4] = PlaneValue(nx[1], ny[1], nz[1], tx[1], ty[1], tz[1], x, y);\]

\[x = tx[1] + 2.0/5.0*(tx[2] - tx[1]);
\]
\[y = ty[1] + 2.0/5.0*(ty[2] - ty[1]);
\]
\[BzPoints[2][3] = PlaneValue(nx[2], ny[2], nz[2], tx[2], ty[2], tz[2], x, y);\]

\[x = tx[1] + 3.0/5.0*(tx[2] - tx[1]);
\]
\[y = ty[1] + 3.0/5.0*(ty[2] - ty[1]);
\]
\[BzPoints[3][2] = PlaneValue(nx[2], ny[2], nz[2], tx[2], ty[2], tz[2], x, y);\]

\[x = tx[1] + 4.0/5.0*(tx[2] - tx[1]);
\]
\[y = ty[1] + 4.0/5.0*(ty[2] - ty[1]);
\]
\[BzPoints[4][1] = PlaneValue(nx[2], ny[2], nz[2], tx[2], ty[2], tz[2], x, y);\]

\[/* \text{Interior points: first set */}\]
\[x = 0.6*tx[0] + 0.2*(tx[2] + tx[1]);
\]
\[y = 0.6*ty[0] + 0.2*(ty[2] + ty[1]);
\]
\[BzPoints[1][1] = PlaneValue(nx[0], ny[0], nz[0], tx[0], ty[0], tz[0], x, y);\]

\[x = 0.6*tx[1] + 0.2*(tx[2] + tx[0]);
\]
\[y = 0.6*ty[1] + 0.2*(ty[2] + ty[0]);
\]
\[BzPoints[1][3] = PlaneValue(nx[1], ny[1], nz[1], tx[1], ty[1], tz[1], x, y);\]

\[x = 0.6*tx[2] + 0.2*(tx[0] + tx[1]);
\]
\[y = 0.6*ty[2] + 0.2*(ty[0] + ty[1]);
\]
\[BzPoints[3][3] = PlaneValue(nx[2], ny[2], nz[2], tx[2], ty[2], tz[2], x, y);\]

\[/* \text{Interior points: second set */}\]
\[for(i = 0; i < 3; i++)\]
\[\text{InteriorPoint(t,i,nx,ny,nz,BzPoints);}\]

\[} \]

\[/* \text{Computer the interior point */}\]
\[\text{InteriorPoint(t,n,nx,ny,nz,BzPoints);}\]
\[\text{struct triangle *t;}\]
\[\text{float BzPoints[6][6], nx[], ny[], nz[];}\]
\[\text{int n;}\]
\[\{ \]
int i, j;
float nx3[3], ny3[3], nz3[3], nx4, ny4, nz4;
float x, y, z, x3, y3, z3, x4, y4, z4;
float tx[3], ty[3], tz[3];
float tx1[3], ty1[3], tz1[3];
struct triangle *curr;

for(i=0; i<3; i++) {
    tx[i]=Point[t->P[i]].x;
    ty[i]=Point[t->P[i]].y;
    tz[i]=Point[t->P[i]].z;
}

TriangleNormal(t, &nx3, &ny3, &nz3);

/* compute the interior point : first */
x=0.2*tx[(n+2)%3]+0.4*(tx[n]-tx[(n+1)%3]);
y=0.2*ty[(n+2)%3]+0.4*(ty[n]-ty[(n+1)%3]);
z=PlaneValue(nx[n], ny[n], nz[n], tx[n], ty[n], tz[n], x, y);
/* z+=0.4*PlaneValue(nx[n], ny[n], nz[n], tx[n], ty[n], tz[n], x, y);
z+=0.2*PlaneValue(nx[(n+2)%3], ny[(n+2)%3], nz[(n+2)%3],
    tx[(n+2)%3], ty[(n+2)%3], tz[(n+2)%3], x, y);
*/

/* line vector */
x4=(tx[(n+1)%3]-tx[n])*0.2;
y4=(ty[(n+1)%3]-ty[n])*0.2;
if(n==0) {
    z4=BzPoints[0][3]-BzPoints[0][2];
}
else if(n==1)
    z4=BzPoints[3][2]-BzPoints[2][3];
else
    z4=BzPoints[2][0]-BzPoints[3][0];

if((curr=t->E[n])!=NULL) {
    i=PointNumber(curr, t->P[n]);
    for(i=0; i<3; i++) {
        tx1[i]=Point[curr->P[i]].x;
        ty1[i]=Point[curr->P[i]].y;
        tz1[i]=Point[curr->P[i]].z;
    }
}

TriangleNormals(curr, nx3, ny3, nz3);
z3+=0.4*PlaneValue(nx3[i], ny3[i], nz3[i], tx1[i], ty1[i], tz1[i], x, y);
z3+=0.4*PlaneValue(nx3[(i+2)%3], ny3[(i+2)%3], nz3[(i+2)%3],
    tx1[(i+2)%3], ty1[(i+2)%3], tz1[(i+2)%3], x, y);
z3+=0.2*PlaneValue(nx3[(i+1)%3], ny3[(i+1)%3], nz3[(i+1)%3],
    tx1[(i+1)%3], ty1[(i+1)%3], tz1[(i+1)%3], x, y);
*/
x3 = Point[curr— >>P[(i+l)%3]].x;
y3 = Point[curr— >>P[(i+l)%3]].y;
z3 = Point[curr— >>P[(i+l)%3]].z;
x3 = (x3 — tx[(n+2)%3]);
y3 = (y3 — ty[(n+2)%3]);
z3 = (z3 — Point[t— >>P[(n+2)%3]].z);

/* get the plane normal */
x4 = y3*z4 — z3*y4;
y4 = z3*x4 — x3*z4;
z4 = x3*y4 — x4*y3;

x3 = 0.6*tx[n]+0.4*tx[(n+1)%3];
y3 = 0.6*ty[n]+0.4*ty[(n+1)%3];

if(n == 0)
  z3 = BzPoints[0][2];
else if(n == 1)
  z3 = BzPoints[2][3];
else z3 = BzPoints[3][0];

z = PlaneValue(nx4, ny4, nz4, x3, z3, x, y);
}
if(n == 0) BzPoints[1][2]=z;
else if(n == 1) BzPoints[2][2]=z;
else BzPoints[2][1]=z;

/* Compute the plane value Given the three normals and a point */
float PlaneValue(nx1, ny1, nz1, x0, y0, z0, x, y)
float nx1, ny1, nz1; /* Three normals */
float x0, y0, z0; /* point the plane passing by */
float x, y; /* The point to be evaluated */
{
  float t1;

  t1 = (nx1*(x— x0)+ny1*(y— y0))/nz1;
  t1 = — t1+x0;
  return(t1);
}

% phardy.c

/* file : phardy, piecewise hardy interpolation*/
/* interpolation with piecewise Hardy method */
/* change piece hardy’s method to make it solve all
the equations. Oct. 30, 1992 */
/* Weibao Wu, UNLV */

#define NSPMAX 50 /* Max. # of sample points */
#define MAXPOINTS 1000
#include <stdio.h>
#include <math.h>
#include "image.h"
#include "globals.h"

/* define structure for each data point */
/* it contains #points around this point, the Hardy constant, and
   a point to the structure of the solution */
static struct phs {int npoints; float cnst;
    struct solution *slv;} *PHS[MAXPOINTS];
struct solution {int point; float X};

#define sqr(x) (x)*(x)
/* point structure, use array, the size is decided by the # of points */
float Rconstant=0.001;
float H_Interpolation2();
float Distance();
float Dist2();
float Dist3();
int OnBoundary(struct triangle *t);
int Hardy_InterpPoint(float x, float y, float *z);
int Hardy_InterpPoint2(struct triangle *t, float x, float y, float *z);

/* Interpolation the data in the array Point[TotalPoints] in ROI */
void Hardy_Interp(x1,y1,x2,y2, deltax, deltay, OutPutPoint)
    float x1,y1,x2,y2;
    float deltax, deltay;
    void (*OutPutPoint)(int, int, float);
{
    int i,j;
    int l,m;
    float x,y,z;

    struct triangle *t;
    if(x1==x2 || y1==y2)
        return;
    if(x1>x2) {
        x=x1; x1=x2; x2=x;
    }
    if(y1>y2) {
        y=y1; y1=y2; y2=y;
    }
    Traversal(Triangle,0); /* Set visit sign=0 */
    /* solve piecewise hardy once for all */
    SolveHardy();
    Traversal(Triangle,0); /* Set visit sign=0 */
    for(i=0; i<FramePointX; i++)
        for(j=0; j<FramePointY; j++)
            if(Hardy_InterpPoint( x, y, &z)) /* interpolated */
                (*OutPutPoint)(j, i, z);
I solve piecewise hardy system of equation for all the points */

SolveHardyQ { int i, visit=1;
extern struct triangle *Triangle;
struct triangle *t;
/* allocate space for each point */
/* PHS= (struct phs *)malloc((long)TotalPoints*sizeof(struct *phs)); */
/* initial the solution pointer */
if(TotalPoints>MAXPOINTS) {
    fprintf(stdout, "Error: too many points\n"); exit(-1);
}
for(i=0; i<TotalPoints; i++)
    PHS[i]=NULL;
/* traverse the triangle net and solve each equation */
/* push the first triangle in queue */
InitQueue();
EnQueue(Triangle);
do {
    OutQueue(&t);
t— ►visit=visit;
SolveHardyTriangle(t); /* solve Hardy equation in a triangle */
    for(i=0; i<3; i++)
        /* if edge has neighbour and has not been visited */
        if((t— »E[i]¬=NULL) && (t— »E[i])¬ »visity¬=visit )
            EnQueue(t— »E[i]);
} while(EmptyQueue()¬=1);

/* solve Hardy equation in a triangle */
SolveHardyTriangle(t)
struct triangle *t;
{ int i,j, l, m, n;
int NSP, ipoint;
int P[3][NSPMAX], NSPO[3];
float X[3][NSPMAX], R[3];
struct triangle *curr;
struct boundary *bnd;
float *A[NSPMAX], B[NSPMAX];
float *ptr, R2;
}
for(i=0; i<3; i++) /* get the patch equation */
if(PHS[(t— »P[i])¬=NULL] { /* if this point is not solved */
curr=t;
for(j=0; j<3; j++)
P[i][j]=curr->P[i];
NSP=3;
do {
    n=PointNumber(curr,t->P[i]);
    if(curr->E[(n+2)%3]==NULL) {
        FindBound(t->P[i],&bnd);
        curr=bnd->t;
        /* add the boundary point*/
        n=PointNumber(curr,t->P[i]);
        P[i][NSP++]=curr->P[(n+1)%3];
    } else
        curr=curr->E[(n+2)%3];
    if(curr!=t) {
        n=PointNumber(curr,t->P[i]);
        P[i][NSP++]=curr->P[(n+1)%3];
    }
    if(NSP>=NSPMAX) error("NSP too large");
} while(curr!=t);
NSP--;
if(NSP>=NSPMAX) error("Matrix too large");
ptr=(float *)malloc((long)NSP*(NSP+1)*sizeof(float));
if(ptr==NULL) error("Allocate space for Matrix A");
FindConstantR(P[i],NSP,&R[i]);
for(j=0;j<NSP;j++)
    /* allocate space for matrix A */
    A[j]=ptr+j*(NSP+1);
for(l=0;l<NSP;l++)
    for(m=0;m<NSP;m++)
        A[l][m]=Dist2(P[i],R[i],l,m);
for(j=0;j<NSP;j++)
    B[j]=Point[P[i][j]].z;
SolveEquation(A,B,X[i],NSP); /* Solve the equation A*X=B */
NSPO[i]=NSP;
free(ptr);
/* add the solution to the structure */
AddSolution(t->P[i], NSP, X[i], P[i], R[i]);
}

/* add the solution to the point structure */
AddSolution(point, NSP, X, P, R)
int point, NSP, *P;
float *X, R;
{
    int i;
    struct solution *tmp;
    /* alloc memory for */
    if((PHS[point]=(struct phs *)malloc((long)sizeof(struct phs)))==NULL){
        fprintf(stdout, "error allocate space for PHS
");exit(-1);
85

PHS[point]→npoints= NSP;
PHS[point]→cnst=R;
if((tmp=PHS[point]→slv=(struct solution *)
malloc((long)NSP*sizeof(struct solution)))==NULL)
    fprintf(stdout, "error allocate space for solution\n"); exit(-1);
}
for(i=0; i<NSP; i++)
    tmp[i].point= P[i];
    tmp[i].X= X[i];
}

/* Interpolate one point at (x,y) */
/* result is z */
int Hardy_ModInterpPoint(float x,float y,float *z)
{
    int l;
    struct triangle *t;
    static int P[3][NSPMAX],NSP[3];
    static float X[3][NSPMAX],R[3];
    static struct triangle *ct=NULL;
    l=InTriangle(&t,x,y);
    if(l<0)
        return 0; /* do not interpolate */
    if(ct!=t) { /* new triangle */
        RetrieveHardy(t,P,X,NSP,R);
        ct=t;
    }
    H_ModInterpPoint2(t,x,y,z,P,X,NSP,R);
    return(1);
}

/* Interpolate one point of triangle t at (x,y) */
/* result is z */
int Hardy_ModInterpPoint2(struct triangle *t, float x, float y, float *z)
{
    static int P[3][NSPMAX],NSP[3];
    static float X[3][NSPMAX],R[3];
    static struct triangle *ct=NULL;
    if(ct!=t) { /* new triangle */
        RetrieveHardy(t,P,X,NSP,R);
        ct=t;
    }
    H_ModInterpPoint2(t,x,y,z,P,X,NSP,R);
    return(1);
}

/* retrieve hardy solutions */
RetrieveHardy(t,P,X,NSP,R)
struct triangle *t;
int P[3][NSPMAX], NSPO[3];
float X[3][NSPMAX], R[3];
{
    int i, j;
    struct solution *tmp;
    for(i=0; i<3; i++) {
        tmp = PHS[t->P[i]]—t—slv;
        R[i] = PHS[t->P[i]]—cnst;
        NSP0[i] = PHS[t->P[i]]—npoints;
        for(j=0; j<NSP0[i]; j++) {
            X[i][j] = tmp[j].X;
            P[i][j] = tmp[j].point;
        }
    }
}

/* Interpolate at (x,y), result=z */
/* P: point number.
   X: solution for the hardy equation */
HInterpPoint2(t,x,y,z,P,X,NSP,R)
struct triangle *t;
float x,y, *z,X[3][NSPMAX], R[3];
int P[3][NSPMAX], NSP[3];
{
    int ij;
    float Area[3], tz[3], z2;
    float w[3];   /* Weight */
    TriangleAreas(t,x,y,Area);
    for(i=0;i<3;i++)
        tz[i] = 0;
    for(i=0;i<3;i++) {   /* Compute the value for three patches */
        for(j=0; j<NSP[i]; j++)
            tz[i] += X[i][j]*Dist3(P[i], R[i], x, y, j);
    }
    Weight(t,x,y,Area,w);
    *z = tz[0]*w[0] + tz[1]*w[1] + tz[2]*w[2];
}

/* FindConstantR: Find the Hardy constant R */
/* R=0.815*The shortest Distance */
FindConstantR(P,NSP,R)
int P[], NSP;
float *R;
{
    int l, m;
    float t1, maxdist, TotalDist;
    float x0, y0, x1, y1, z0, z1;
    float zmax, zmin;

z0=0;
zmin=zmax=Point[P[0]].z;
for(l=0;l<NSP;l++) {
    zl=Point[P[l]].z; /* Average of Z */
    z0+=zl;
    if(zmax<zl) zmax=zl;
    if(zmin>zl) zmin=zl;
}
z0=z0/NSP;
z1=0;
for(l=0;l<NSP;l++)
    zl+=(z0— Point[P[l]].z)*(z0—Point[P[l]].z);
z1=sqrt(zl/NSP);

x0=Point[P[0]].x; y0=Point[P[0]].y;
x1=Point[P[1]].x; y1=Point[P[1]].y;
maxdist=0.0; TotalDist=0.0;
for(l=0;l<NSP;l++) {
    x0=Point[P[l]].x;
y0=Point[P[l]].y;
    for(m=l+1;m<NSP;m++) {
        x1=Point[P[m]].x;
y1=Point[P[m]].y;
        /*
        tl=Distance(x0,y0,x1,y1); */
        tl=(x0—xl)*(x0—xl)+(y0—yl)*(y0—yl);
        TotalDist+=tl;
        if(maxdist<tl) maxdist=tl;
    }
}
*R=Rconstant*maxdist;

/* Compute Weight */
Weight(t,x,y,Area,w)
struct triangle *t;
float Area[3],w[3];
float x,y;
{
    int i;
    float t1;
    float tx[3],ty[3];

    for(i=0;i<3;i++) {
        tx[i]=Point[t—»P[i]].x;
yt[i]=Point[t—P[i]].y;
    }

    /* Compute Weight */
    for(i=0;i<3;i++) {
        w[i]=(sqr(Area[(i+2)%3])+sqr(Area[i]))*sqr(Area[(i+1)%3]);
\[ t_1 = \sqrt{(x-tx[i+1])^2 + (y-ty[i+1])^2}; \]
\[ t_1 = t_1 * \left( \sqrt{(x-tx[i+2])^2 + (y-ty[i+2])^2} \right); \]
\[ w[i] = w[i] * t_1; \]
\[ \] for (i = 0; i < 3; i++) {
    if (x == tx[i] && y == ty[i]) {
        w[i] = 1.0; w[(i+1)%3] = 0.0; w[(i+2)%3] = 0.0;
        break;
    }
} \]
\[ t_1 = w[0] + w[1] + w[2]; \] /* Normalize */
\[ \]
/* Calculate distance from p1 to p2 */
float Dist2(P, R, p1, p2)
int P[3];
float R;
int p1, p2;
{
    float x0, y0, x1, y1;
    float d;
    x0 = Point[P[p1]].x;
    y0 = Point[P[p1]].y;
    x1 = Point[P[p2]].x;
    y1 = Point[P[p2]].y;
    d = (x1 - x0)^2 + (y1 - y0)^2;
    d = sqrt(d + R);
    return(d);
} \]

float Dist3(P, R, x0, y0, p)
int P[3];
float R;
int p;
float x0, y0;
{
    float x1, y1;
    float d;
    x1 = Point[P[p]].x;
    y1 = Point[P[p]].y;
    d = (x1 - x0)^2 + (y1 - y0)^2;
    d = sqrt(d + R);
    return(d);
} \]

% ftriangle.c
/* ftriangle.c */
/* Weiibo Wu Oct.7, 1991
Deluynay Trianglation for surface interpolation */
#include <stdio.h>
#include <math.h>
#include "image.h"

/* static functions */
static EmptyStack();
static InitStack();
static Push();
static Pop();
void DrawTriangles(void (*out_put_line)(POINT *, POINT *));

/* point structure, use array, the size is decided by the # of points */
extern struct point *Point;
extern int TotalPoints;
extern float wmx,wnx,wmy,wny;
extern struct triangle *Triangle;
extern struct boundary *Boundary;

struct stack {int Edge;struct triangle *t0; struct triangle *t1;
  struct stack *prev;
} stack;
struct stack *Stack;
struct queue {struct triangle *t; struct queue *next;};
struct queue *head, *tail;
float Distance();

/* Trianglation the points */
Trianglation()
{
  int i,j,k;
  struct triangle *CurrTriangle;
  /* Get the first triangle */
  FirstTriangle();
  /* Insert the following Points to the already exist triangle net work*/
  for(i=3;i>TotalPoints;i++)
    InsertPoint(i);
}

/* Tranverse the trianglated net, visit only once for each point */
/* Broad first */
Tranverse(t,visit)
struct triangle *t;
int visit;
{
  int i,j;
  /* push the first triangle in queue */
  InitQueue();
  EnQueue(t);
do {
OutQueue(&t);
t->visit=visit;
for(i=0;i<3;i++) {
    /* if edge has neighbour and has not been visited */
    if((t->E[i]!NULL) & & (t->E[i])— visit ^visit )
        EnQueue(t— >E[i]);
}
} while(EmptyQueue()!1);
}

void DrawTriangles(void (*out_put_line)(POINT *, POINT *))
{
int i, j;
struct triangle *t;
/* push the first triangle in queue */
Tranverse(Triangle, 0);
InitQueue();
EnQueue(Triangle);
do {
    OutQueue(&t);
t— > visit =1;
    out_put_line(&Point[t— >P[0]],&Point[t— >P[1]]);
    out_put_line(&Point[t— >P[0]],&Point[t— >P[2]]);
    out_put_line(&Point[t— >P[1]],&Point[t— >P[2]]);
    for(i=0;i<3;i++) {
        /* if edge has neighbour and has not been visited */
        if((t->E[i]!NULL) & & (t— >E[i])— visit !1 )
            EnQueue(t— >E[i]);
    }
} while(EmptyQueue()!1);
Tranverse(Triangle, 0);
/* UnmarkedTriangle: to see if there is an triangle has not been 
marked as visited , and return 1, if is, else return 0 */
/* the visit filed has two bits TRAVELED CONTOUR, TRAVELED=2, CONTOUR=1 */
/* TRAVELED is signed is visited, CONTOUR is signed if Traced. */
int UnmarkedTriangle(tt,Contour, Visit )
{
    int i, j, mask;
    struct triangle *t;
    mask=~Visit;
    SetVisitMask(mask, Visit);
    /* set all as not visited, do not change CONTOUR bit */
    /* push the first triangle in queue */
t=Triangle;
    InitQueue();
    EnQueue(t);
do {
    OutQueue(&t);
    i=(t->visit)&Contour;
    if(((t->visit)&Contour)==0) { /* contour not set yet */
        *tt=t;
        return(1);
    }
    t->visit=t->visit|Visit;
    for(i=0;i<3;i++) {
        /* if edge has neigbour and has not been visited */
        if((t->E[i]!=NULL) & & ((t->E[i])->visit&Visit)!=Visit)
            EnQueue(t->E[i]);
    }
} while(EmptyQueue()! =1);
return(0);
}

/* AND visit bits: Logic AND the visit bit */
/* input : AndMask */
SetVisitMask(visitmask, Visit)
int visitmask;
int Visit;
{
    int i, j;
    struct triangle *t;
    t=Triangle;
    /* push the first triangle in queue */
    InitQueue();
    EnQueue(t);
    do {
        OutQueue(&t);
        t->visit=t->visit&visitmask;
        for(i=0;i<3;i++) {
            /* if edge has neigbour and has not been visited */
            if((t->E[i]!=NULL) & & ((t->E[i])->visit&Visit)!=0)
                EnQueue(t->E[i]);
        }
    } while(EmptyQueue()! =1);
}

/* Find the unvisited triangle */
/* input : t: triangle. */
/* output: triangle */

/* Find the points in the zoon of influence */
/* Input : */
t : triangle.
x0, y0 : Coordinates of the point.
Radius : zoon of influence
CorPoints: index of points in the Radius.
Rpoints : # points in the CorPoints array.
findCorPoints(t, Radius, xO, yO, CorPoints, Rpoints)
int CorPoints[], *Rpoints;
float Radius, xO, yO;
struct triangle *t;
{
    int i, j, k, Rtotal;
    float x1, y1, dist;
    struct triangle *tmp;
    Rtotal=0;
    InitQueue();
    for(i=0;i<3;i++) {
        x1=Point[t->P[i]].x; y1=Point[t->P[i]].y;
        dist=Distance(xO, yO, x1, y1);
        if(dist<Radius) {
            CorPoints[Rtotal++]=t->P[i]; Point[t->P[i]].visit=1;
        }
    }
    EnQueue(t);
    do {
        OutQueue(&t);
        for(i=0;i<3;i++)
            if((tmp=t->E[i])!=NULL)
            {
                k=PointNumber(t->E[i], t->P[i]);
                x1=Point[tmp->P[(k+1)%3]].x; y1=Point[tmp->P[(k+1)%3]].y;
                dist=Distance(xO, yO, x1, y1);
                if(Point[tmp->P[(k+1)%3]].visit!=1 && (dist<Radius)) { /* not visited */
                    Point[tmp->P[(k+1)%3]].visit=1;
                    CorPoints[Rtotal++]=tmp->P[(k+1)%3];
                    EnQueue(tmp);
                }
            }
        } while(EmptyQueue()!=1);
    } while(EmptyQueue()^1);
    *Rpoints=Rtotal;
    for(i=0;i<Rtotal; i++) /* mark points as not visited */
        Point[CorPoints[i]].visit=0;
}

/* Insert point n in Point[n] to the already exist Triangle net work */
insertPoint(n)
int n;
{
    struct triangle *t,*t0,*t1,*t2,*t3,*tmp;
    struct boundary *curr,*curr1,*curr2;
    float x,y,x1,y1,x2,y2;
    int i,j,k,last;
    int Edge;
    x=Point[n].x; y=Point[n].y;
    /* printf("%d",n); */
    /* wpoint(n,RED); */
    /* getch(); */
}
/* Find the triangle which enclose point P, if P is out side the current, then return -1, if P is on the boundary of a point, return 0 else return 1 */
i=InTriangle(&t,x,y);
InitStack();
if(i==0) { /* the point is on the triangle */
SplitTriangle2(t1,&t0,&t1,&t2,&t3,n);
Edge=1; Triangle=t0;
if(t0^NULL & & t0==E[Edge]^NULL)
    Push(Edge,t0,t0==E[Edge]);
if(t1^NULL & & t1==E[Edge]^NULL)
    Push(Edge,t1,t1==E[Edge]);
if(t2^NULL & & t2==E[Edge]^NULL)
    Push(Edge,t2,t2==E[Edge]);
if(t3^NULL & & t3==E[Edge]^NULL)
    Push(Edge,t3,t3==E[Edge]);
}
else if(i==1) { /* the point is in the triangle t*/
/* split t into three, point P as the center of the new triangles */
SplitTriangle(t1,&t0,&t1,&t2,n);
/* Set new Triangle start point */
Edge=1;
Triangle=t0;
if(t0==E[Edge]^NULL)
    Push(Edge,t0,t0==E[Edge]);
if(t1==E[Edge]^NULL)
    Push(Edge,t1,t1==E[Edge]);
if(t2==E[Edge]^NULL)
    Push(Edge,t2,t2==E[Edge]);
}
else if (i==−1) { /* the point is out side the existing triangle */
OutBoundary(n);
}
/* Do the swap using stack */
while(EmptyStack()!=1) {
    Pop(&Edge,&t1,&t2);
t1==visit==−1; t2==visit==−1;
if(Swap(Edge,&t1,&t2)) { /* if the quadrilateral cause the edge swap */
    Triangle=t1;
    /* Push the two edges not incident to point P */
    if(t1==E[0]^NULL)
        Push(0,t1,t1==E[0]);
    if(t1==E[2]^NULL)
        Push(2,t1,t1==E[2]);
    if(t2==E[0]^NULL)
        Push(0,t2,t2==E[0]);
    if(t2==E[2]^NULL)
        Push(2,t2,t2==E[2]);
}
/* for points outside the triangles boundary */
OutBoundary(n)
{
    struct triangle *tmp,*tmp2,*tmp1;
    struct boundary *curr,*curr1,*curr2;
    struct boundary *tmpbndl,*tmpbnd2,*tmpbnd;
    float x,y,x1,y1,x2,y2;
    int i,j,k,last;
    int Edge;
    x=Point[n].x; y=Point[n].y;
    /* find all the points which are visible from point P */
    /* the boundary order is anticlockwise */
    curr=Boundary;
    /* find the transient edges which make point n first in left
    and then in right */
    x1=Point[curr->P[0]].x; y1=Point[curr->P[0]].y;
    x2=Point[curr->P[1]].x; y2=Point[curr->P[1]].y;
    i=InLeft(x,y,x1,y1,x2,y2);
    do{
        last=i;
        curr1=curr;
        curr=curr->next;
        x1=Point[curr->P[0]].x; y1=Point[curr->P[0]].y;
        x2=Point[curr->P[1]].x; y2=Point[curr->P[1]].y;
        i=InLeft(x,y,x1,y1,x2,y2);
    } while(last<0 || i>0);
    /* find the transient edges which make point n first in the left */
    curr=curr1;
    do{
        curr=curr->next;
        x1=Point[curr->P[0]].x; y1=Point[curr->P[0]].y;
        x2=Point[curr->P[1]].x; y2=Point[curr->P[1]].y;
        i=InLeft(x,y,x1,y1,x2,y2);
    } while(i<0);
    /* curr1 and curr2 are the two boundary edges */
    curr2=curr;
    tmp=(struct triangle *)malloc((long)sizeof(struct triangle));
    tmpbndl=(struct boundary *)malloc((long)sizeof(struct boundary));
    tmpbnd2=(struct boundary *)malloc((long)sizeof(struct boundary));
    if(tmp==NULL) error("Error: Allocate Space");
    if(tmpbndl==NULL) error("Error: Allocate Space");
    if(tmpbnd2==NULL) error("Error: Allocate Space");
    tmp1=NULL;
    /* for the temporary new boundary */
    tmpbndl->P[0]=curr1->P[0];
    tmpbndl->P[1]=curr1->P[1];
    tmpbndl->P[2]=n;
    tmpbndl->t=tmp;
/* tmp1 tmp tmp2 */
/* curr1 curr curr2 */
curr=curr1—next;
do {
  /* Push all the boundary edges between curr and curr1 into stack */
  /* create new triangle tmp */
tmp2=(struct triangle *)malloc((long)sizeof(struct triangle));
  if(tmp2==NULL) error("Error: Allocate Space ");
  i=PointNumber(curr—t,curr—P[0]);
  (curr—t)—E[i]=tmp;
  if(curr—t!=NULL)
    Push(1,tmp,curr—t);
  tmp1=tmp;
  tmp=tmp2;
  curr=curr—next;
  /* wline(tmp1—>P[0],tmp1—>P[1],WHITE); 
   wline(tmp1—>P[0],tmp1—>P[2], WHITE); */
} while(curr!=curr2);
free(tmp2);
tmp1—E[0]=NULL;
/* for the boundary */
tmpbnd2—P[1]=curr2—P[0];
tmpbnd2—P[0]=n;
tmpbnd2—t=tmp1;
/* for the boundary */
/* free all the boundary between curr1 and curr2 */
curr=curr1—next;
while(curr!=curr2) {
  tmpbnd=curr—next;
  free(curr);
  curr=tmpbnd;
}

Boundary=curr1;
/* Add two new edges */
curr1—next=tmpbnd1;
tmpbnd1—next=tmpbnd2;
tmpbnd2—next=curr2;
}

/* Find the triangle which enclose point P, if P is out side the 
current boundary, then return -1, if P is on the boundary, return 0 
else if p is in the boundary return 1 */
int InTriangle(t22,x,y)
float x,y;
struct triangle **t22;
{
  int i,j,k;
}
struct triangle *Curr;
float x1,y1,x2,y2;
float t1,t2,t0;
Curr=Triangle;
while(1) {
    /* If P is in the Current Triangle */
    x1=Point[Curr— *P[0]].x; y1=Point[Curr— *P[0]].y;
    x2=Point[Curr— *P[1]].x; y2=Point[Curr— *P[1]].y;
    i=InLeft(x,y,x1,y1,x2,y2);
    x1=Point[Curr— *P[1]].x; y1=Point[Curr— *P[1]].y;
    x2=Point[Curr— *P[2]].x; y2=Point[Curr— *P[2]].y;
    j=InLeft(x,y,x1,y1,x2,y2);
    x1=Point[Curr— *P[2]].x; y1=Point[Curr— *P[2]].y;
    x2=Point[Curr— *P[0]].x; y2=Point[Curr— *P[0]].y;
    k=InLeft(x,y,x1,y1,x2,y2);

    /* P is in the triangle */
    if(i==1 && j==1 && k==1) {
        *t22=Curr;
        Triangle=Curr;
        return(1);
    }

    /* if P is on one or two edges of the triangle */
    if(i>=0 && k>=0 && j>=0) {
        *t22=Curr;
        Triangle=Curr;
        return(0);
    }

    /* if P is outside the boundary */
    /* if one of the edge is the boundary and point P is outside the edge
      then point p is outside the boundary */
    if(Curr— E[0]==NULL && i==−1) {
        *t22=Curr;
        Triangle=Curr;
        return(−1);
    }
    if(Curr— E[1]==NULL && j==−1) {
        *t22=Curr;
        Triangle=Curr;
        return(−1);
    }
    if(Curr— E[2]==NULL && k==−1) {
        *t22=Curr;
        Triangle=Curr;
        return(−1);
    }
    if(i==−1)
        Curr=Curr— E[0];
    else if(j==−1)
        Curr=Curr— E[1];
    else if(k==−1)
        Curr=Curr— E[2];
}
Curr=Cur->E[1];
else /* k==l */
  Cur=Cur->E[2];
/* go back the while loop */
}

/* Get the first Triangle with anti.clock wise point order */
/* Initialize the Boundary */
FirstTriangle()
{
  int i,j,k;
  float x,y;
  struct triangle *Curr;
  struct boundary *curr2,*curr3;
  Triangle=(struct triangle *)malloc((long)sizeof(struct triangle));
  if(Triangle==NULL) error("Error: Alloc space");
  Boundary=(struct boundary *)malloc((long)sizeof(struct boundary));
  if(Boundary==NULL) error("Error: allocate Space");
  Triangle->P[0]=1; Triangle->P[1]=1; Triangle->P[2]=1;
  Cur=Triangle;
  x=(Point[0].x+Point[1].x+Point[2].x)/3.0;
  y=(Point[0].y+Point[1].y+Point[2].y)/3.0;
  /* x y must in the triangle by P0,P1,P2 */
  /*Determine the order of the three points, let P0 be the first point */
  Cur->P[0]=0;
  /* If (x,y) in the left side of line P0-P1, then P1 is the second point */
  i=InLeft(x,y,Point[0].x,Point[0].y,Point[1].x,Point[1].y);
  j=InLeft(x,y,Point[1].x,Point[1].y,Point[2].x,Point[2].y);
  k=InLeft(x,y,Point[2].x,Point[2].y,Point[0].x,Point[0].y);
  if(i>0 && j>0 && k>0) {
    Cur->P[1]=1; Cur->P[2]=2;
  }
  else 
    Cur->P[1]=2; Cur->P[2]=1;
}
/* Initialize the boundary , Boundary is a cylic linked list*/
curr2=(struct boundary *)malloc((long)sizeof(struct boundary));
if(curr2==NULL) error("Error: allocate Space");
curr3=(struct boundary *)malloc((long)sizeof(struct boundary));
if(curr3==NULL) error("Error: allocate Space");
Boundary->P[0]=Triangle->P[0];
Boundary->P[1]=Triangle->P[1];
Boundary->t=Triangle;
Boundary->next=curr2;
curr2->P[0]=Triangle->P[1];
curr2->P[1]=Triangle->P[2];
curr2->t=Triangle;
curr2->next=curr3;
curr3->P[0]=Triangle->P[2];
curr3— P[1]=Triangle— P[0];
curr3— t=Triangle;
curr3— next=Boundary;

/* Draw the triangle */
wline(Triangle->P[0], Triangle->P[1], WHITE);
wline(Triangle->P[1], Triangle->P[2], WHITE);
wline(Triangle->P[2], Triangle->P[0], WHITE); */

/* Determine if point (x,y) is in the left side of the line (x1,y1)-(x2,y2)
if YES, return 1, if on the line then return 1, else return -1 */
InLeft(x,y,x1,x2,y1)
float x,y,x1,x2,y1;
{
    float t;
    t=(x2—x1)*(y—y1)—(y2—y1)*(x—x1);
    if(t>0) return 1;
    else if(t==0) return 0;
    else return —1;
}

/* Split triangle t into three t0 t1 t2 triangles, Point n will be the center
of the new triangles */
SplitTriangle(t,t0,t1,t2,t2,n)
struct triangle *t,**t0,**t1,**t2;
{
    int i;
    struct triangle *Curr,*t0,*t1,*t2;
    struct boundary *bnd;
    /* allocate space for new triangles */
t0=(struct triangle *)malloc((long)sizeof(struct triangle));
if(t0==NULL) error("Error: Alloc Space");
t1=(struct triangle *)malloc((long)sizeof(struct triangle));
if(t1==NULL) error("Error: Alloc Space");
t2=(struct triangle *)malloc((long)sizeof(struct triangle));
if(t2==NULL) error("Error: Alloc Space");

/* for the three new triangles */
t0->P[0]=n; t0->P[1]=t->P[0]; t0->P[2]=t->P[1];
t1->P[0]=n; t1->P[1]=t->P[1]; t1->P[2]=t->P[2];
t2->P[0]=n; t2->P[1]=t->P[2]; t2->P[2]=t->P[0];
t0->E[0]=t2; t0->E[1]=t->E[0]; t0->E[2]=t1;
t1->E[0]=t0; t1->E[1]=t->E[1]; t1->E[2]=t2;
t2->E[0]=t1; t2->E[1]=t->E[2]; t2->E[2]=t0;
/* wline(n,t->P[0],BLUE);
wline(n,t->P[1],BLUE);
wline(n,t->P[2],BLUE); */
/* change the neighbour relation */
Curr=t->E[0];
if(Curr!=NULL) {
    i=PointNumber(Curr,t->P[1]);
    Curr->E[i]=t0;
else { /* Change Boundary */
    FindBound(t->P[0],&bnd);
    /* find the boundary pointer with his first point being curr1->P[i] */
    bnd->t=t0;
}

Curr=t->E[1];
if(Curr!=NULL){
    i=PointNumber(Curr,t->P[2]);
    Curr->E[i]=t1;
}
else { /* Change Boundary */
    FindBound(t->P[1],&bnd);
    /* find the boundary pointer with his first point being curr1->P[i] */
    bnd->t=t1;
}

Curr=t->E[2];
if(Curr!=NULL){
    i=PointNumber(Curr,t->P[0]);
    Curr->E[i]=t2;
}
else { /* Change Boundary */
    FindBound(t->P[2],&bnd);
    /* find the boundary pointer with his first point being curr1->P[i] */
    bnd->t=t2;
}

*t00=t0; *t11=t1; *t22=t2;
free(t);
}

/*Split triangle t into four t0 t1 t2 t3 triangles, Point n will be the center of the new triangles, and point n is on the edge of t */
SplitTriangle2(t,t00,t11,t22,t33,n)
struct triangle *t,**t00,**t11,**t22,**t33;
{
    int i,j,ji;
    float x,y,x1,y1,x2,y2;
    struct triangle *Curr,*t0,*t1,*t2,*t3,*jt;
    struct boundary *bnd,*bndtmp;
    /* allocate space for new triangles */
    t0=(struct triangle *)malloc((long)sizeof(struct triangle));
    if(t0==NULL) error("Error: Alloc Space");
    t1=(struct triangle *)malloc((long)sizeof(struct triangle));
    if(t1==NULL) error("Error: Alloc Space");
    t2=(struct triangle *)malloc((long)sizeof(struct triangle));
    if(t2==NULL) error("Error: Alloc Space");
    t3=(struct triangle *)malloc((long)sizeof(struct triangle));
    if(t3==NULL) error("Error: Alloc Space");
    /* to see point n is in which triangle */
    for(i=0;i<3;i++) {
        x=Point[n].x; y=Point[n].y;
        }
\begin{verbatim}
x1=Point[t->P[i]].x; y1=Point[t->P[i]].y;
x2=Point[t->P[(i+1)%3]].x; y2=Point[t->P[(i+1)%3]].y;
j=InLeft(x,y,x1,y1,x2,y2);
if(j==0) break;
}
if(j!=0) error("Point is not on the edge of a triangle");
j=i;
jt=t->E[j];
if(jt!=NULL) {
    ji=PointNumber(jt,t->P[j]);
    t2=P[0]=n; t2->P[1]=jt->P[ji];
    t2->P[2]=jt->P[(ji+1)%3];
    t3=P[0]=n; t3->P[1]=jt->P[(ji+1)%3];
    t3->P[2]=jt->P[(ji+2)%3];
    t2->E[0]=t1; t2->E[1]=jt->E[ji];
    t2->E[2]=t3;
    t3->E[0]=t2; t3->E[1]=jt->E[(ji+1)%3];
    t3->E[2]=t0;
/* wline(n,jt->P[0],BLUE);*/
    wline(n,jt->P[1],BLUE);
    wline(n,jt->P[2],BLUE); /*
    free(t2); free(t3);
    t2=NULL; t3=NULL;
    */
    for the four new triangles */
    t0->P[0]=n; t0->P[1]=t->P[(j+1)%3];
    t0->P[2]=t->P[(j+2)%3];
    t1->P[0]=n; t1->P[1]=t->P[(j+2)%3];
    t1->P[2]=t->P[j];
    t0->E[0]=t3; t0->E[1]=t->E[(j+1)%3];
    t0->E[2]=t1;
    t1->E[0]=t0; t1->E[1]=t->E[(j+2)%3];
    t1->E[2]=t2;
/* wline(n,t->P[0],BLUE);*/
    wline(n,t->P[1],BLUE);
    wline(n,t->P[2],BLUE); /*
    /\ change the neighbour relation /*
    Curr=t->E[(j+1)%3];
    if(Curr!=NULL) {
        i=PointNumber(Curr,t->P[(j+2)%3]);
        Curr->E[i]=t0;
    }
    else { /* Change Boundary */
        FindBound(t->P[(j+1)%3],&bnd);
        /* find the boundary pointer with his first point being curr1->P[i] */
        bnd->t=t0;
    }
    Curr=t->E[(j+2)%3];
    if(Curr!=NULL) {
        i=PointNumber(Curr,t->P[j]);
        Curr->E[i]=t1;
    }
    else { /* Change Boundary */
        FindBound(t->P[(j+2)%3],&bnd);
        /* find the boundary pointer with his first point being curr1->P[i] */
        bnd->t=t1;
    }
\end{verbatim}
if(jt!=NULL) {
    Curr=jt->E[ji];
    if(Curr!=NULL) {
        i=PointNumber(Curr,jt->P[(ji+1)%3]);
        Curr->E[i]=t2;
    } else { /* Change Boundary */
        FindBound(jt->P[ji],&bnd); /* find the boundary pointer with his first point being curr->P[ji] */
        bnd->t=t2;
    }
    Curr=jt->E[(ji+1)%3];
    if(Curr!=NULL) {
        i=PointNumber(Curr,jt->P[(ji+2)%3]);
        Curr->E[i]=t3;
    } else {
        i=FindBound(jt->P[(ji+1)%3],&bnd);
        bnd->t=t3;
    }
} else { /*jt==NULL */
    i=FindBound(t->P[ji],&bnd);
    bndtmp=(struct boundary *)malloc((long)sizeof(struct boundary));
    if(bndtmp==NULL) error("Error: Alloc Space");
    bnd->t=t1; bnd->P[1]=n;
    bndtmp->t=t0; bndtmp->P[0]=n; bndtmp->P[1]=t->P[(j+i)%3];
    bndtmp->next=bnd->next;
    bnd->next=bndtmp;
}

*t00=t0; *t11=t1; *t22=t2; *t33=t3;
free(t);
if(t!=NULL) free(t);
/* Swap: to swap the edge if possible and form two new triangles */
Swap(Edge,t11,t22)
int Edge;
struct triangle **t11,**t22;
{
    int i,j,k,l;
    struct triangle *t1, *t2;
    t1=**t11; t2=**t22;
    /* if either of the triangle is NULL, return */
    if(t1==NULL || t2==NULL )
        return(0);
    /* First of all, decide if the Quadratic is concave, return */
    if(Concave(t1,t2,Edge))
        return(0);
    /* to apply the Circle criteria to see if the edge need to be swapped */
if PO of t1 is colinear with the P1 and P2 then not swap */
if(Criteria(t1,t2,Edge)==0) /* No need to swap */
    return(0);

/* swap the edge */
SwapEdge(t11,t22,Edge);
/* Swap */
return(1);
}

/* SwapEdge */
SwapEdge(t11,t22,Edge)
struct triangle ***t11,***t22;
{
    int i,j,k,l;
    struct triangle *t1, *t2;
    struct triangle *curr1, *curr2,*tmpptr;
    struct boundary *bnd;

t1=*t11; t2=*t22;
/* Check Stack, to take all the accurences of t1 and t2 triangles in the 
Stack,because we will change the triangle and the relations with the neighbour */
CheckStack(t1,t2);
i=PointNumber(t2,t1->P[Edge]);
i=(i+1)%3;
/* wline(t1->P[Edge],t1->P[(Edge+1)%3],BLACK); 
wline(t1->P[(Edge+2)%3],t2->P[i],YELLOW); */
curr1=(struct triangle *)malloc((long)sizeof(struct triangle));
if(curr1==NULL) error("Error: Alloc Space");
curr2=(struct triangle *)malloc((long)sizeof(struct triangle));
if(curr2==NULL) error("Error: Alloc Space");
curr1->P[0]=t1->P[Edge];
curr1->P[1]=t1->P[(Edge+2)%3];
curr1->P[2]=t1->P[(Edge+1)%3];
curr2->P[0]=t1->P[(Edge+2)%3];
curr2->P[1]=t1->P[(Edge+1)%3];
curr2->P[2]=t2->P[i];
curr1->E[0]=t2->E[(i+2)%3];
curr1->E[1]=curr2;
curr1->E[2]=t1->E[(Edge+2)%3];
curr2->E[0]=t1->E[(Edge+1)%3];
curr2->E[1]=curr1;
curr2->E[2]=t2->E[i];

if((tmpptr=curr1->E[0])!=NULL){
    j=PointNumber(tmpptr,curr1->P[1]);
    tmpptr->E[j]=curr1;
}
tmp=(y2−y1)∗(x0−x1)−(x1−x2)∗(y1−y0);
if(tmp==0) /* No swap */
    return(0);
/* Find the center of the circle */
/* Find the center of the circle */
mu=((y2−y1)∗(y10−y12)−(x10−x12)∗(x1−x2))/tmp;
xc=x10+mu∗(y0−y1);
yc=y10+mu∗(x1−x0);
/* the Radius of the circle */
r=Distance(xc,yc,x0,y0);
/* r1=Distance(xc,yc,x1,y1); 
r2=Distance(xc,yc,x2,y2); */
tmp=Distance(xc,yc,x3,y3);
if(tmp>r) /* No need to swap */
    return(0);
/* Swap the edge */
return(1);
}
/* if the Quadratic is concave */
Concave(t1,t2,Edge)
int Edge;
struct triangle *t1, *t2;
{
    int i,j,l,m,k;
    float x,y,x1,y1,x2,y2;

    i=PointNumber(t2,t1->P[Edge]);
    j=(i+1)%3;
    x=Point[t1->P[Edge]].x; y=Point[t1->P[Edge]].y;
    x1=Point[t1->P[(Edge+2)%3]].x; y1=Point[t1->P[(Edge+2)%3]].y;
    x2=Point[t2->P[j]].x; y2=Point[t2->P[j]].y;
    k=InLeft(x,y,x1,y1,x2,y2);
    l=InLeft(x,y,x1,y1,x2,y2);
    if((k>0 & & l>0)||(k<0 & & l<0))
        return(1);
    else return(0);
}
/* FindBound :Find the boundary pointer with his first point as n */
FindBound(n,b)
int n;
struct boundary **b;
{
    struct boundary *curr;
    curr=Boundary;
    while(curr->P[0]!=n) curr=curr->next;
    *b=curr;
if((tmp.ptr=currl->E[2])!=NULL){
    j=PointNumber(tmp.ptr,currl->P[0]);
    tmp.ptr->E[j]=currl;
}

if((tmp.ptr=curr2->E[0])!=NULL) {
    j=PointNumber(tmp.ptr,curr2->P[1]);
    tmp.ptr->E[j]=curr2;
}
if((tmp.ptr=curr2->E[2])!=NULL) {
    j=PointNumber(tmp.ptr,curr2->P[0]);
    tmp.ptr->E[j]=curr2;
}
free(t1); free(t2);
* t11=currl; *t22=curr2;
/* if curr1 and curr2 are boundary, then update them */
for(i=0;i<3;i++) {
    if(curr1->E[i]==NULL) { /* it is boundary */
        FindBound(curr1->P[i],&bnd);
        /* find the boundary pointer with his first point being curr1->P[i] */
        bnd->t=currl;
    }
    if(curr2->E[i]==NULL) {  /* it is boundary */
        FindBound(curr2->P[i],&bnd);
        /* find the boundary pointer with his first point being curr1->P[i] */
        bnd->t=curr2;
    }
}

/* Using circle criteria to see if t1 t2 need to swap 
   if swap, return 1, else 0 */
Criteria(t1,t2,Edge)
int Edge;
struct triangle * t1,*t2;
{
    int i;
    float x0,y0,x1,y1,x2,y2,x3,y3,x,y;
    float x12,y12,x10,y10,xc,yc;
    float mu,lambda,r;
    /* tmp=(x2-x1)(y-y1)-(y2-y1)*(x-x1); */
    x0=Point[t1->P[0]].x; y0=Point[t1->P[0]].y;
    x1=Point[t1->P[1]].x; y1=Point[t1->P[1]].y;
    x2=Point[t1->P[2]].x; y2=Point[t1->P[2]].y;
    i=PointNumber(t2,t1->P[Edge]);
    i=(i+1)%3;
    x3=Point[t2->P[i]].x; y3=Point[t2->P[i]].y;
    /* Calculate the center of the circle for points p0,p1,p2 */
    x12=(x1+x2)/2.0; y12=(y1+y2)/2.0;
    x10=(x1+x0)/2.0; y10=(y1+y0)/2.0;
/* Find the nearest triangle of point (x,y). (x,y) is a point outside the triangle mesh */

FindNearestTriangle(x,y,t22)
float x,y;
struct triangle **t22;
{
    int i,l,n,i1,i2;
    int finish=0;
    struct triangle *t;
    float x0,y0,x1,y1,x2,y2;
    struct boundary *curr;
    curr=Boundary;
    while(!finish) {
        x0=Point[curr— *P[0]].x; y0=Point[curr— +P[0]].y;
        x1=Point[curr— +P[l]].x; y1=Point[curr— »P[l]].y;
        l=InLeft(x,y,x0,y0,x1,y1);
        if(l<0) { /* The point is not in the triangle */
            t=curr— * t;
            n=PointNumber(t,curr— *P[1]);
            x2=Point[t— →P[(n+l)%3]].x; y2=Point[t— →P[(n+l)%3]].y;
            i1=InLeft(x,y,x1,y1,x2,y2);
            i2=InLeft(x,y,x2,y2,x0,y0);
            if(i1>0 && i2>0) {
                *t22=t; finish=l;
            }
            else curr=curr— >next;
        }
        else curr=curr— >next;
    }
}

/* PointNumber(t,n): find the point or edge number of triangle t with the point[n] */
int PointNumber(t,n)
struct triangle *t;
int n;
{
    int i=0;
    while(t—→P[i]!=n) {
        i++;
        if(i>3)
            error(" Point Number can't find");
    }
    return(i);
}

/* Initial stack to empty */
static InitStackQ()
{
    /* The first stack element is an empty item, it is used as an indicator */
to see if the stack is empty. only this item's prev is NULL */
struct stack *Curr;
Stack=(struct stack *)malloc((long)sizeof(struct stack));
if(Stack==NULL) error("Error: Alloc Space");
Curr=(struct stack *)malloc((long)sizeof(struct stack));
if(Curr==NULL) error("Error: Alloc Space");
Stack->Edge=-1; Stack->t0=NULL; Stack->t1=NULL;
Curr->Edge=-1; Curr->t0=NULL; Curr->t1=NULL;
Stack->prev=Stack;
Curr->prev=Stack;
Stack=Curr;
}

/* Push Stack */
/* push edge, triangle t0 t1 into stack */
/* Stack: is the current place to put the push value */
static Push(Edge,t0,t1)
int Edge;
struct triangle *t0,*t1;
{
struct stack *Curr;
Curr=(struct stack *)malloc((long)sizeof(struct stack));
if(Curr==NULL) error("Error: Alloc Space");
Stack->Edge=Edge;
Stack->t0=t0;
Stack->t1=t1;
/* Initial new place for the next value */
Curr->prev=Stack;
Stack=Curr;
}

/* Pop out edge, triangle t0 t1 from the stack */
static Pop(Edge,t1,t2)
int *Edge;
struct triangle **t0,**t1;
{
int i,j,k;
struct stack *Curr;
/* find the value to be popped */
Curr=Stack;
Stack=Stack->prev;
if(Stack->prev==NULL) error("POP Stack UnderFlow ");
*Edge=Stack->Edge;
*t0=Stack->t0;
*t1=Stack->t1;
free(Curr);
}

/*EmptyStack: if the stack Empty? . if empty, return 1, else return 0 */
static EmptyStackQ()
{
if((Stack->prev)->prev==NULL)
return(1);
else return(0);
}

/* CheckStack: Take out all the occurrences of t1 t2 from the stack */
CheckStack(t1,t2)
struct triangle *t1,*t2;
{
struct stack *curr1,*curr2,*tmp;
curr1=Stack;
curr2=curr1—prev;
while((curr2—prev)^NULL) { /* Stack not empty */
if(SameTriangle(curr2,t1,t2)==1) {
	tmp=curr2—prev; /* By pass the current element */
	free(curr2);
	curr1—prev=tmp;

curr2=curr1;
}
curr1=curr2;
if(curr1^NULL && (curr1—prev)^NULL)
curr2=curr1—prev;
}
/* to see if curr->t0 and curr->t1 contain the same triangle as t1 t2 */
int SameTriangle(curr,t1,t2)
struct stack *curr;
struct triangle *t1,*t2;
{
if((curr—t0)—P[0]==t1—P[0] && (curr—t0)—P[1]==t1—P[1]
&&((curr—t0)—P[2]==t1—P[2])
return(1);
else if((curr—t1)—P[0]==t1—P[0] && (curr—t1)—P[1]==t1—P[1]
&&((curr—t1)—P[2]==t1—P[2])
return(1);
else if((curr—t0)—P[0]==t2—P[0] && (curr—t0)—P[1]==t2—P[1]
&&((curr—t0)—P[2]==t2—P[2])
return(1);
else if((curr—t1)—P[0]==t2—P[0] && (curr—t1)—P[1]==t2—P[1]
&&((curr—t1)—P[2]==t2—P[2])
return(1);
else
return(0);
}

/* Initial queue */
InitQueue()
{
head=(struct queue *)malloc((long)sizeof(struct queue));
if(head==NULL)
error("Allocate space for Queue head");
head—next=NULL; head—t=NULL;
tail=head;
}

if queue is empty, return 1, else return 0 */
EmptyQueue()
{
    if (tail==head)
        return(1);
    else return 0;
}

/* Enqueue: put an element into the queue */
EnQueue(t)
struct triangle *t;
{
    struct queue *curr;
    curr=(struct queue *)malloc((long)sizeof(struct queue));
    if(curr==NULL)
        error("Allocate space for Queue");
    curr->next=NULL;
    curr->t=t;
    tail->next=curr;
    tail=curr;
}

/* Outqueue: the first element out the queue */
OutQueue(t)
struct triangle **t;
{
    struct queue *curr;
    if(tail==head)
        error("Queue is already empty");
    curr=head->next;
    free(head);
    head=curr;
    *t=curr->t;
}

/* Distance between two points */
float Distance(x1,y1,x2,y2)
float x1,y1,x2,y2;
{
    float t;
    t=(x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
    t=sqrt(t);
    return(t);
}

% drawcontour.c

/* display contour map */
/* Weibao Wu, June 16, 1992 */
/* change for X11 XSDV package, July, 92 */
#include <math.h>
#include <stdio.h>
#include "image.h"
#include "globals.h"

int OnBoundary(struct triangle *t);

DrawContour(InterpPoint, OutPutLine);

void (*InterpPoint)(struct triangle *t, float x, float y, float *z);
void OutPutLine(POINT *, POINT *);
{
  int i,j,k;
  int x,y,c;
  char buf[20];
  static float level[MAX_LEVELS];
  static float MaxLevel, MinLevel;
  static int nlevels, done=0;
  fprintf(stdout, "Contouring begin... An ");
  fprintf(stdout, "Set new levels [Y/N] ");
  gets(buf);
  if(buf[0]!='Y' && buf[0]!='n')
    do {
      printf("MaxX=\%,10.6f MinX=\%,10.6f\n", mx, nx);
      printf("MaxY=\%,10.6f MinY=\%,10.6f\n", my, ny);
      printf("MaxZ=\%,10.6f MinZ=\%,10.6f\n", mz, nz);
      printf(" select # levels : ");
      scanf("%d", &nlevels);
      printf(" select MaxLevel : ");
      scanf("%f", &MaxLevel);
      printf(" select MinLevel : ");
      scanf("%f", &MinLevel);
      if((MaxLevel<nz) || (MinLevel>mz))
        printf("Level error, select again\n");
      else if((nlevels<=0) || (nlevels>MAX_LEVELS)) {
        printf("error: depth and n levels must be >0 and <%d\n", MAX_LEVELS);
      }
      else done=1;
    } while(!done);

  if(nlevels==1) level[0]=MaxLevel;
  else {
    for(i=0; i<nlevels; i++)
      level[i]=MinLevel+(MaxLevel-MinLevel)*(float)(nlevels-1-i)/(nlevels-1);
  }
  Contour(level, nlevels, 4, OutPutLine, InterpPoint); /* level */
  fprintf(stdout, "Contouring end\n");
}
/* Contour the scattered data: Triangle contour */
/* input : level[], #levels, Depth: the depth of subdivision */
Contour(level, nlevels, Depth, OutPutLine, InterpPoint)
float level[];
int nlevels, Depth;
void (*OutPutLine)(POINT *, POINT *);
int (*InterpPoint)(struct triangle *t, float x, float y, float z);
{
  int i;
  struct triangle *t;
  struct point P[3];
t=Triangle;
Traverse(t, 0);
InitQueue();
EnQueue(t);
do {
  OutQueue(&t);
  t—*visit=1;
  for(i=0;i<3;i++) {
    /* if edge has neighbour and has not been visited */
    EnQueue(t—>E[i]);
  }
  for(i=0; i<3; i++) {
    P[i].x=Point[t—>P[i]].x;
    P[i].y=Point[t—>P[i]].y;
    P[i].z=Point[t—>P[i]].z;
  }
  /* if(!OnBoundary(t)) */
  trace(t, P, level, nlevels, 0, Depth, OutPutLine, InterpPoint);
} while(EmptyQueue()!=1);
return;
}

/* trace the contour line */
/* Trace is recursively called */
/* input : P[]: triangle three points, */
/* t: triangle pointer. nlevels in level[] */
trace(t, P, level, nlevels, currdepth, Depth, OutPutLine, InterpPoint)
struct triangle *t;
struct point P[];
float level[];
int nlevels, currdepth, Depth;
void (*OutPutLine)();
void (*InterpPoint)(struct triangle *t, float x, float y, float z);
{
  int i,j;
  struct point Pm[3], P0[3], P1[3], P2[3], P3[3];
if(currdepth<Depth) { /* subdivide the triangle */
    FindMiddlePoints(t, P, Pm, InterpPoint);
    P0[0].x=P[0].x; P0[0].y=P[0].y; P0[0].z=P[0].z;
    P0[1].x=Pm[0].x; P0[1].y=Pm[0].y; P0[1].z=Pm[0].z;
    P0[2].x=Pm[2].x; P0[2].y=Pm[2].y; P0[2].z=Pm[2].z;
    trace(t, P0, level, nlevels, currdepth+1, Depth, OutPutLine, InterpPoint);
    P1[0].x=P[1].x; P1[0].y=P[1].y; P1[0].z=P[1].z;
    P1[1].x=Pm[1].x; P1[1].y=Pm[1].y; P1[1].z=Pm[1].z;
    P1[2].x=Pm[0].x; P1[2].y=Pm[0].y; P1[2].z=Pm[0].z;
    trace(t, P1, level, nlevels, currdepth+1, Depth, OutPutLine, InterpPoint);
    P2[0].x=P[2].x; P2[0].y=P[2].y; P2[0].z=P[2].z;
    P2[1].x=Pm[2].x; P2[1].y=Pm[2].y; P2[1].z=Pm[2].z;
    P2[2].x=Pm[1].x; P2[2].y=Pm[1].y; P2[2].z=Pm[1].z;
    trace(t, P2, level, nlevels, currdepth+1, Depth, OutPutLine, InterpPoint);
    trace(t, Pm, level, nlevels, currdepth+1, Depth, OutPutLine, InterpPoint);
}
else { /* output the contour */
    OutPutContour(P, level, nlevels, OutPutLine);
}

/* output the contour lines in the triangle at levels */
/* input : P[3] triangle three points, levels: at levels */
OutPutContour(P, level, nlevels, OutPutLine)
struct point P[];
float level[];
void (*OutPutLine)();
int nlevels;
{
    int i, j;
    struct point *Tpoints[3], *tmp;
    struct point cpnts[2];  /* contour points */
    /* if all three points are equal, do not output */
    if((P[0].z==P[1].z) & (P[0].z==P[2].z))
        return;
    for(i=0; i<3; i++)
        Tpoints[i]=&P[i];
    /* bubble sort the three points */
    for(i=2; j>0; j--)
        for(i=0; i<j; i++)
            if(Tpoints[i].z>Tpoints[i+1].z) {
                tmp=Tpoints[i];
                Tpoints[i]=Tpoints[i+1];
                Tpoints[i+1]=tmp;
            }

    for(i=0; i<nlevels; i++) {
        if((level[i]<Tpoints[2].z) & (level[i]>Tpoints[1].z)) {
            LineIntersectPoint(Tpoints[2], Tpoints[1], level[i], &cpnts[0]);
            LineIntersectPoint(Tpoints[2], Tpoints[0], level[i], &cpnts[1]);
            OutPutLine(&cpnts[0], &cpnts[1]);
        }
    }
}
    OutPutLine(&Tpoints[2], &Tpoints[1]);
    LineIntersectPoint(Tpoints[0], Tpoints[2], level[i], &cpnts[0]);
    LineIntersectPoint(Tpoints[0], Tpoints[1], level[i], &cpnts[1]);
    OutPutLine(&cpnts[0], &cpnts[1]);
}
else if((level[i]==Tpoints[1]→z) & (level[i]==Tpoints[0]→z))
    OutPutLine(&Tpoints[1], &Tpoints[0]);
}

/* to see if there is a contour line in the triangle */
/* return 1 if there is non. */
NoIntersection(P, level, nlevels)
struct point P[];
float level[];
int nlevels;
{
    int i, j;
    struct point *Tpoints[3], *tmp;
    struct point cpnts[2];    /* contour points */
    if((P[0].z==P[1].z) & (P[0].z==P[2].z)) /* no intersection */
        return (1);
    for(i=0; i<3; i++)
        Tpoints[i]=&P[i];
    /* bubble sort the three points */
    for(j=2; j>0; j--)
        for(i=0; i<j; i++)
            if(Tpoints[i]→z>Tpoints[i+1]→z) {
                tmp=Tpoints[i];
                Tpoints[i]=Tpoints[i+1];
                Tpoints[i+1]=tmp;
            }
    for(i=0; i<nlevels; i++)
        if((level[i]≤Tpoints[2]→z) & (level[i]≥Tpoints[0]→z)) {
            return(0); /* intersection here */
        }
    return(1); /* no intersection */
}

/* find middle points */
/* input : three points from the triangle */
/* output: three middle points */
/* change FindMiddlePoint from linear interpolation to Gauss */

FindMiddlePoints(t, P, Pm, InterpPoint)
struct triangle *t;
struct point P[], Pm[];
void (*InterpPoint)(struct triangle *t, float x, float y, float *z);
{
  int i;
  /*Find the three triangles around t and get the plane equations*/
  for(i=0; i<3; i++) {
    Pm[i].x=(P[i].x+P[(i+1)%3].x)*0.5;
    Pm[i].y=(P[i].y+P[(i+1)%3].y)*0.5;
    InterpPoint(t, Pm[i].x, Pm[i].y, &Pm[i].z);
  }
}

/* intersection points:
   Input :two points, and level
   output:intersection points at level */

LineIntersectPoint(P0, P1, level, P)
struct point *P0, *P1, *P;
float level;
{
  float t, t2;
  t2=P1->z-P0->z;
  if(t2==0) {
    printf("LineIntersection error t2=0\n");
    exit(—1);
  }
  t=(level-P0->z)/t2;
  if(fabs(t)> 1.0) {
    printf("LineIntersection error t>1\n");
    exit(—1);
  }
  P->x=P0->x+(P1->x-P0->x)*t;
  P->y=P0->y+(P1->y-P0->y)*t;
  P->z=level;
}

% equation.c

/* equation */
/* Solve equation by Gauss elimination method */
/* Weibao Wu, Nov. 8, 1991 */
#include <stdio.h>
#include <math.h>
/* Solve Equation */
inputs:
  A: matrix A[NSP][NSP+1]
  B: vector B[NSP];
  NSP: number of variables to be solved
Output:
X[NSP]: solution of AX=B

// SolveEquation(A,B,X,NSP)
float **A,B[],X[];
int NSP;
{
    int i,j,k,l,m;
    int r,c; /* row col. index */
    float t,t1;
    /* Solve the equation by Gauss elimination method */
    /* Externed matrix */
    for(i=0;i<NSP;i++)
        A[i][NSP]=B[i];
    /* Gauss elimination */
    for(i=0;i<NSP;i++)
    {
        ExchangeMax(A,i,NSP); /* exchange the max value in the col. to (i,i) */
        for(l=i+1;l<NSP;l++)
        {
            t=A[l][i]/A[i][i];
            for(m=i;m<NSP+l;m++)
                A[l][m]=A[l][m]-A[i][m]*t;
        }
    }
    Solution(A,X,NSP);
}

/* input:
   A: matrix to be inverted.
   NSP : # points of the matrix /row
   output
   A: inversed matrix */
InverseMatrix(A,NSP)
float **A;
int NSP;
{
    int i,j,k,l,m;
    int r,c; /* row, col index */
    float *B[100]; /* aux matrix */
    float t,t1,*ptr;
    if(NSP>100) {perror("InverseMatrix: NSP too large"); exit(-1);}
    ptr=(float *)malloc((long)NSP*NSP*sizeof(float));
    if(ptr==NULL) {perror("Allocate space for aux matrix B"); exit(-1);}
    for(j=0;j<4;j++)
        B[i]=ptr+j*NSP;
    /* Solve the equation by Gauss elimination method */
    /* Externed matrix=ID matrix */
    for(i=0;i<NSP;i++)
        B[i][i]=1.0;
    /* Gauss elimination */
for(i=0;i<NSP;i++) {
  ExchangeMax(A,i,NSP); /* exchange the max value in the col. to (i,i) */
  for(l=i; l<NSP; l++)
    A[i][l]=A[i][l]/A[i][i]; /* Normalize itself */
  for(l=i+1;l<NSP;l++) {
    t=A[l][i];
    for(m=0; m<NSP;m++)
      if(m!=i) {
        B[l][m]=B[l][m]−A[i][m]∗t;
      }
  }
}
for(i=0;i<NSP;i++)
  A[i][i]=B[i][i];
free(ptr);
}

/* Exchange the current max col. value with (n,n) */
ExchangeMax(A,n,NSP)
float **A;
int n,NSP;
{
  int i,j,m;
  float t;
  t=A[n][n]; m=n;
  for(i=n+1;i<NSP;i++) /* find the max */
    if(fabs(t) <fabs(A[i][n])) {
      t=A[i][n]; m=i;
    }
  if(m!=n) /* Exchange value */
    for(i=0;i<NSP+1;i++) {
      t=A[n][i];
      A[n][i]=A[m][i];
      A[m][i]=t;
    }
}

/*for Upper triangle Matrix */
/* Input: 
  UP:  Upper triangle, UP[NSP][NSP+1]
  NSP: # of input points 
Output:
  X: solution the equation
*/
Solution(UP,X,NSP)
float *UP[],X[];
int NSP;
{
int i,j,k,l;
float t=0;
for(i=NSP-1;i≥0;i--) {
t=0;
for(j=NSP−1;j>i;j--)
t+=UP[i][j]•X[j];
X[i]=(UP[i][NSP]−t)/UP[i][i];
}
}