Massachusetts Institute of Technology
Department of Civil and Environmental Engineering

Final Technical Report

to

U.S. Army Research Office

Funding Number: DAAD 19-00-1-0436

Agency Report Number: 40368EV

FRACTURE FLOW RESEARCH

Volume 1

Modeling Rock Fracture Intersections - Application in the Boston Area and Estimation of the Well-Test Flow Dimension

by

Jean Louis Z. Locsin

Herbert H. Einstein

August 2005

REPORT DOCUMENTATION PAGE	Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE August 4, 2005

3. REPORT TYPE AND DATES COVERED
Final Report July 1, 2000 – Dec. 31, 2004

4. TITLE AND SUBTITLE
Fracture Flow Research
Volume 1: Modeling Rock Fracture Intersections – Application in the Boston Area and Estimation of the Well-Test Flow Dimension
~~Volume 2: Modeling Joint Patterns Using Combinations of Mechanical and Probabilistic Concepts~~

5. FUNDING NUMBERS
DAAD19-00-1-0436

6. AUTHOR(S)
Herbert H. Einstein, Jean-Louis Locsin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
Department of Civil and Environmental Engineering
77 Massachusetts Avenue, 1-342
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
40368-EV

40368.1-EV

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Fractures govern flow, deformation and strength of rock masses. Fracture flow is important with regard to resource extraction (water, gas, oil) as well as groundwater contamination. Fractures through their effect on deformability and strength goven stability of tunnels and slopes in rock and the behavior of building,-, bridge – and dam foundations. Very importantly, they also strongly affect penetration resistance.

The research consisted of two components, modeling of joint (fracture) intersections and fracture pattern modeling. The two volumes of the final report correspond to these two components.

Fracture intersections govern connectivity of fracture patterns and fracture flow (deformability, strength). The research developed an algorithm with which orientation and length of fracture intersections can be represented. The algorithm was applied and tested with a synthetic case and the fracture pattern in the Boston area. Also, an attempt at simplified flow dimension models was undertaken. The results show that complete numerical modeling is better than simplified modeling.

Fracture pattern modeling so far was limited to geometric models or simple mechanical models. The research developed a new model to represent fracture (joint) patterns in sedimentary rock specifically, layer perpendicular joints. One model is mechanically based (flaw model) the other one (rejection model) is quasi-mechanical. Both models have probabilistic aspects. The models' predictions were compared to fracture patterns observed in the field. The models, particularly the flaw model perform satisfactorily. Issues requiring further study are joint spacing smaller than saturation and joints which cross more than one layer.

14. SUBJECT TERMS
Rock Fracture, Fracture Flow, Joint Patterns, Fracture Intersections

15. NUMBER OF PAGES
345

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION ON THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

List of Tables

1 Introduction

Fracture connectivity is important in assessing the flow behavior of a fracture formation as well as the resistance of a rock mass. Dershowitz (1984) introduced parameters to describe fracture connectivity that are based on fracture intersection geometry. The more familiar parameters include C_1, which is the number of intersections between fractures per unit volume and C_{8i}, defined as the extent of a cluster of fractures in the direction i.

Meyer (1999), incorporated into GEOFRAC the capability to group the modeled fractures into isolated clusters of fractures and calculate the physical extent of these clusters in three-dimensions. The physical extents of the fracture sub-networks in three dimensions are described using the parameters C_{8x}, C_{8y}, and C_{8z} that were proposed by Dershowitz (1984). These are the dimensions of a box in the x, y and z directions that can completely contain the fracture sub-network. Meyer used this feature to study the connectivity of the fracture networks in the Boston Basin and the simulations show that the sub-networks tend to propagate a greater distance in the vertical direction as opposed to the horizontal direction suggesting more vertical connectivity in terms of physical extent.

The work in this report will proceed from determining the existence of the connections between the fractures and building the fracture clusters to calculating the geometry and orientations of these connections, the main contribution being an algorithm for calculating the geometry of each fracture intersection. First, a background of the fracture modeling concepts developed by Ivanova (1998) that are used in GEOFRAC will be presented. The development of the algorithm for determining the geometry of the fracture intersections will then follow. The capability of modeling the intersections will then be used to study the fracture intersections in the Boston Basin. The simulations will be performed using five different modeling volume sizes: 10^3, 12^3, 14^3, 15^3 and 20^3 m^3. The resulting distribution of intersection lengths and the distribution of the orientations of the fracture intersections will be studied for the various modeling volume sizes. The distribution of the intersection orientations is also compared to the calculated orientations (i.e. the orientations to be expected using the mean orientations of the fracture sets) in order to assess the degree in which each fracture set is involved in the formation of these intersections.

Another parameter that describes fracture connectivity and is linked directly to the flow behavior of a fracture network is the well-test flow dimension. Well-tests are performed to determine the flow properties as well as the geometry of the fracture network in a rock mass. The well-test flow dimension describes how the flow properties and the area of the flow paths vary with distance from the well. In order to obtain the well-test flow dimension for a simulated fracture network, finite element flow simulations are needed. However, Dershowitz (1996) proposed a method for estimating the well-test flow dimension directly from the change in the flow area or flow width with distance from the well (distance-flow area or distance-flow width relationships). This method requires that the geometry of each intersection as well as the geometry of the flow paths from the well into the fracture network be known. Since the flow in a fracture network passes through individual fractures by way of the fracture intersections located on these fractures, the flow area has been defined as a function of the lengths of these fracture intersections. Dershowitz suggested that the flow width existing between two intersections be the average of the lengths of these two intersections multiplied by a factor that depends on the orientation of the line connecting the midpoints of these intersections. This opens the possibility that a very short intersection will be paired with a very long intersection resulting in a flow width that may be representative of neither intersection. Hence, two new approaches for deriving the distance-flow width relationships will also be presented. The first one calculates the flow width based on the length of an individual intersection (i.e. no averaging is done), the second approach divides the modeling volume into smaller cubical elements to which groups of individual intersections are assigned. The cubical elements act as individual conductive elements and the connections among these cubical elements represent the pathways that the flow takes. After the flow widths of the paths in the fracture networks are computed, the distance-flow width relationships for each network are constructed. The flow dimension values are then derived from these relationships. Calculating the flow dimension from distance-flow width relationships is an approximate approach, the three different methods of defining the flow width leading to three different approaches. These being approximate approaches, questions regarding their accuracy arise. It is therefore necessary to compare the flow dimension results from these approaches to those of finite element flow simulations.

As shown in **Figure 1.1**, the flow dimension can be thought of as a link between the connectivity of a fracture network and its flow behavior. The connectivity can be described using the geometries of the fracture intersections by the parameters C_1 to C_{8i}. The flow behavior, on the other hand, can be described using the well-test flow dimension. The flow dimension is generally obtained by running finite element flow simulations on the fracture network but physically, the flow dimension also represents the change in flow area with radial distance from the well. The approximate methods mentioned in the previous paragraph calculate this change in flow area based on the lengths of the individual fracture intersections.

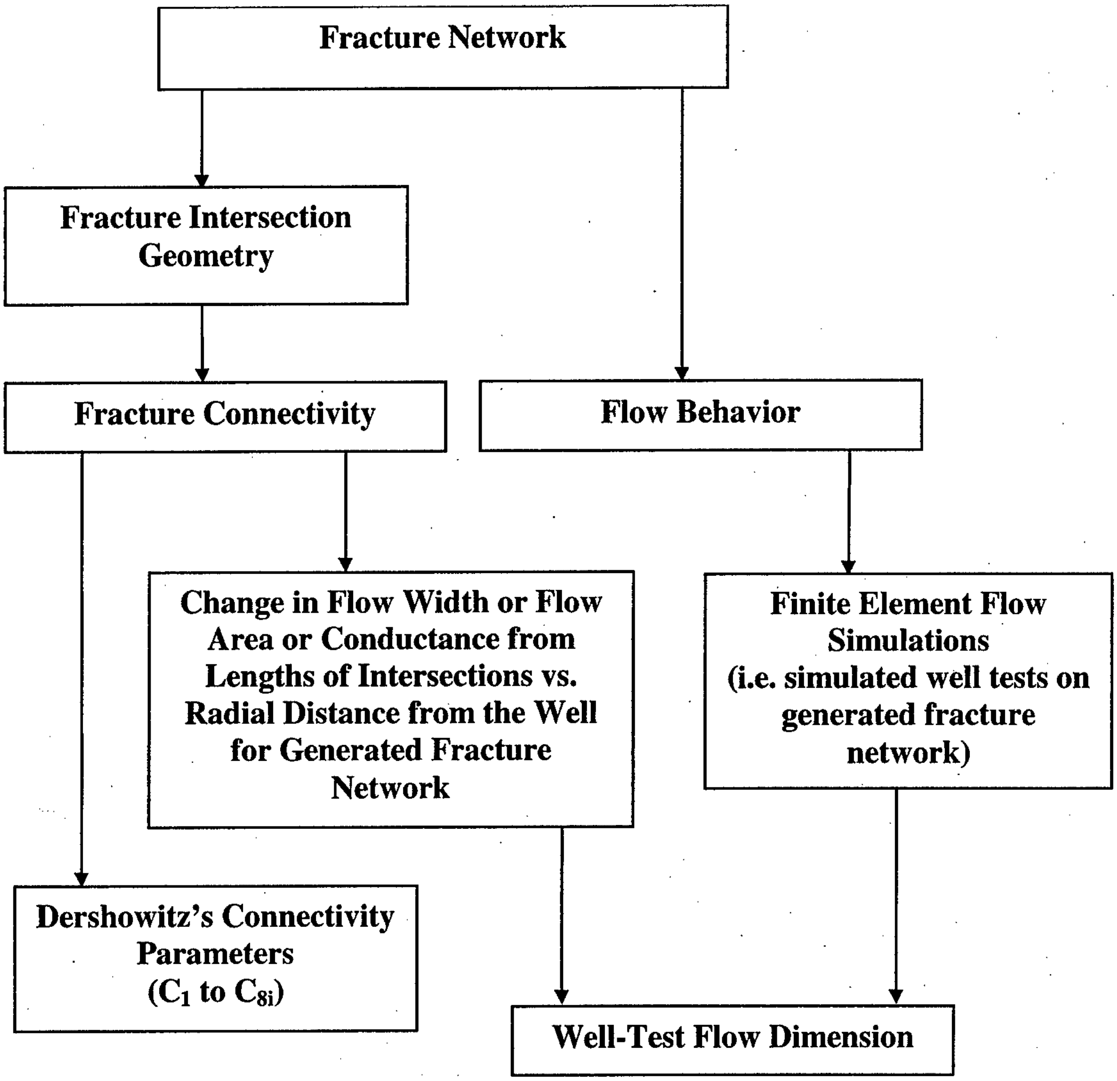

Fracture Network
Fracture Intersection Geometry
Fracture Connectivity
Flow Behavior
Change in Flow Width or Flow Area or Conductance from Lengths of Intersections vs. Radial Distance from the Well for Generated Fracture Network
Finite Element Flow Simulations (i.e. simulated well tests on generated fracture network)
Dershowitz's Connectivity Parameters (C_1 to C_{8i})
Well-Test Flow Dimension

Figure 1.1 – Fracture connectivity and flow behavior

2 GEOFRAC: Fracture Modeling Procedures

The fracture model on which GEOFRAC is based was developed by Ivanova (1998). Ivanova's model, in turn, is an extended version of Veneziano's model. Veneziano used two stochastic processes to model the fracture network. First, an anisotropic Poisson network of planes is generated in three-dimensional space. Then a Poisson network of lines is generated on these planes to produce polygonal shapes. The polygons then undergo a random process, a so called marking procedure, that decides if a polygon is either fractured or intact rock. An obvious limitation of this model is that the resulting fractures generated on one specific plane remain coplanar after the marking procedure. Ivanova eliminated this limitation by introducing another stochastic process that modifies the location (by shifting) and orientation (by rotation) of the fractures in order to model specific geologic features. Ivanova also introduced combinations of criteria to be used in the marking procedure as well as the capability to mark the created polygons as a function of their distances to a structure (marking with respect to faults). Meyer (1999), later on, enabled the user to define the marking zones more sharply thereby introducing a fourth stochastic process. To summarize, the four stochastic processes are:

- Primary Process – Modeling the stress field orientation using a homogeneous, anisotropic Poisson plane network (**Figure 2.1**).

- Secondary Process – The planes are tessellated into smaller polygons using a homogeneous Poisson line process. The polygons are marked as fractured or intact rock according to marking rules. **Figure 2.2** and **Figure 2.3**.

- Tertiary Process – Zones are defined within the modeling volume and polygons located in the zones are retained or discarded according to probabilities assigned to each zone. Cutting probabilities are assigned to structures such as faults to control the propagation of fractures through the fault. **Figure 2.4**

- Quaternary Process – The remaining fracture polygons are translated and rotated to reflect major geologic features. **Figure 2.5** and **Figure 2.6**.

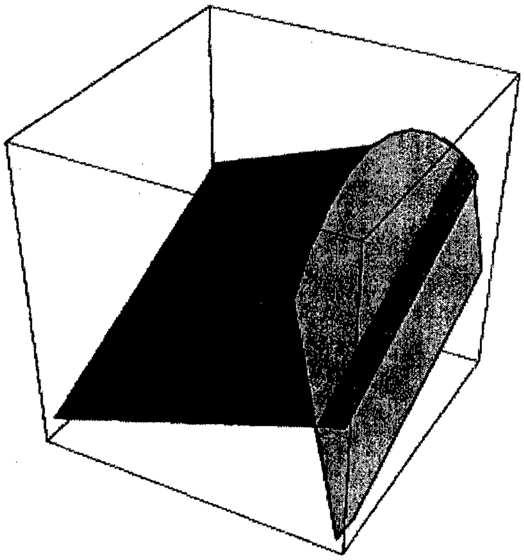


Figure 2.1 – Primary process, Poisson plane network (from Ivanova, 1998)

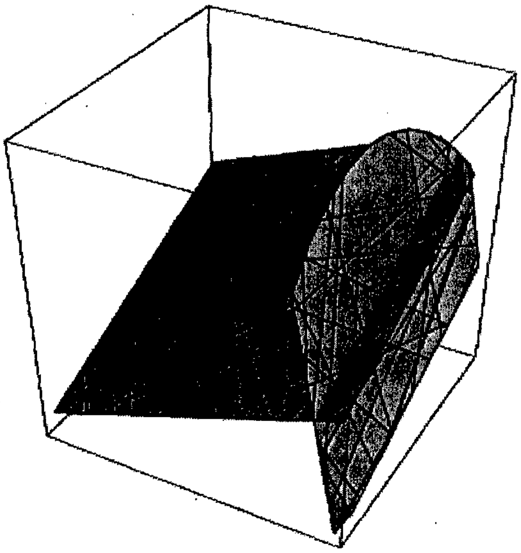


Figure 2.2 – Poisson line tessellation on the planes (from Ivanova, 1998)

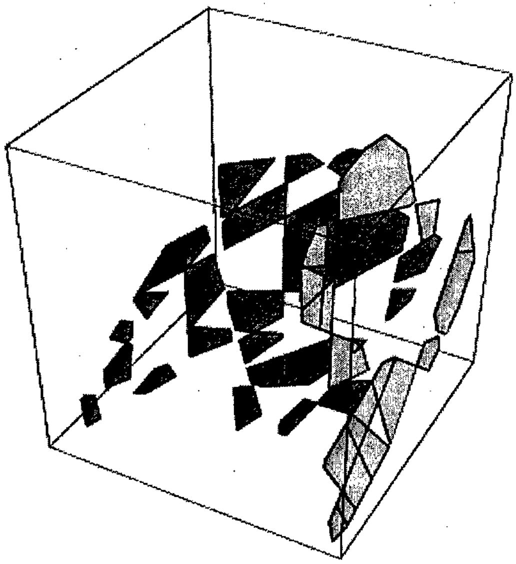


Figure 2.3 – Polygon marking process produces fractured and intact rock (from Ivanova, 1998)

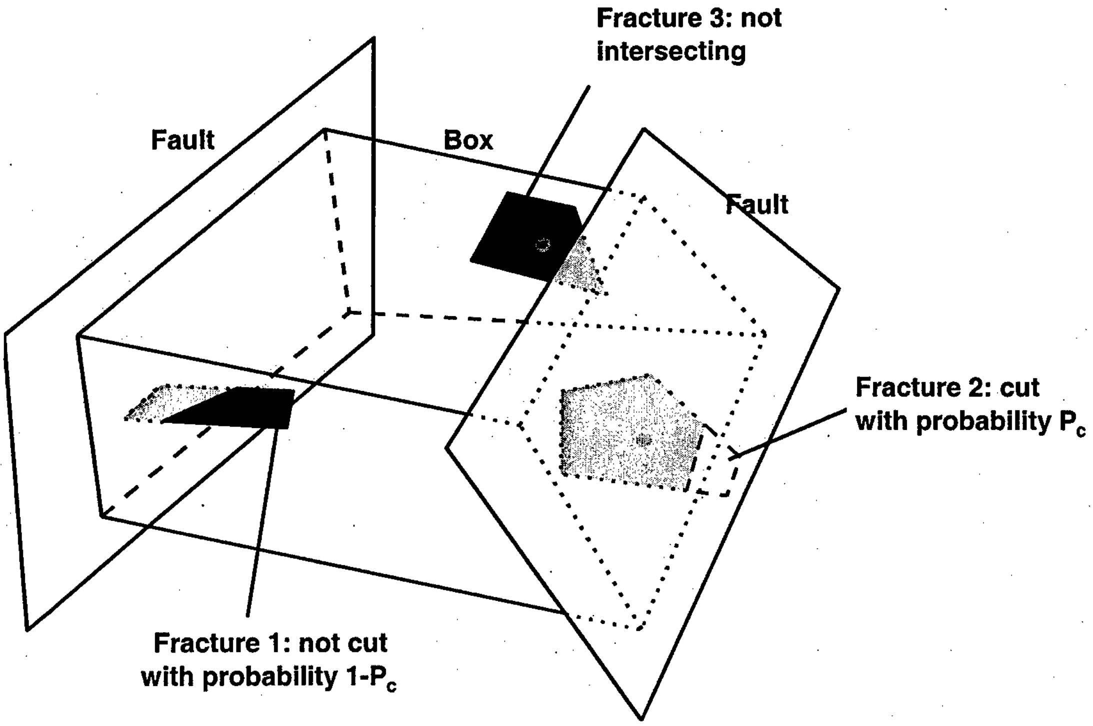

Fracture 3: not intersecting
Fault
Box
Fault
Fracture 2: cut with probability P_c
Fracture 1: not cut with probability $1-P_c$

Figure 2.4 – Zone marking process (from Meyer, 1999)

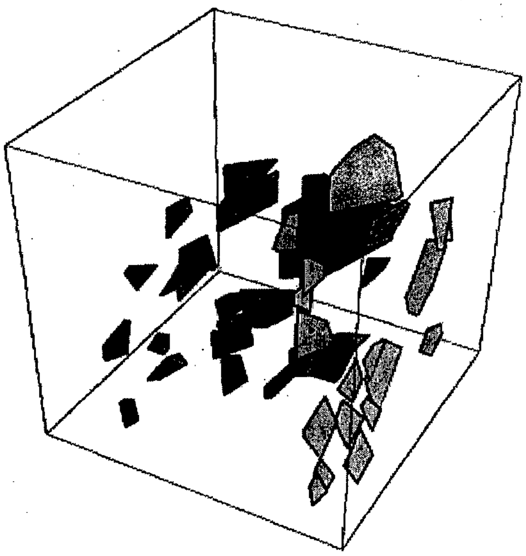


Figure 2.5 – Translation of remaining polygons (from Ivanova, 1998)

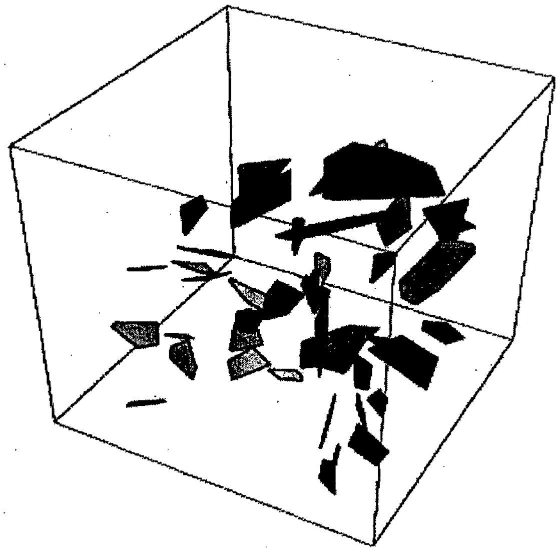


Figure 2.6 – Rotation of remaining polygons (from Ivanova, 1998)

As far as fracture connectivity is concerned, GEOFRAC's capabilities are limited to the division of the fracture network into sub-networks of isolated fracture clusters. GEOFRAC also calculates the horizontal and vertical extents of these sub-networks. The work in this report adds to GEOFRAC the capability to model the individual intersections formed between the fractures. The algorithm for the calculation of the length of intersection will make use of elements and procedures found in Ivanova (1998). Important concepts developed by Ivanova (1998) are presented below as a background to the algorithm for determining the geometry of the intersections that will be discussed in **Chapter 3**.

2.1 Defining the Modeling Volume

The modeling volume is the space defined by the input values $\mathbf{X_m}$, $\mathbf{Y_m}$, and $\mathbf{Z_{box}}$ (see **Figure 2.7**). Only the fractures within the modeling volume will be considered. **Figure 2.7** also shows the global coordinate system (**OXYZ**).

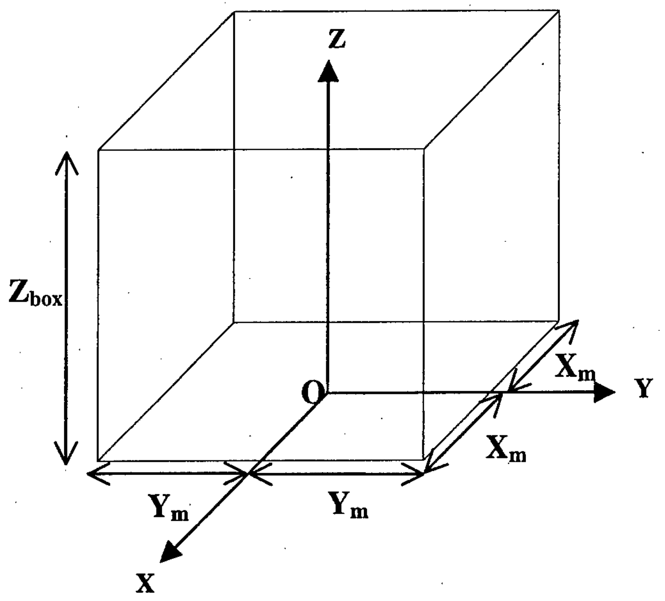


Figure 2.7 – Modeling volume and the global axes

The space shown in **Figure 2.7** represents the basic modeling volume used in GEOFRAC. The top surface can also be entered as quadratic or a cubic surface. A quadratic surface is defined in the program by six floating values (*A*, *B*, *C*, *D*, *E* and *F*) and it is represented in three-dimensional space by the following equation:

$$Z = AX^2 + BXY + CY^2 + DX + EY + F$$

Equation 2.1

Use of a quadratic surface to define the top of the modeling volume helps model the topographic surface more realistically. A cubic surface on the other hand is defined by 10 floating values (*A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I* and *J*) in the equation:

$$Z = AX^3 + BX^2Y + CXY^2 + DY^3 + EX^2 + FXY + GY^2 + HX + IY + J$$

Equation 2.2

A cubic surface is used to model folds.

2.2 Generation of a Plane in Space

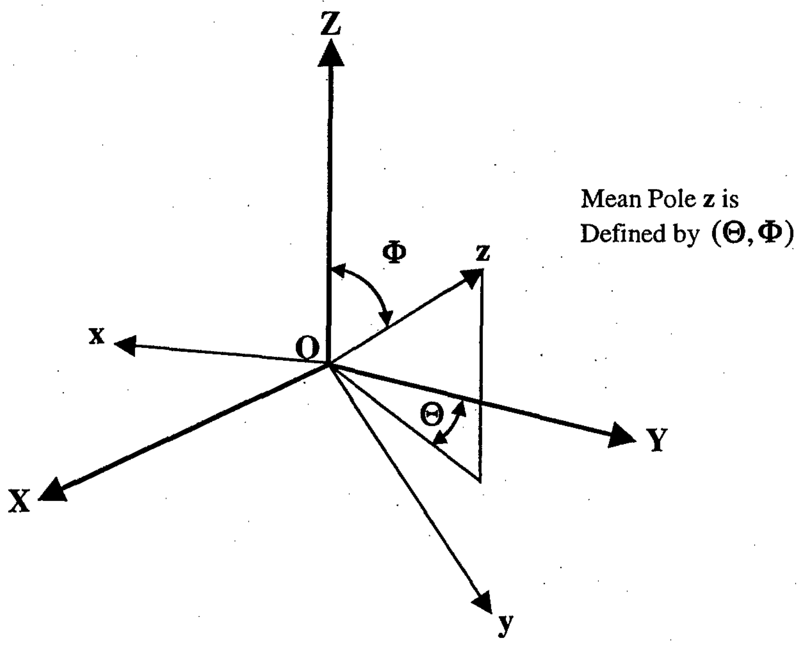
After the modeling volume is defined, planes are generated to intersect it. Ivanova referred the equation of a plane to the frame of reference of a fracture set (**Oxyz**). For clarity, the frames of reference that Ivanova used to define planes and the tessellation lines are shown in **Figure 2.8** to **Figure 2.10** and they are defined below.

OXYZ – Global Frame of Reference (**+Y** is North and **+X** is East, **Figure 2.8**)

Oxyz – Frame of Reference of a Fracture Set (defined by **Θ** and **Φ**, the azimuth and the latitude of the pole **z**, respectively, **Figure 2.8** and **Figure 2.9**)

O'x'y'z' – Local Frame of Reference of a Plane (**Figure 2.10**)

Ox"y"z" – Frame of Reference Parallel to that of a plane but with the same origin as the Global Frame (**Figure 2.9** and **Figure 2.10**)



Mean Pole **z** is Defined by (Θ, Φ)

Figure 2.8 – Global axes and mean pole z of a fracture set

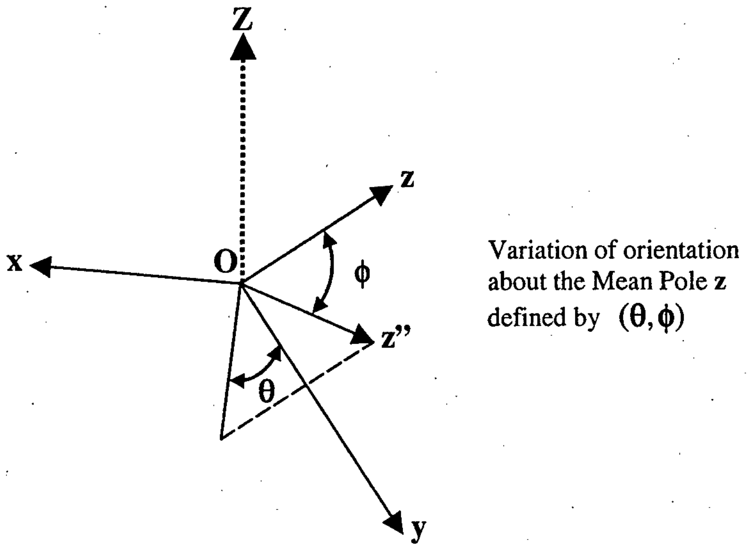

Z
z
x
O
ϕ
z"
θ
y
Variation of orientation about the Mean Pole z defined by (θ,ϕ)

Figure 2.9 – Axes formed by the mean pole z of a fracture set and the pole z" of a generated plane

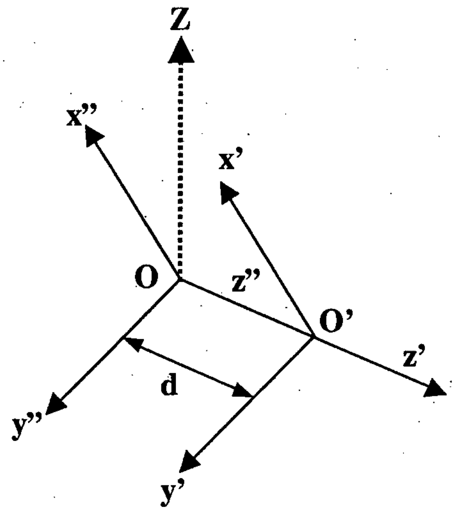


Figure 2.10 – Local frame of reference of a plane

To create the planes in the Primary process as in **Figure 2.1**, Ivanova used the frame of reference of a fracture set as the basis around which the orientations and locations of the generated planes must vary. The equation of a plane in the frame of reference of the fracture set (**Oxyz**) is:

$$x\sin\theta\sin\phi + y\cos\theta\sin\phi + z\cos\phi = d$$

Equation 2.3

where **d** is the distance from the global origin to the plane (**Figure 2.10**) and the pair (θ,ϕ) represents the azimuth and latitude of the normal to the plane. The joint probability density function of (θ, ϕ) can be inferred from measured data. The possible PDF's include uniform or partial uniform, one-parameter or two-parameter Fisher and Bingham and constant orientation (Ivanova, 1998).

To produce a homogeneous Poisson plane network of intensity μ is equivalent to performing a Poisson point process in the region:

$$\{(\mathbf{d}, \theta, \phi) : -\infty < \mathbf{d} < \infty, 0 \le \theta \le \pi, 0 \le \phi \le \pi\}$$

with non-homogeneous intensity function of the type:

$$\mu(\mathbf{d}, \theta, \phi) = \mu \mathbf{f}_{\theta,\phi}(\theta, \phi)$$

where $\mathbf{f}_{\theta,\phi}(\theta, \phi)$ is the joint probability density function of θ and ϕ while μ is a positive constant (Ivanova, 1998).

2.3 Poisson Line Tessellation on the Plane

Once the planes have been generated, Poisson lines are created on these planes to divide them into polygons. The equation of the line (shown below) is referred to the **O'x'y'z'** axes.

$$x'\cos\alpha + y'\sin\alpha = D$$

Equation 2.4

The parameters α and **D** are shown below in **Figure 2.11**.

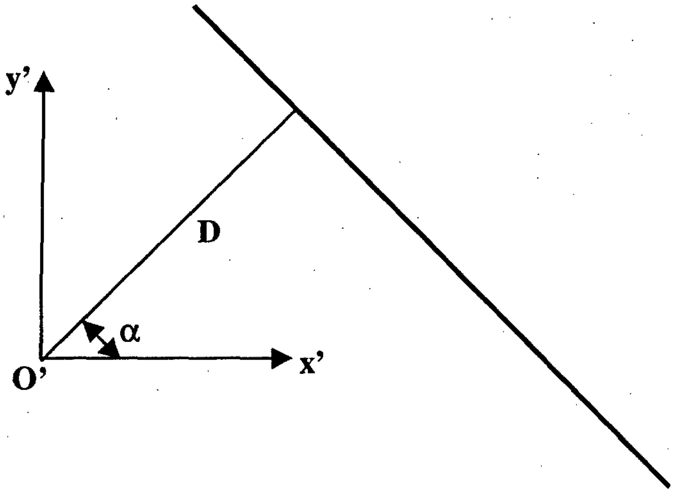


Figure 2.11 – Tessellation line defined by α and D

To create a homogeneous Poisson line network of intensity λ is equivalent to a Poisson point process in the region:

$$\{(\mathbf{D},\alpha) : 0 \leq \mathbf{D} \leq \infty, 0 \leq \alpha \leq 2\pi\}$$

with intensity function of the points of the type:

$$\lambda(\mathbf{D},\alpha) = \lambda \mathbf{f}_\alpha(\alpha)$$

where λ is a positive constant and $\mathbf{f}_\alpha(\alpha)$ is the probability density function of α in the interval $[0,2\pi]$.

GEOFRAC then assembles the polygons formed by the Poisson line tessellation. The next section discusses how the resulting polygons are classified as fractured or intact rock.

2.4 Marking the Polygons Formed by the Poisson Line Tessellation

After the polygons have been formed, GEOFRAC calculates the area, equivalent radius and the elongation of the polygons among other things. The program then discards the polygons according to user-defined limits on parameters such as shape (convexity, magnitude of internal angles and elongation) and size (area or equivalent radius) of the polygons. The remaining polygons are termed "good" polygons and the process is called the marking process.

For the original polygons (i.e. all those formed by the line tessellation), the mean and covariance matrices for the number of vertices and the polygon area are given by (Ivanova, 1998):

$$\begin{bmatrix} N \\ A \end{bmatrix} \sim \underline{\mu} = \begin{bmatrix} 4 \\ \dfrac{\pi}{\lambda^2} \end{bmatrix}, \underline{\Sigma} = \begin{bmatrix} \dfrac{(\pi^2-8)}{2} & \dfrac{\pi(\pi^2-8)}{2\lambda^2} \\ \dfrac{\pi(\pi^2-8)}{2\lambda^2} & \dfrac{\pi^2(\pi^2-2)}{2\lambda^4} \end{bmatrix}$$

N = Number of vertices
A = Area of the polygon

Ivanova defined "good" polygons (using the shape as the basis) with the following rules:

1. The polygon is convex and has at least four vertices
2. All internal angles are at least 60°
3. The polygon elongation is not more than 1.6

Using the Best Linear Unbiased Estimator, Ivanova calculated values of $E[A|N=n]$ for $n \geq 5$ in terms of the original $E[A]$ and showed that the expected area of a polygon given it has n vertices is $E[A|N=n] = kE[A]$ and that $k \geq 2$ for the range of n values considered. This means that larger polygons will tend to have more vertices. It also follows that due to rule number 1 above, it is more likely for a large polygon than it is for a smaller polygon to be retained as a potential fracture. As a consequence, the expected area of "good" polygons ($E[A']$) is larger than the expected area of the polygons produced by the line tessellation. Therefore, the input value for the expected area of the fractures has to be underestimated by a factor C_A to accommodate the rise in the expected area as a result of the marking procedure. The factor C_A is defined as (Ivanova, 1998):

$$C_A = \frac{E[A']}{E[A]}$$

$E[A']$ = Expected area of "good" polygons
$E[A]$ = Expected area of the polygons produced by the line tessellation

Another consequence of the marking procedure is that the total area of the fractures is substantially decreased resulting in a lower value of fracture intensity, P_{32} compared to the desired value. The desired value of P_{32} is achieved in the Primary process (**Figure 2.1** on page 18) from the generated planes (i.e. the total area of the generated planes divided by the volume is the desired P_{32}). A fraction of the area of each generated plane will be discarded as a result of the marking procedure making the final P_{32} lower than the desired intensity. It is therefore necessary to overestimate the total plane area to accommodate the reduction in fracture area due to the marking procedure. This is done by overestimating the Poisson plane intensity, μ, by dividing it by the factor γ. This factor is defined as (Ivanova, 1998):

$$\gamma = \frac{A'_T}{A_T}$$

A'_T = Total area of "good" polygons
A_T = Total area of polygons originally produced by the line tessellation (or the area of the generated plane that is within the modeling volume)

Numerical simulations by Ivanova (1998) show that when the marking rule follows only the three guidelines listed above, only 40% of the total area of the polygons produced by the Poisson line tessellation (or the area of the original plane) are considered fractured rock. The simulations also reveal that following the same three marking guidelines, the resulting expected area of "good" polygons is 2.2 times the expected area of the polygons formed by the Poisson line tessellation. This corresponds to values of $\gamma = 0.4$ and $C_A = 2.2$. Marking by shape can also be combined with other criteria such as relative size. **Table 2.1** below shows the values of C_A and γ for different marking rules. The first entry shows the values for marking according to the three rules given previously. The succeeding entries show the values for marking according to shape and according to the criterion listed. For example, when marking according to the three original rules **and** following the additional rule that $A'_i > 2E[A]$ (the area of the ith "good" polygon must be greater than twice the expected area of the polygons originally produced by the line tessellation), the values $C_A = 5.0$ and $\gamma = 0.30$ must be used. GEOFRAC allows a maximum input value of $\gamma_{\max} = 0.4$.

MARKING RULE	C_A	γ
According to the 3 rules above	2.2	0.4
$A'_i > E[A]$	3.6	0.36
$A'_i > 2E[A]$	5.0	0.30
$R'_{e,i} > 3E[R'_e]$	1.8	0.38
$R'_{e,i} > 2E[R'_e]$	1.4	0.23
$A'_i > E[A]$, $R'_{e,i} > 3E[R'_e]$	3.6	0.36
$A'_i > E[A]$, $R'_{e,i} > 2E[R'_e]$	3.4	0.36

Table 2.1 – Values of the factors C_A and γ for different marking rules (from Ivanova, 1998)

Theoretically, C_A and γ do not depend on the line intensity, λ, but only on the marking rule as long as the marking rule only governs shape and relative size (Ivanova, 1998).

To summarize, the fracture intensity and the fracture size are modeled in the following manner:

1. The desired mean fracture size and fracture intensity are chosen ($E[A']$ and P_{32})
2. The Poisson plane intensity is calculated as $\mu = P_{32} / \gamma$
3. The Poisson planes are generated in the modeling volume
4. The expected area of the polygons formed by the Poisson line tessellation is computed using the input expected area: $E[A] = E[A'] / C_A$
5. The Poisson lines are generated using an intensity $\lambda = (\pi / E[A\})^{1/2}$
6. The marking procedure is performed.

GEOFRAC also has the capability of introducing variations in fracture intensity in space. This is done by marking the polygons as fractured (or intact) as a function of distance to a pre-defined structure. This capability allows one to model more realistic fracture intensities throughout the modeling volume. For example, more intense fracturing may occur in the vicinity of a fault compared to areas far away from it. Higher probabilities of retention (assigning as fractured) can be assigned near the fault to simulate this condition. The area to which a certain probability of retention is assigned is called a zone. Each zone is bounded by the minimum and maximum distance within which the assigned retention probability is applicable. Meyer (1999) improved on the zone marking process by introducing a "box" marking algorithm. The user can define an irregular box and assign a retention probability to any polygon located within the box. A "cutting" probability can also be assigned to discontinuities such as faults to control the termination or continuation of a fracture that intersects the fault (see **Figure 2.4**).

2.5 Translation and Rotation of the Remaining Polygons

So far, the fractures (polygons) all lie on the originally defined planes and this does not necessarily correspond to reality. It is possible to translate (shift) the fractures by translating them by a distance **dz'$_{\mathbf{max}}$**. The quantity **dz'$_{\mathbf{max}}$**, defined below, is added or subtracted to the **z'** coordinates of each of the vertices of the selected fractures.

$$dz'_{\max} = C\frac{(E[R'_e])^2}{R'_e}$$

Equation 2.5

where

R'_e = equivalent radius of a "good" polygon
$E[R'_e]$ = expected value of the equivalent radius
C = a constant that dictates the degree of coplanarity of the fractures

The lower the value of C the more coplanar the resulting fractures. **Equation 2.5** also suggests that the amount of shift is dependent on the size of the fracture. Smaller fractures have larger shift distances compared to larger fractures.

Fracture polygon rotation is another process that is used in GEOFRAC to model more closely the change in the direction of the stress field (and the change in fracture orientation) relative to those assumed in the Primary process. For example, **Figure 2.12** shows the orientations of the fracture traces based on the stress field orientation used in the Primary process. **Figure 2.13** shows the traces of the same fractures in **Figure 2.12** subjected to rotation to reflect the existence of a dome structure.

Ivanova (1998) describes details regarding the fracture translation procedure as well as fracture rotation. Other geologic conditions where these processes apply are also discussed extensively.

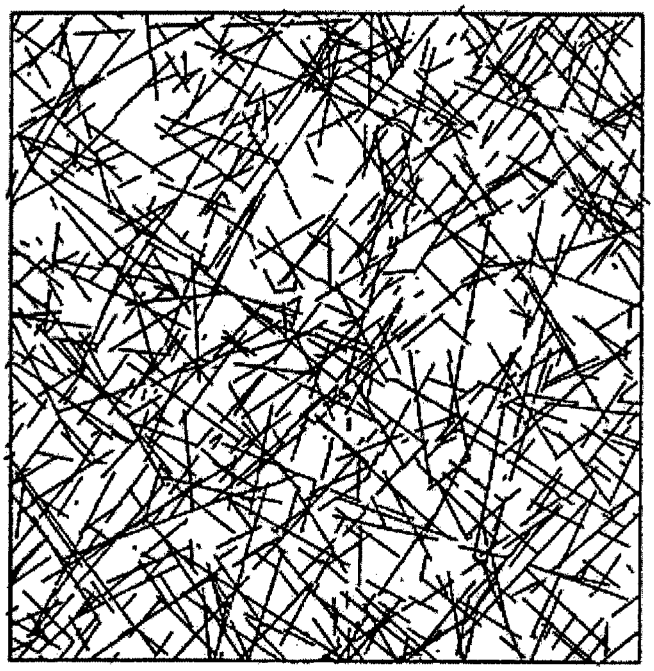


Figure 2.12 – Trace outcrop of fractures with orientations related to the general stress field directions (from Ivanova, 1998)

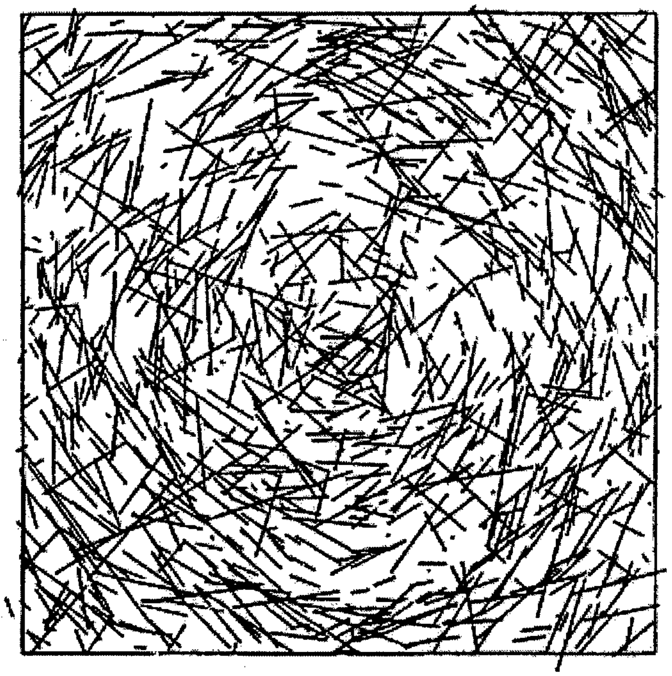


Figure 2.13 – Trace outcrop of the fractures from Figure 2.12 with their orientations also related to a circular dome structure (from Ivanova, 1998)

3 Procedure for Obtaining the Intersection Lengths between Fractures

3.1 Overview

The original version of GEOFRAC has also been modified by Meyer to include the capability of assembling the fractures into networks. The current endeavor aims to calculate the intersection lengths between the fractures generated by GEOFRAC. The processes involving the generation of planes and lines as well as the marking procedure remain as described above. New data organization and calculation procedures will be presented in the following sections in order to determine the fracture intersection geometry.

After a plane is generated within the modeling volume, the algorithm calculates the coordinates of the vertices of the polygon formed by the intersection. The Poisson lines are generated on the plane and the coordinates of the intersection points between the lines are determined. The coordinates of the points are organized and stored in such a way that it will be easy to access and use them in the next step, polygon assembly. In the polygon assembly process, the vertices are grouped into sets that define the fractures. When the fracture polygons have been assembled, the parameters needed to apply the marking procedure can be obtained. As discussed in **Chapter 2**, this procedure discards polygons with properties that do not comply with user-defined limits and the remaining polygons are then translated in proportion to their sizes. The most important new step is the next one in which the intersection between two fracture polygons is calculated. Meyer (1999) extended the capabilities of GEOFRAC to include the ability to divide the entire fracture network into smaller sub-networks. Meyer then studied the connectivity of the individual sub-networks.

3.2 Intersection between the Plane and the Modeling Volume

The first step in finding the vertices of the fracture polygons is to find the vertices of the plane on which the Poisson lines will be generated. The vertices formed by the line tessellation will be computed every time a new line is formed on the plane.

For clarity, the equation of the plane presented in **Chapter 2** will be repeated here. In the frame of reference of the fracture set, the equation of a plane is given by:

$$x\sin\theta\sin\phi + y\cos\theta\sin\phi + z\cos\phi = d$$

Equation 2.3

The global form of this equation is:

$$\begin{aligned}&(\cos\Theta\sin\theta\sin\phi + \sin\Theta\cos\Phi\cos\theta\sin\phi + \sin\Theta\sin\Phi\cos\phi)X \\ &+ (\cos\Theta\cos\Phi\cos\theta\sin\phi + \cos\Theta\sin\Phi\cos\phi - \sin\Theta\sin\theta\sin\phi)Y \\ &+ (\cos\Phi\cos\phi - \sin\Phi\cos\theta\sin\phi)Z = d\end{aligned}$$

Equation 3.1

For convenience, the above equation will be written as:

$$aX + bY + cZ = d$$

Equation 3.2

For a modeling volume that is cubical in shape **(Figure 3.1)**, the six bounding planes are given by:

$$\begin{aligned} X &= X_m \quad Y = Y_m \quad Z = Z_{box} \\ X &= -X_m \quad Y = -Y_m \quad Z = 0 \end{aligned}$$

Substituting each of these values into **Equation 3.2**, the following are obtained:

Intersections with planes **$X = X_m$** and **$X = -X_m$**

$$bY + cZ = d \mp aX_m$$

Equation 3.3

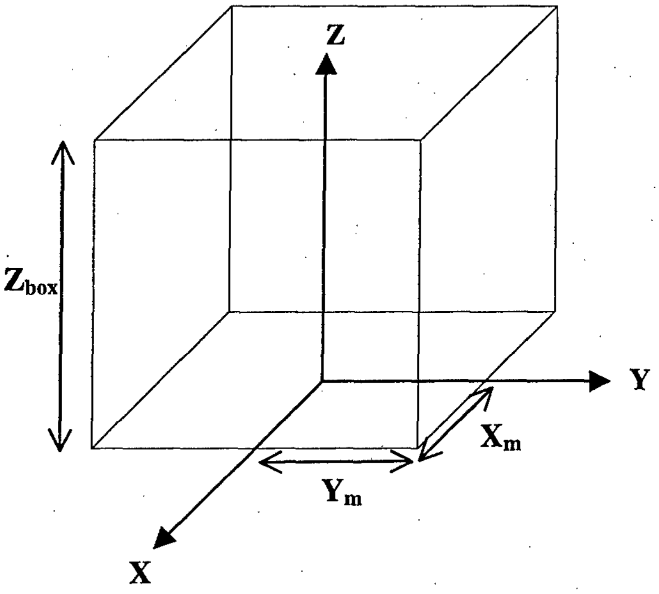


Figure 3.1 – Global axes and modeling volume

Intersections with planes **$Y = Y_m$** and **$Y = -Y_m$**

$$aX + cZ = d \mp bY_m$$

Equation 3.4

Intersections with planes **$Z = Z_{box}$** and **$Z = 0$**

$$aX + bY = d - cZ_{box}$$

$$aX + bY = d$$

Equation 3.5

In **Equation 3.3**, substitutions for Y or Z can be made. $Y = Y_m$ or $Y = -Y_m$ can be plugged in and corresponding values of **Z** are solved for. **$Z = Z_{box}$** or **$Z = 0$** can be substituted and values of Y can be solved for. These values should satisfy either **($0 \le Z \le Z_{box}$)** or **($-Y_m \le Y \le Y_m$)** depending on what is solved for (the point is within the modeling volume). For example, in **Figure 3.2**, the vertices V_2 and V_3 are found on the line of intersection **(LOI)** between the plane and the **$X = X_m$** boundary. When the substitutions $Y = Y_m$ or $Y = - Y_m$ are made into the equation of the **LOI**, Z_2 or Z_3 are found and are both within the modeling volume.

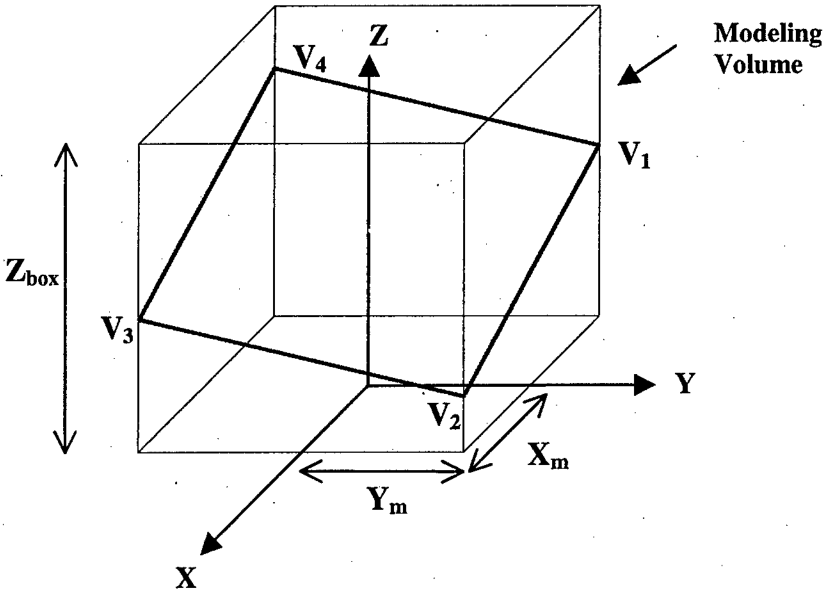


Figure 3.2 – Modeling volume with a generated plane

Equation 3.4 and **Equation 3.5** are used to find the other vertices in the same manner, imposing the limiting values on the solved **X** or **Z** in **Equation 3.4** and on **X** or **Y** in **Equation 3.5**. The fact that the vertices of the plane will always be located along the edges of the cubical modeling region makes it easier to locate them. At least two of the three coordinates of a vertex will be equal to the **X**, **Y**, or **Z** values at the edges. For example, V_4 has coordinates **($-X_m$, $-Y_m$, Z_4)**, V_1 has **($-X_m$, Y_m, Z_1)** and so on. A special case where all three of the coordinates of a vertex are equal to the values at the edges is when the plane intersects a corner of the modeling volume.

Figure 3.3 shows another mode of intersection. Here, the plane intersects the modeling volume on the **Z = 0** and **Z = Z_{box}** planes. The plane in **Figure 3.2** will also intersect the **Z = 0** and **Z = Z_{box}** planes but the intersection will not occur within the modeling volume.

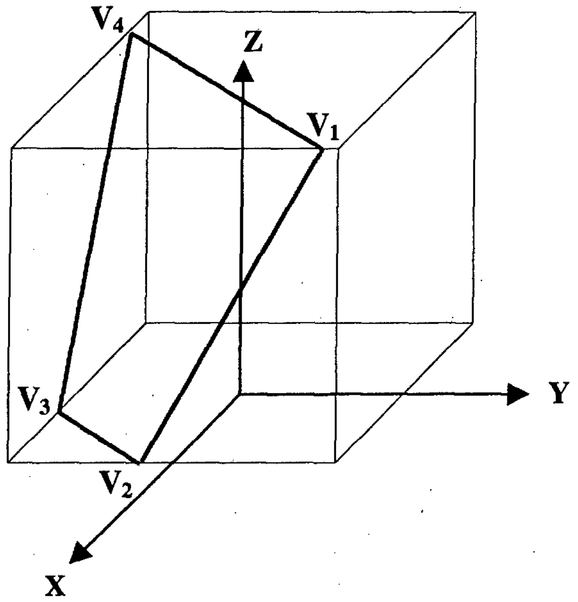


Figure 3.3 – Possible mode of intersection between the plane and the modeling volume

Figure 3.4 shows that planes can intersect the modeling volume on all of the boundaries. In this case, there will be "valid" coordinates that satisfy all six equations, **Equation 3.3** to **Equation 3.5**.

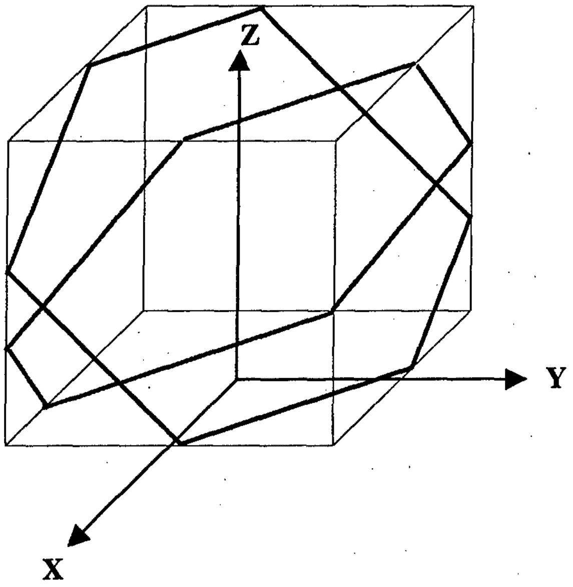


Figure 3.4 – Two intersecting generated planes

3.3 Poisson Line Intersection Calculations and Data Organization

The equation of a line in the frame of reference of the plane **(O'x'y'z')** is:

$$x'\cos\alpha + y'\sin\alpha = D$$

Equation 2.4

Where α and **D** are shown in **Figure 3.5**.

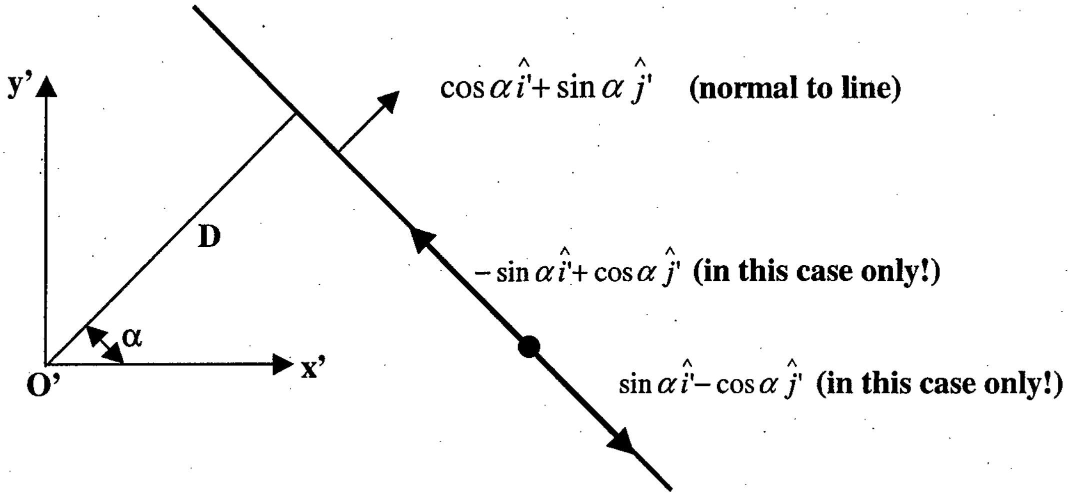

y'
$\cos\alpha\,\hat{i}' + \sin\alpha\,\hat{j}'$ (normal to line)
D
$-\sin\alpha\,\hat{i}' + \cos\alpha\,\hat{j}'$ (in this case only!)
α
O'
x'
$\sin\alpha\,\hat{i}' - \cos\alpha\,\hat{j}'$ (in this case only!)

Figure 3.5 – Generated Poisson line with the normal vector and the direction vectors

The intersection point between two lines with α, **D** and α_1, $\mathbf{D_1}$ is given by:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \dfrac{\sin\alpha_1}{\sin\alpha(\cot\alpha\sin\alpha_1 - \cos\alpha)} & \dfrac{1}{\cos\alpha - \cot\alpha\sin\alpha_1} \\ \dfrac{\cos\alpha_1}{\cos\alpha(\tan\alpha\cos\alpha_1 - \sin\alpha)} & \dfrac{1}{\sin\alpha - \tan\alpha\cos\alpha_1} \end{bmatrix} \begin{bmatrix} D \\ D_1 \end{bmatrix}$$

Equation 3.6

The coordinates of an intersection point are checked to see if it is indeed within the modeling volume. If it is not, then it is discarded. The valid points can then be arranged in a matrix as shown in **Table 3.1**. All points in one row belong to a single line; therefore, the line segments (which are also sides of the polygons) that lie on this line can be determined from the points found on this row. The diagonals are shown dashed.

This is a symmetric matrix. Some of the cells will be vacant because of the possibility of two lines being parallel or of the lines intersecting somewhere outside of the modeling volume.

The vectors for a line should also be stored as information. The vector for the line is defined only by the angle α in **Equation 2.4**. The vector normal to the line is $\cos\alpha\hat{i}'+\sin\alpha\hat{j}'$ so the line itself will have either $-\sin\alpha\hat{i}'+\cos\alpha\hat{j}'$ or $\sin\alpha\hat{i}'-\cos\alpha\hat{j}'$ as unit vectors depending on which direction the line is traversed. This information can also be stored as data in the matrix in **Table 3.1**.

For each point of intersection, there are usually four adjacent points, two along each of the lines intersecting at that point. For example, in **Figure 3.6** point **A** is the point of intersection between the lines defined by α_0, $\mathbf{D_0}$ and α_1, $\mathbf{D_1}$. In the matrix shown in **Table 3.1**, point **A** corresponds to **(x'$_0$, y'$_0$)**. The points adjacent to **A** are **B$_2$**, **B$_9$**, **B$_6$**, and **B$_8$**.

Line	α_0, D$_0$	α_1, D$_1$	α_2, D$_2$	α_3, D$_3$	α_4, D$_4$	α_5, D$_5$	α_6, D$_6$	α_7, D$_7$	α_8, D$_8$
α_0, D$_0$	------	x'$_0$, y'$_0$	x'$_1$, y'$_1$		x'$_2$, y'$_2$		x'$_3$, y'$_3$	x'$_4$, y'$_4$	
α_1, D$_1$		------		x'$_5$, y'$_5$		x'$_6$, y'$_6$		x'$_7$, y'$_7$	x'$_8$, y'$_8$
α_2, D$_2$			------		x'$_9$, y'$_9$	x'$_{10}$, y'$_{10}$		x'$_{11}$, y'$_{11}$	
α_3, D$_3$				------		x'$_{12}$, y'$_{12}$	x'$_{13}$, y'$_{13}$		x'$_{14}$, y'$_{14}$
α_4, D$_4$					------		x'$_{15}$, y'$_{15}$	x'$_{16}$, y'$_{16}$	x'$_{17}$, y'$_{17}$
α_5, D$_5$			SYMMETRIC			------	x'$_{18}$, y'$_{18}$		x'$_{19}$, y'$_{19}$
α_6, D$_6$							------	x'$_{20}$, y'$_{20}$	x'$_{21}$, y'$_{21}$
α_7, D$_7$								------	x'$_{22}$, y'$_{22}$
α_8, D$_8$									------

Table 3.1 – Matrix of intersection point coordinates. This matrix is symmetric. A blank cell means no intersection occurs between the lines or if the intersection occurs outside the modeling volume.

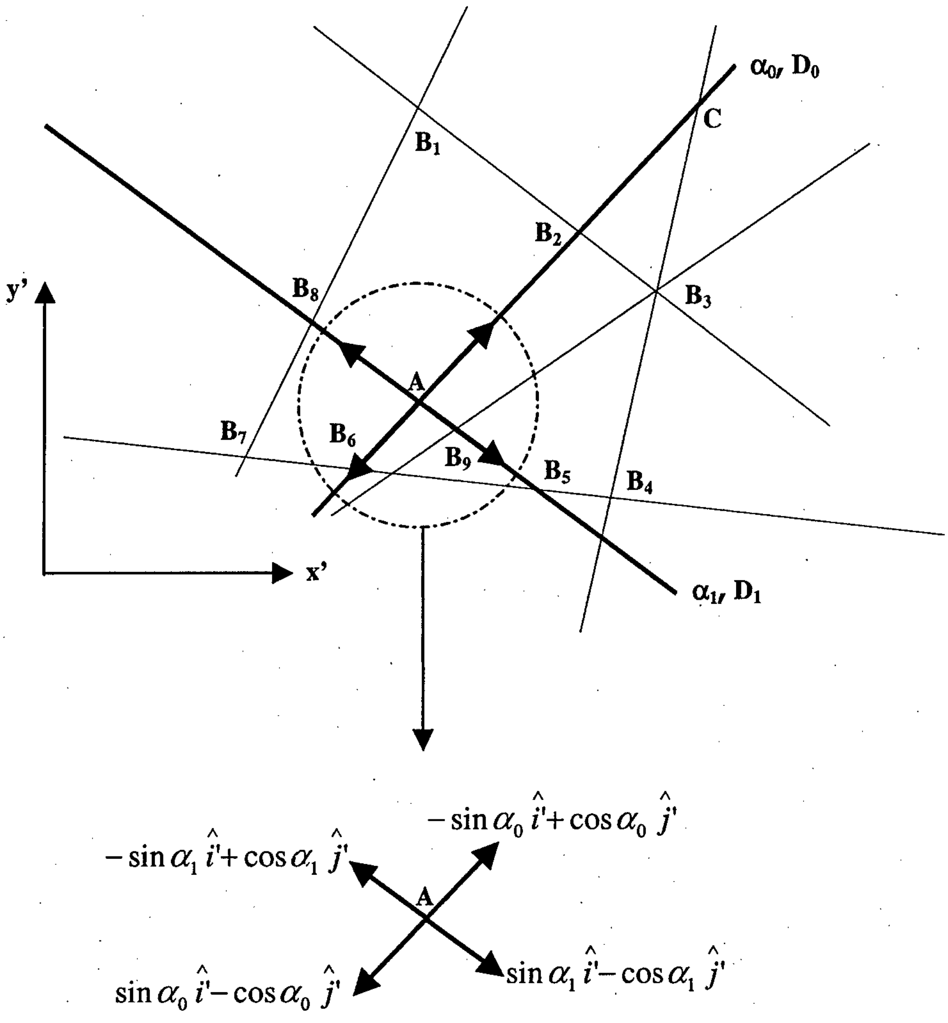

α0, D0
C
B1
B2
B3
B8
y'
A
B7
B6
B9
B5
B4
x'
α1, D1
$-\sin\alpha_0 \hat{i}' + \cos\alpha_0 \hat{j}'$
$-\sin\alpha_1 \hat{i}' + \cos\alpha_1 \hat{j}'$
A
$\sin\alpha_0 \hat{i}' - \cos\alpha_0 \hat{j}'$
$\sin\alpha_1 \hat{i}' - \cos\alpha_1 \hat{j}'$

Figure 3.6 – Points adjacent to point A and the vectors to each of these points

It can be seen that all the points contained in the row α0, **D0** in **Table 3.1** should be tested in order to get **B2** and **B6** in **Figure 3.6**. In order to do this, the vectors that radiate away from point **A** will be required. These are as follows:

$$-\sin\alpha_0 \hat{i}' + \cos\alpha_0 \hat{j}'$$

Equation 3.7

$$\sin\alpha_0 \hat{i}' - \cos\alpha_0 \hat{j}'$$

Equation 3.8

$$-\sin\alpha_1 \hat{i}' + \cos\alpha_1 \hat{j}'$$

Equation 3.9

$$\sin\alpha_1\,\hat{i}'-\cos\alpha_1\,\hat{j}'$$

Equation 3.10

For the points along the α_0, **D₀** line, the following tests are done on the coordinates of each intersection point **(x'ᵢ, y'ᵢ)** found in the row (for the line defined by **(α_0, D₀)**, **(x'ᵢ, y'ᵢ)** may be one of the following points: **(x'₁, y'₁)**, **(x'₂, y'₂)**, **(x'₃, y'₃)**, **(x'₄, y'₄)** and so on):

$$\frac{(x'_i-x'_0)}{-\sin\alpha_0}=\frac{(y'_i-y'_0)}{\cos\alpha_0}=k>0?$$

Equation 3.11

or

$$\frac{(x'_i-x'_0)}{\sin\alpha_0}=\frac{(y'_i-y'_0)}{-\cos\alpha_0}=k>0?$$

Equation 3.12

The vector $(x'_i-x'_0)\hat{i}'+(y'_i-y'_0)\hat{j}'$ is parallel to both the unit vectors in **Equation 3.7** and **Equation 3.8** and is a scalar multiple of these vectors. If the scalar multiplier k is **negative**, the point is located in the **opposite direction** and if it is **positive** then it points in the **same direction** as the unit vector. For example, if the coordinates of the point **B₂** are used as the pair **(x'ᵢ, y'ᵢ)** in **Equation 3.11**, the resulting value of k would be positive since the vector in **Equation 3.11** points from **A** to **B₂**. The tests in **Equation 3.11** and **Equation 3.12** basically show which side of point **(x'₀, y'₀)** the point **(x'ᵢ, y'ᵢ)** is located. The points with the smallest positive value of k for each of these two tests are selected as the adjacent points located on each side of **(x'₀, y'₀)**.

This procedure is repeated using the direction vectors of the other intersecting line **α_1, D₁** in order to locate the adjacent points **B₈** and **B₉**. The direction vectors are given by **Equation 3.9** and **Equation 3.10** (in the direction of **B₈** and **B₉**, respectively). Instead of the tests in **Equation 3.11** and **Equation 3.12**, the following will now be used:

$$\frac{(x'_i-x'_0)}{-\sin\alpha_1}=\frac{(y'_i-y'_0)}{\cos\alpha_1}=k>0?$$

Equation 3.13

or

$$\frac{(x'_i - x'_0)}{\sin\alpha_1} = \frac{(y'_i - y'_0)}{-\cos\alpha_1} = k > 0?$$

Equation 3.14

indicating the directions that the line α_1, **D_1** will be traversed in order to find **B_8** and **B_9**.

So the procedure can be visualized as taking a point in the matrix and testing each point in the same row and the same column to find the adjacent points.

As a matter of efficiency, it would be good if the algorithm does not repeat the process on points already considered in previous runs. In **Figure 3.6** for example, while finding the adjacent points for **A** (namely **B_2**, **B_9**, **B_6**, and **B_8**) the algorithm will also identify **A** as an adjacent point to each of these points so that when it is time to find the points adjacent to say **B_2**, there is no need to search for **A** again.

The information has to be stored in a form that is easy to process for the next procedure. The data are stored in sets containing the point of intersection between two (or more) lines, the points adjacent to it and the components of the unit vectors to get to these points. All these data are part of the computation process described above.

The set of data for a given point of intersection, say **(x'_0, y'_0)** (point **A** in **Figure 3.6**) between the lines defined by α_0, **D_0** and α_1, **D_1** will look like this:

Point	**A (starting pt.)**	**B_2**	**B_6**	**B_8**	**B_9**
Coordinates	**(x'_0, y'_0)**	**(x'_{24}, y'_{24})**	**(x'_{25}, y'_{25})**	**(x'_{26}, y'_{26})**	**(x'_{27}, y'_{27})**
Direction		**$<-\sin\alpha_0, \cos\alpha_0>$**	**$<\sin\alpha_0, -\cos\alpha_0>$**	**$<-\sin\alpha_1, \cos\alpha_1>$**	**$<\sin\alpha_1, -\cos\alpha_1>$**

Table 3.2 – Adjacent point data for point A

Each set is identified by its starting point, in this case **A**. The points adjacent to **A** are described by their **x'** and **y'** coordinates as well as the direction of the vector from **A**. These data will be necessary in the process of assembling the polygons.

The set may have more data if more than two lines intersect at the given point. The set defining point **B_3** in **Figure 3.6** will have lines radiating from it in six different directions defined by the directions of the three different lines intersecting there.

The algorithm proceeds to assemble the fracture polygons.

3.4 Polygon Assembly Procedure

3.4.1 Polygon Assembly

The assembly of polygons is done point by point and in a clockwise fashion (**Figure 3.7**). At this stage, it is necessary to create a procedure to avoid "**straying.**" Straying can be defined by referring to **Figure 3.7**. Say, if the polygon **AB$_2$B$_3$B$_9$A** is to be assembled and the process begins at **A**. The algorithm traverses the line defined by α_0 and **D$_0$** towards **B$_2$**. At **B$_2$**, the algorithm has four choices: proceed to **C**, turn counter-clockwise to **B$_1$**, turn back towards **A** or turn clockwise toward **B$_3$.** If the algorithm chooses any of the **first three** options, then it is "**straying.**" The algorithm should change its direction from **AB$_2$** to **B$_2$B$_3$**. The **valid** change in direction (clockwise in our case) from the one currently being traversed is the greatest among the available changes in direction at the point of intersection but less than 2π radians.

The first step is to choose a point to start from (the **starting point**) and assemble the polygons that have this point as a common vertex. Say point **A** in **Figure 3.7** is chosen. From **A**, four directions can be traversed one at a time depending on which polygon will be assembled. At this point, the data set in **Table 3.2** is accessed and one of the four vectors is chosen as the initial direction that will be traversed.

If the polygon **AB$_2$B$_3$B$_9$A** will be assembled, the first direction taken will be $-\sin\alpha_0\,\hat{i}' + \cos\alpha_0\,\hat{j}'$ which is parallel to the line segment **AB$_2$**. The line segment **AB$_2$** is stored in the data set created for this polygon. At **B$_2$**, there are four directions to choose from and since the polygon formation is done in a clockwise manner, **B$_3$** should be the next point. **B$_3$** will also have its own data set containing all points adjacent to it as well as the directions to these points and this set will contain six adjacent points and six directions because three lines intersect there. The same is done at **B$_9$**. At each point, the maximum possible change in direction has to take place but it will be limited to less than 2π radians clockwise as discussed above. The cross product between the current direction and each of the possible directions at the point are taken. Only the cross products that result in a negative **z'** component are valid meaning that only vectors whose direction is clockwise from the current one are taken.

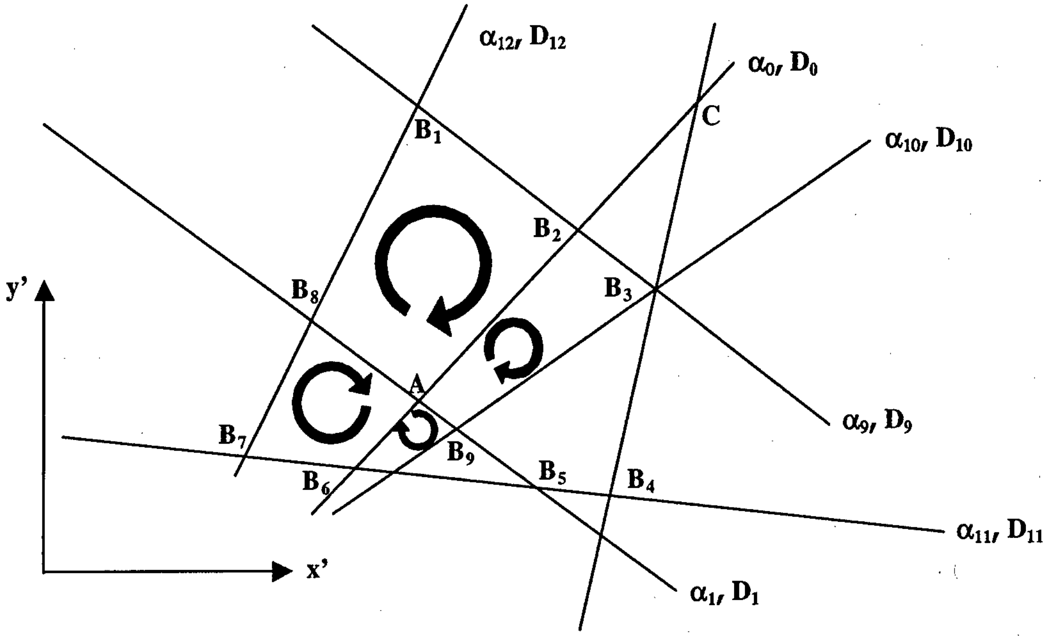


Figure 3.7 – Clockwise assembly of polygons using A as a starting point

For two vectors **E** and **F** where

$$E = e_1 \hat{i}' + e_2 \hat{j}' + e_3 \hat{k}'$$

Equation 3.15

$$F = f_1 \hat{i}' + f_2 \hat{j}' + f_3 \hat{k}'$$

Equation 3.16

The cross-product, $E \times F$, is given by:

$$E \times F = (e_2 f_3 - e_3 f_2)\hat{i}' + (e_3 f_1 - e_1 f_3)\hat{j}' + (e_1 f_2 - e_2 f_1)\hat{k}'$$

Equation 3.17

So in this case, **E** would be the current direction and **F** would be any one of the possible directions in the set. The restriction would be for $(e_1 f_2 . e_2 f_1)$ to be less than **zero** for the change to be clockwise (using the right-hand rule, this would point into the paper) from the current direction. A counter-clockwise change would have a $(e_1 f_2 . e_2 f_1)$ value greater than zero.

After the undesirable directions have been weeded out, the angles between the current vector and each of the valid vectors are taken. Note that in **Equation 3.15** to **Equation 3.17**, anything that has a subscript "3" will be zero because the lines are expressed in 2-D space. The dot product between the two vectors is used to find the clockwise angle between them. Since the vectors only have two components (in the **x'** and **y'** directions):

$$\beta = \cos^{-1}(e_1 f_1 + e_2 f_2) \quad where \quad 0 < \beta < 2\pi$$

Equation 3.18

The valid point with the maximum β (and less than 2π) will be the next point in the assembly procedure. Actually, this check need not be performed if the point is an intersection between only two lines. The cross-product check will suffice because there will only be one direction out of the four that will pass. Therefore, the angle check (greatest but less than 2π radians clockwise) should be done only if more than two lines intersect at the point such as $\mathbf{B_3}$ in **Figure 3.7.**

This process is repeated until **A** is hit and a polygon is formed.

The process is repeated for the other three directions (to $\mathbf{B_8}$, $\mathbf{B_6}$, and $\mathbf{B_9}$) thus, three other polygons are formed. In the case of having $\mathbf{B_3}$ as a starting point, six polygons are formed.

However, it will be necessary to avoid repeating the assembly of a polygon once another vertex is taken as a starting point. The solution to this problem will be discussed next.

3.4.2 Procedure for Avoiding Repetitive Polygon Assembly

In this algorithm, it is possible to avoid repeating the assembly of a polygon once another vertex is taken as a starting point. For example, going back to **Figure 3.7**, say the polygon assembly is finished for **A** and the procedure starts again but this time it takes $\mathbf{B_8}$ as the starting point. It is not necessary to repeat the polygons $\mathbf{AB_8B_1B_2A}$ and $\mathbf{AB_6B_7B_8A}$ since they have already been formed when **A** was taken as the starting point. Therefore, great savings in calculation will be achieved if certain directions involved in previous assemblies are somehow marked against future consideration. In the case where $\mathbf{B_8}$ is the starting point, it will be more efficient to neglect the vectors pointing towards **A** and $\mathbf{B_1}$ in order to prevent the re-assembly the two polygons just mentioned above.

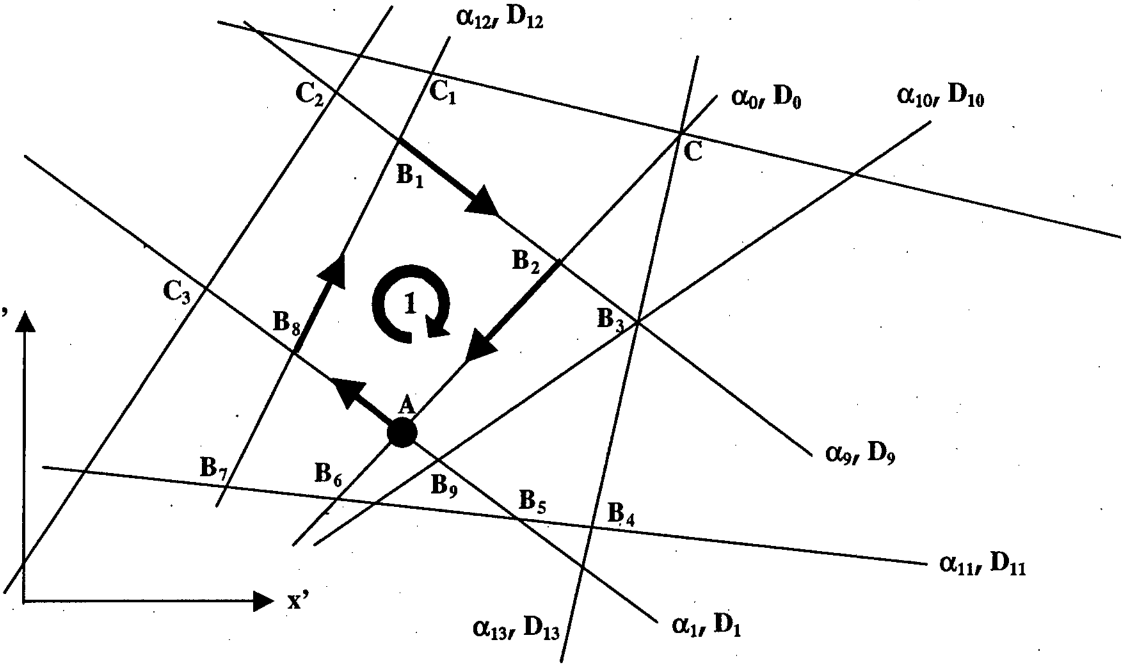


Figure 3.8 – Clockwise formation of Polygon 1 from point A

The number of times a polygon is assembled is equal to the number of vertices it has because **each of these vertices can be starting points** for polygon assembly. Polygon **$AB_8B_1B_2A$** above (referred to as polygon **1** for brevity) can be assembled when any one of its four vertices is made a starting point.

The way to avoid re-assembly is to discard the vectors that have already been used in one assembly. In other words, the number of initial directions to choose from at a starting point will be reduced if an **adjacent** point has been used as a starting point in a previous polygon assembly. For example, when **A** is taken as a starting point (**Figure 3.8**) then one of the polygons assembled will be polygon **1**. The figure shows the arrows of **valid** directions at the vertices. Recall from the section on Polygon Assembly that validity of the direction is checked by calculating the **z'** component of the cross product between the current direction and the other directions at the point. These directions will be discarded and they will no longer be available when **B_8**, **B_1**, or **B_2** are taken as starting points. Therefore, the creation of polygon **1** can no longer be initiated from these points.

Point	**A** (starting pt.)	**B_2**	**B_6**	**B_8**	**B_9**
Coordinates	**(x'_0, y'_0)**	**(x'_{24}, y'_{24})**	**(x'_{25}, y'_{25})**	**(x'_{26}, y'_{26})**	**(x'_{27}, y'_{27})**
Direction		**$<-\sin\alpha_0, \cos\alpha_0>$**	**$<\sin\alpha_0, -\cos\alpha_0>$**	**$<-\sin\alpha_1, \cos\alpha_1>$**	**$<\sin\alpha_1, -\cos\alpha_1>$**

Table 3.3 – Adjacent point data for point A

Point	B_8 (starting pt.)	B_1	B_7	C_3	A
Coordinates	(x'_{26}, y'_{26})	(x'_{28}, y'_{28})	(x'_{29}, y'_{29})	(x'_{30}, y'_{30})	(x'_0, y'_0)
Direction		$<-\sin\alpha_{12},\cos\alpha_{12}>$	$<\sin\alpha_{12},-\cos\alpha_{12}>$	$<-\sin\alpha_1,\cos\alpha_1>$	$<\sin\alpha_1,-\cos\alpha_1>$

Table 3.4 – Adjacent point data for point B_8

Point	B_1 (starting pt.)	C_1	B_8	C_2	B_2
Coordinates	(x'_{28}, y'_{28})	(x'_{31}, y'_{31})	(x'_{26}, y'_{26})	(x'_{32}, y'_{32})	(x'_{24}, y'_{24})
Direction		$<-\sin\alpha_{12},\cos\alpha_{12}>$	$<\sin\alpha_{12},-\cos\alpha_{12}>$	$<-\sin\alpha_9,\cos\alpha_9>$	$<\sin\alpha_9,-\cos\alpha_9>$

Table 3.5 – Adjacent point data for point B_1

Point	B_2 (starting pt.)	C	A	B_1	B_3
Coordinates	(x'_{24}, y'_{24})	(x'_{33}, y'_{33})	(x'_0, y'_0)	(x'_{28}, y'_{28})	(x'_{34}, y'_{34})
Direction		$<-\sin\alpha_0,\cos\alpha_0>$	$<\sin\alpha_0,-\cos\alpha_0>$	$<-\sin\alpha_9,\cos\alpha_9>$	$<\sin\alpha_9,-\cos\alpha_9>$

Table 3.6 – Adjacent point data for point B_2

To make clear what is written above, the steps are now discussed in more detail. Turning to **Figure 3.8** and **Table 3.3** with **A** as the starting point, to form polygon **1** the vector pointing to **B_8** is taken initially (assembly is done clockwise). At point **B_8**, the possible directions are checked. The data in **Table 3.4** are processed and the valid direction will be the one that points to **B_1**. At **B_1** and subsequently, **B_2**, the same is done. It can be observed that if **B_8** is the starting point, there are four polygons that can be formed resulting from the four different directions that can be traversed initially (to **B_1**, to **B_7**, to **C_3**, and to **A**). Polygon **1** is formed when the direction to **B_1** is taken initially. In order to avoid this repetition, the data in column **B_1** in **Table 3.4** should be deleted when polygon **1** is formed the first time around (when **A** was the starting point). This results in a reduction of the number of available initial directions once **B_8** is made the starting point. So with **A** as the starting point, the data in column **B_8** in **Table 3.3**, column **B_1** in **Table 3.4**, column **B_2** in **Table 3.5** and column **A** in **Table 3.6** will be deleted after the assembly of polygon **1**.

Table 3.7 to **Table 3.9** show the modified data sets for points **B_8**, **B_1**, and **B_2** after the formation of polygon **1** using **A** as the starting point. Looking at **Figure 3.8** and these new data sets, it can be observed that if any of these three points are eventually made the starting point, none of the initial directions will cause the formation of polygon **1**. These have already been eliminated when **1** was formed for the first time (when **A** was the starting point). Remember that polygon assembly is done in a clockwise manner.

The deletion process continues as other points are taken as starting points. Less and less data need to be handled as the process continues.

Point	B_8 (starting pt.)	B_7	C_3	A
Coordinates	(x'_{26}, y'_{26})	(x'_{29}, y'_{29})	(x'_{30}, y'_{30})	(x'_0, y'_0)
Direction		$<\sin\alpha_{12},-\cos\alpha_{12}>$	$<-\sin\alpha_1,\cos\alpha_1>$	$<\sin\alpha_1,-\cos\alpha_1>$

Table 3.7 – Remaining adjacent points to B_8 after formation of polygon 1

Point	B_1 (starting pt.)	C_1	B_8	C_2
Coordinates	(x'_{28}, y'_{28})	(x'_{31}, y'_{31})	(x'_{26}, y'_{26})	(x'_{32}, y'_{32})
Direction		$<-\sin\alpha_{12},\cos\alpha_{12}>$	$<\sin\alpha_{12},-\cos\alpha_{12}>$	$<-\sin\alpha_9,\cos\alpha_9>$

Table 3.8 – Remaining adjacent points to B_1 after formation of polygon 1

Point	B_2 (starting pt.)	C	B_1	B_3
Coordinates	(x'_{24}, y'_{24})	(x'_{33}, y'_{33})	(x'_{28}, y'_{28})	(x'_{34}, y'_{34})
Direction		$<-\sin\alpha_0,\cos\alpha_0>$	$<-\sin\alpha_9,\cos\alpha_9>$	$<\sin\alpha_9,-\cos\alpha_9>$

Table 3.9 – Remaining adjacent points to B_2 after formation of polygon 1

Figure 3.9 shows the remaining initial directions for the three points after the formation of polygon **1** with **A** as the starting point. Note that no initial direction at any of the three points will cause the formation of polygon **1**.

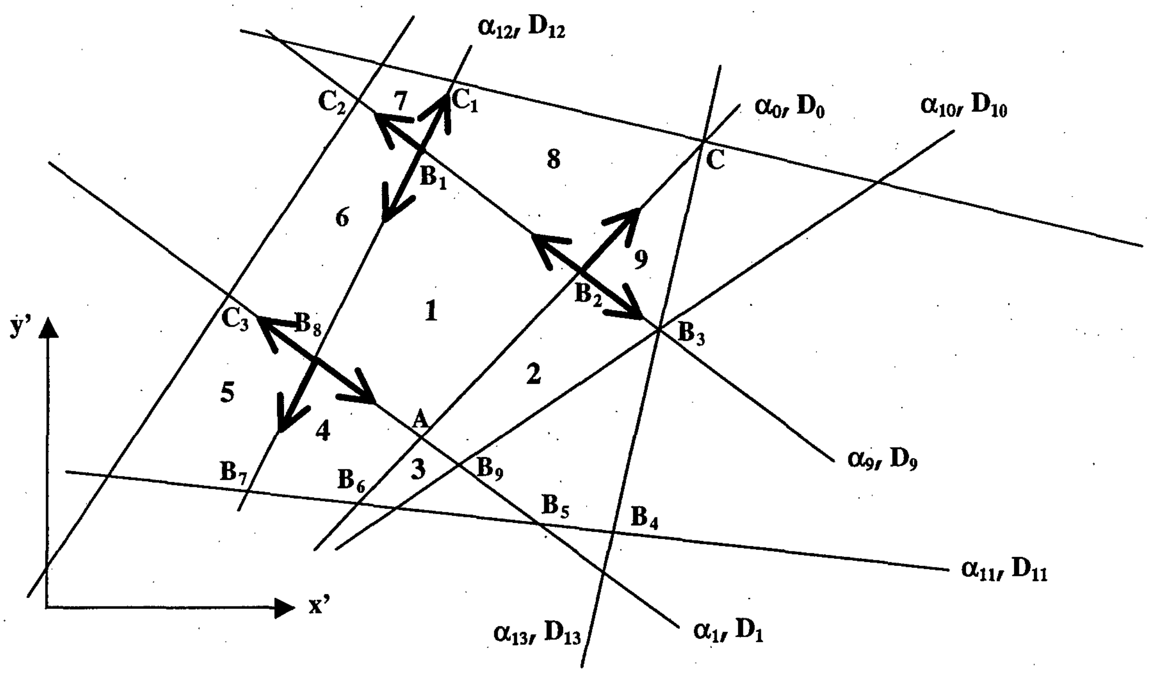


Figure 3.9 – Remaining initial directions for B_8, B_1 and B_2 after the formation of polygon 1

At **A**, the polygons **1**, **2**, **3**, and **4** will be formed. This will further reduce the initial directions for $\mathbf{B_2}$ and $\mathbf{B_8}$. **Figure 3.10** shows the initial directions after **all** the polygons at A are formed.

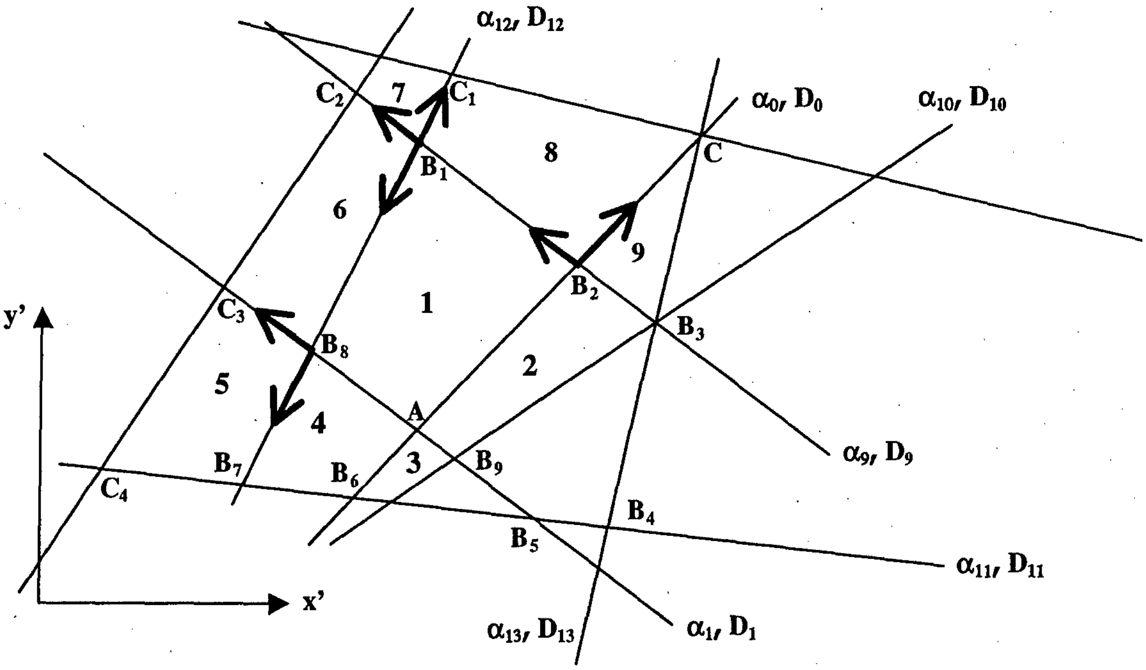


Figure 3.10 – Remaining valid directions for B_8, B_1 and B_2 after the formation of polygons 1, 2, 3 and 4

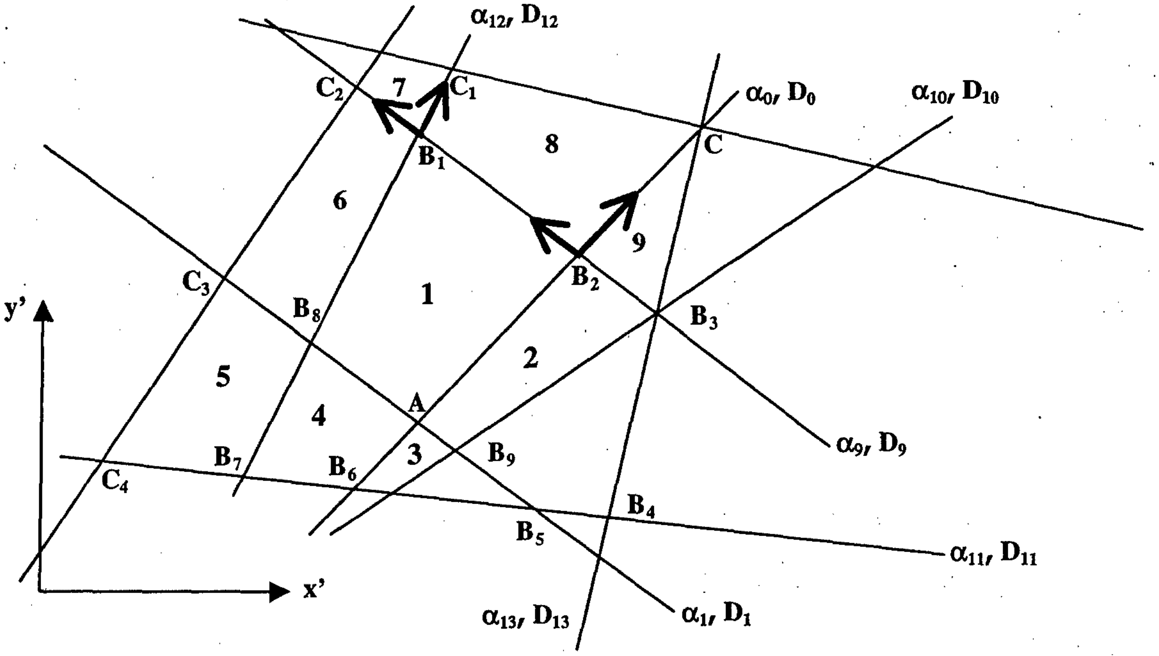


Figure 3.11 – Remaining directions for B_1 and B_2

So at $\mathbf{B_8}$, the vectors that can form polygons **1** and **4** have already been removed. Only polygons **5** and **6** can be formed there. At $\mathbf{B_2}$, the vectors needed to form polygons **1** and **2** have also been removed leaving the vectors necessary for the formation of **8** and **9**. After $\mathbf{B_8}$ is made the starting point and all the remaining polygons there have been formed, the directions at $\mathbf{B_1}$ as well as those of **B**$_7$, **C**$_4$, **C**$_3$, and **C**$_2$ will be reduced because of the assembly of polygons **5** and **6**. **Figure 3.11** shows the remaining initial directions for **B**$_1$. These allow only the formation of **7** and **8**. As more and more points are considered as starting points, more and more directions at the adjacent points are deleted.

3.5 Marking Procedure and Polygon Translation

After the polygons have been formed, the areas, the coordinates of the centers (x'_c, y'_c) and the elongation of the polygons are calculated in order to apply the marking rule. To calculate the area and the coordinates of the center, the following equations are used. These are taken from Meyer (1999).

$$Area = \frac{1}{2}(x'_1\, y'_2 + \ldots + x'_{noP}\, y'_1 - y'_1\, x'_2 - \ldots - y'_{noP}\, x'_1)$$

Equation 3.19

$$x'_c = \frac{(x'_1 + x'_2 + \ldots + x'_{noP})}{noP}$$

$$y'_c = \frac{(y'_1 + y'_2 + \ldots + y'_{noP})}{noP}$$

Equation 3.20

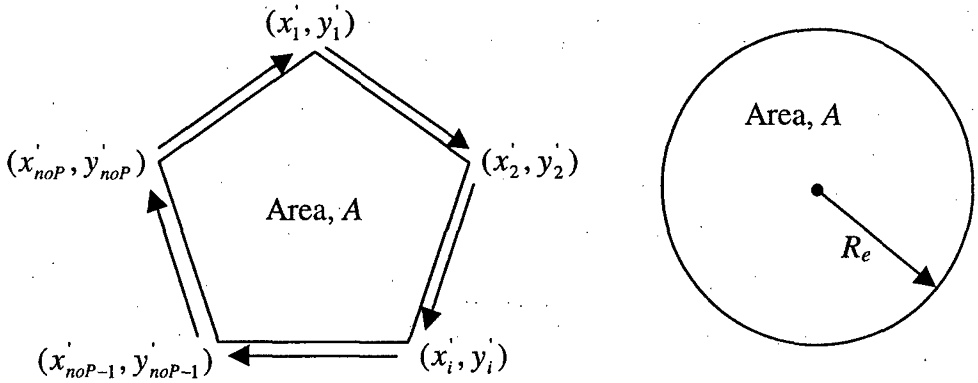


Figure 3.12 – Computation of the center of a fracture polygon and its equivalent radius

The computations are done in the frame of reference of the plane. The equivalent radius can also be computed and is given by:

$$R_e = \left(A/\pi\right)^{1/2}$$

Equation 3.21

The elongation ***e*** is also calculated.

$$e = \frac{R_{\max}}{R_e}$$

Equation 3.22

where $\mathbf{R_{max}}$ is the maximum distance from the center of the polygon to any one of its vertices. For any vertex, the distance to the center of the polygon is

$$R = \sqrt{(x'-x'_c)^2 + (y'-y'_c)^2}$$

Equation 3.23

A typical requirement for being a "good" polygon is that the angle between two adjacent sides is **at least 60 degrees** (see Ivanova, 1998). Since the coordinates of the vertices are already known, the vectors parallel to the line segments connecting them can be determined. Using the dot product between two vectors, the angle **δ** between the adjacent sides can be calculated.

$$\delta = 180 - \beta = 180 - \cos^{-1}\|E \cdot F\| = 180 - \cos^{-1}(e_1 f_1 + e_2 f_2)$$

Equation 3.24

The vectors **E** and **F** define the current direction and the valid available direction respectively at a point (see **Figure 3.13**). With this information, the marking procedure can be performed.

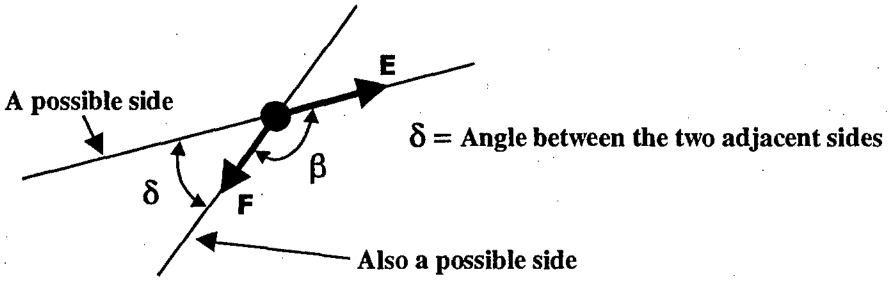


Figure 3.13 – Vectors E and F which define the current and valid available direction, respectively

After the marking procedure is completed, the "good" fractures (fractures that have been retained) are translated along the **z'** axis (see **Figure 2.10** in page 23) a distance **dz'_{max}**. Only the **z'** coordinates of the vertices will be changed (see **Figure 3.14** below).

$$(x', y', z') \Rightarrow (\underline{x}, \underline{y}, \underline{z} = z' \pm dz'_{\max})$$

Equation 3.25

but **x'** = $\underline{x}$, **y'** = $\underline{y}$ and **z' = 0** because the all vertices lie on the **x'-y'** plane, so the coordinates become

$$(x', y', \pm dz_{\max})$$

After the translation, the equation of the plane containing the fracture becomes

$$x \sin\theta \sin\phi + y \cos\theta \sin\phi + z \cos\phi = d \pm dz_{\max}$$

Note that θ and ϕ do not change with translation.

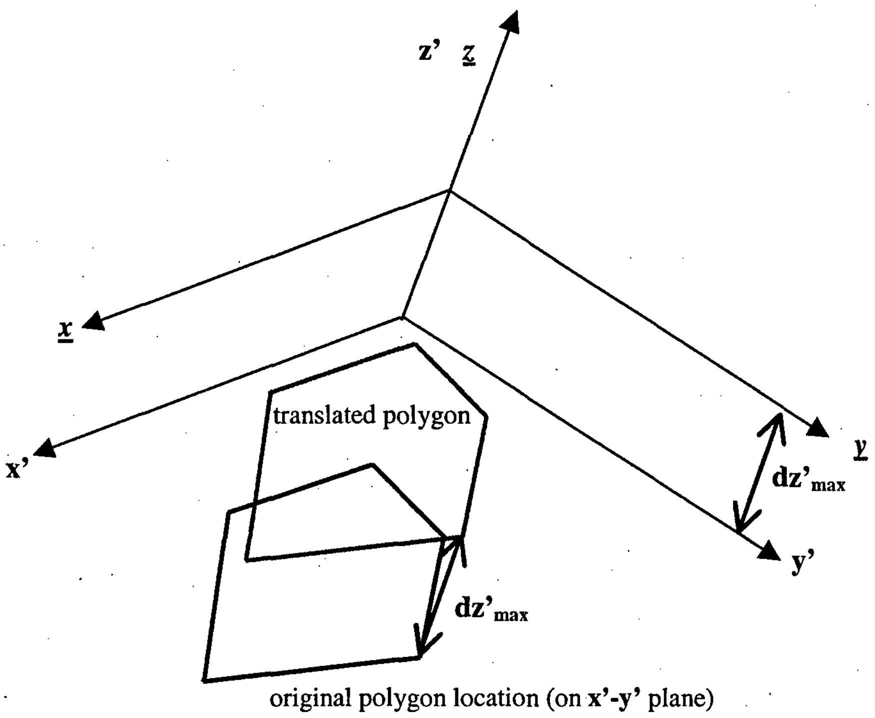


Figure 3.14 – Translation of a polygon by an amount dz'$_{max}$

3.6 Calculating the Intersection Length between Two Fractures

3.6.1 Determining Valid Intersection Points

After the marking and translation procedures, the remaining polygons are tested for intersection. In this procedure, a single polygon is taken and all the other polygons in the modeling volume are tested against it. The geometry and orientation of the intersection for each pair of intersecting fracture polygons is then calculated. Since the coordinates of all the vertices of each polygon have been determined in the polygon assembly procedure, the equation of the plane on which each polygon lies can be determined. The equations of the planes are then used to determine the equations of the line of intersection (**LOI**) between two planes containing a fracture polygon. The line segment of intersection between two intersecting fracture polygons lies on the **LOI** between the planes on which they lie **(Figure 3.15)**. The problem, however, lies in the calculation of the endpoints of the line segment of intersection between the two fracture polygons. In **Figure 3.15**, if all of the sides of the polygons **S** and **T** are extended, they will intersect the **LOI** (dashed lines). However, not all of these intersection points will be "**valid**" (the intersection point between the side that has been extended and the **LOI** has to be between the two endpoints of that side) so the "**validity**" of each point must be checked.

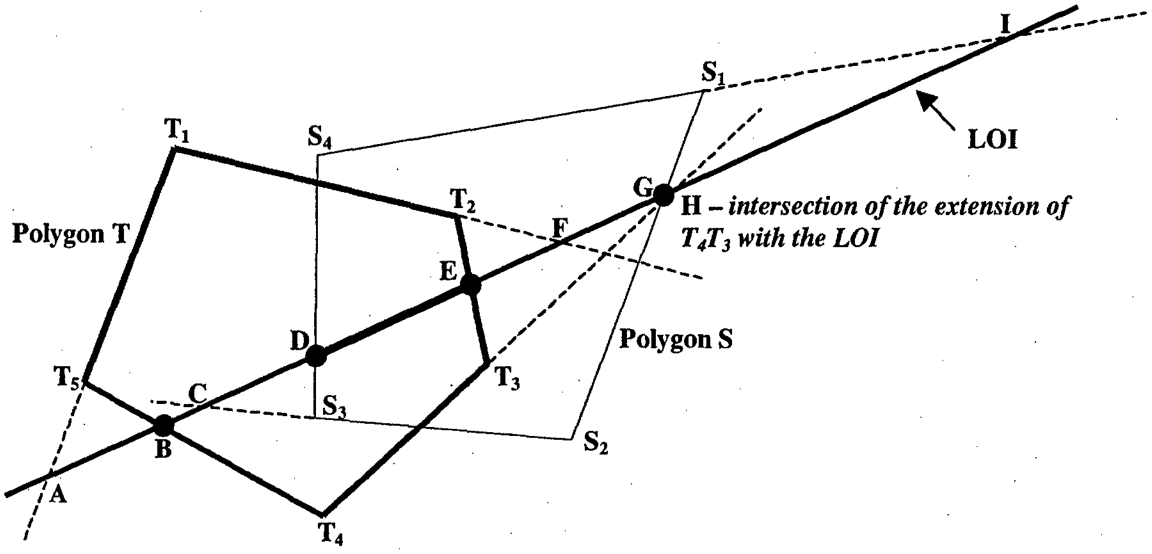


Figure 3.15 – Intersecting polygons with sides extended to intersect the line of intersection

The first thing to do is to locate the intersection points ***A***, ***B***, ***C***, ***D***, ***E***, ***F***, ***G***, ***H*** and ***I***. The data available are the coordinates of each vertex. Therefore, the global form of the equations of a line containing three of these points that are collinear (for example: **T$_1$**, **T$_2$** and **F**) can be obtained. Also, the global forms of the equations for the **LOI** are necessary. To obtain the equations for the **LOI**, the fact that the fractures have been translated and/or rotated should be taken into consideration.

The original equation of the plane containing a fracture in the frame of reference of the fracture set is:

$$x\sin\theta\sin\phi + y\cos\theta\sin\phi + z\cos\phi = d$$

Equation 2.3

After the polygons are assembled and the marking procedure performed, the polygons are translated a distance **dz'$_{max}$**. The equation of the plane becomes:

$$x\sin\theta\sin\phi + y\cos\theta\sin\phi + z\cos\phi = d \pm dz'_{\max}$$

Equation 3.26

where

$$dz'_{\max} = C\frac{(E[R'_e])^2}{R'_e}$$

Equation 2.5

In the global coordinate system, this equation would be:

$$\begin{gathered}(\cos\Theta\sin\theta\sin\phi + \sin\Theta\cos\Phi\cos\theta\sin\phi + \sin\Theta\sin\Phi\cos\phi)X \\ + (\cos\Theta\cos\Phi\cos\theta\sin\phi + \cos\Theta\sin\Phi\cos\phi - \sin\Theta\sin\theta\sin\phi)Y \\ + (\cos\Phi\cos\phi - \sin\Phi\cos\theta\sin\phi)Z = d \pm dz'_{\max}\end{gathered}$$

Equation 3.27

For simplicity, it will be written as:

$$aX + bY + cZ = d \pm dz'_{\max}$$

Equation 3.28

So for two planes with equations

$$a_i X + b_i Y + c_i Z = d_i \pm dz'_{i\max}$$

Equation 3.29

and

$$a_k X + b_k Y + c_k Z = d_i \pm dz'_{k\max}$$

Equation 3.30

The following equations will define the **LOI** of the two planes in the global frame of reference:

$$Z = \frac{(a_i b_k - a_k b_i)X - [(d_i \pm dz'_{i\max})b_k - (d_k \pm dz'_{k\max})b_i]}{(c_k b_i - c_i b_k)}$$

Equation 3.31

and

$$Z = \frac{(a_k b_i - a_i b_k)Y - [(d_i \pm dz'_{i\max})a_k - (d_k \pm dz'_{k\max})a_i]}{(c_k a_i - c_i a_k)}$$

Equation 3.32

It is helpful that the data resulting from the polygon assembly procedure will be in the form of clockwise-sequentially arranged points. For example, polygon **T** in **Figure 3.15** will have its points arranged in this manner:

Point	T_1	T_2	T_3	T_4	T_5
Coordinates	(x'_1, y'_1)	(x'_2, y'_2)	(x'_3, y'_3)	(x'_4, y'_4)	(x'_5, y'_5)

Table 3.10 – Clockwise arrangement of vertices of polygon T

Note that the coordinates in **Table 3.10** are still in the local frame of reference of the fracture plane. Recall that in the polygon assembly procedure, work is done in the **local frame of the fracture** because it is much more convenient to do it that way (**z'** is zero since all the vertices lie on the **x'-y'** plane). To calculate the **LOI**, two fractures must be in the same frame of reference so in order to proceed, a transformation of coordinates from the local to the global system is necessary. **Table 3.11** shows the coordinates in **Table 3.10** in global coordinates.

Point	T_1	T_2	T_3	T_4	T_5
Coordinates	(X_1, Y_1, Z_1)	(X_2, Y_2, Z_2)	(X_3, Y_3, Z_3)	(X_4, Y_4, Z_4)	(X_5, Y_5, Z_5)

Table 3.11 – Global coordinates of vertices of polygon T

The coordinates in **Table 3.11** are then substituted in **Equation 3.31** and **Equation 3.32** to find out if any of the vertices fall exactly on the **LOI**. In **Figure 3.16**, the vertex $\mathbf{T_2}$ lies on the **LOI** and it is evident that the equations of the lines containing the line segments $\mathbf{T_1T_2}$ and $\mathbf{T_2T_3}$ need not be derived for the calculation of their intersection points with the **LOI**. Since the points are stored in the manner shown in **Table 3.11** it will be easy to find out which vertices are adjacent to the vertex that lies on the **LOI**. In this case, $\mathbf{T_1}$ and $\mathbf{T_3}$ are the adjacent vertices to $\mathbf{T_2}$ so only the equations of the lines containing the line segments $\mathbf{T_3T_4}$, $\mathbf{T_4T_5}$ and $\mathbf{T_5T_1}$ need to be derived to get the other three intersection points. Two of which are the intersections between the **LOI** and the dashed lines in **Figure 3.16** and the other is located between $\mathbf{T_4}$ and $\mathbf{T_5}$. In the case where two of the vertices lie on the **LOI** then there will be no need for further calculations for that fracture since all the fractures are convex and will therefore have only two points lying on the **LOI**.

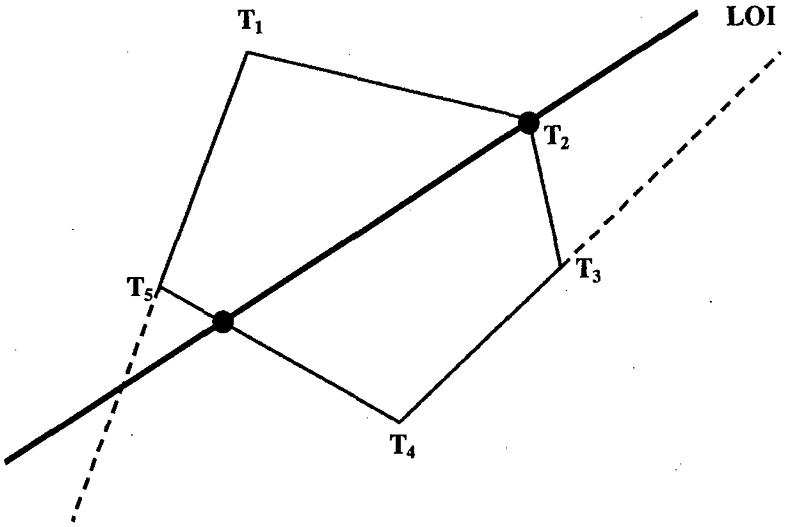


Figure 3.16 – Polygon T with two sides extended to intersect the line of intersection

For lines whose equations need to be reconstructed to obtain the intersection point, the coordinates of the vertices in the local frame of the plane can be converted to their corresponding values in the global frame. These values can then be used to construct the parametric equations of the line. For two points **T_i (X_i, Y_i, Z_i)** and **T_k (X_k, Y_k, Z_k)**, the line connecting them is defined by the following equations:

$$\begin{aligned} X &= (1-t)X_i + tX_k \\ Y &= (1-t)Y_i + tY_k \\ Z &= (1-t)Z_i + tZ_k \\ \textit{where} \quad & -\infty < t < +\infty \end{aligned}$$

Equation 3.33

Z_i and **Z_k** are substituted in the third equation and the parameter **t** is solved for. The resulting expression for **t**, which is in terms of **Z**, is substituted back into the first two. Two equations for **Z**, one in terms of **X** and the other in terms of **Y**, are obtained from the substitution. These are equated to **Equation 3.31** and **Equation 3.32** (the equations defining the **LOI**), respectively, and the values of **X** and **Y** are calculated. These are the **X** and **Y** coordinates of the intersection point between the **LOI** and the line segment connecting **T_i (X_i, Y_i, Z_i)** and **T_k (X_k, Y_k, Z_k)**. The **Z** coordinate of the intersection point can be computed by substituting the calculated **X** in **Equation 3.31** or by substituting **Y** in **Equation 3.32.**

For example, the **X** and **Y** coordinates of the intersection point **B** (see **Figure 3.15**) between the line connecting the vertices **T_4** and **T_5** and the **LOI** are given below:

$$X_B = \frac{N_1(Z_4X_5 - X_4Z_5) + M_1(X_5 - X_4)}{L_1(X_5 - X_4) - N_1(Z_5 - Z_4)}$$

Equation 3.34

$$Y_B = \frac{N_2(Z_4Y_5 - Y_4Z_5) + M_2(Y_5 - Y_4)}{L_2(Y_5 - Y_4) - N_2(Z_5 - Z_4)}$$

Equation 3.35

where

(X_4, Y_4, Z_4) are the global coordinates of point T_4
(X_5, Y_5, Z_5) are the global coordinates of point T_5

and L_1, M_1 and N_1 are defined as

$$L_1 = a_i b_k - a_k b_i$$
$$M_1 = (d_i \pm dz'_{i\max})b_k - (d_k \pm dz'_{k\max})b_i$$
$$N_1 = c_k b_i - c_i b_k$$

while L_2, M_2 and N_2 are defined as

$$L_2 = a_k b_i - a_i b_k$$
$$M_2 = (d_i \pm dz'_{i\max})a_k - (d_k \pm dz'_{k\max})a_i$$
$$N_2 = c_k a_i - c_i a_k$$

a_i, b_i, c_i, d_i and a_k, b_k, c_k, d_k are the parameters that define the **LOI** in **Equation 3.31** and **Equation 3.32** between the two planes containing the intersecting fractures.

Z_B can be calculated by substituting **X_B** into **Equation 3.31** or by substituting **Y_B** into **Equation 3.32**. Both will give the same value for **Z_B**.

Once the locations of the points ***A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*** and ***I*** in **Figure 3.15** have been computed, a checking procedure is needed to filter out points ***A*, *C*, *F*, *H*** and ***I***, the intersection points that do not actually belong to any fracture. This procedure is discussed in the following paragraphs. After the filtering procedure, another process (which will be presented in **Section 3.6.2**) is needed to determine that line segment ***DE*** is the intersection of the two fractures.

After the points of intersection between each side of the polygon and the **LOI** are calculated, each of them has to be tested for validity (if the point is between two vertices like point ***B*** in **Figure 3.15**). In **Figure 3.15**, it will help to recognize that the vector pointing from ***B*** to **T_4** has a **direction opposite** that of the vector pointing from ***B*** to **T_5** and that ***B*** is a "**valid**" point of intersection. For an "**invalid**" (one that does not actually belong to the polygon) point of intersection such as ***I***, the vectors pointing from this point to each of the vertices S_1 and S_4 have the **same direction**. The vectors and the relationships between them for the points discussed are shown below. These equations essentially outline the procedure for checking the points of intersection.

Note that $\hat{i}$ is the vector (1,0,0), $\hat{j}$ is (0,1,0) and $\hat{k}$ is (0,0,1).

For ***B***

$$(X_{T_4} - X_B)\hat{i} + (Y_{T_4} - Y_B)\hat{j} + (Z_{T_4} - Z_B)\hat{k} = K\left[(X_{T_5} - X_B)\hat{i} + (Y_{T_5} - Y_B)\hat{j} + (Z_{T_5} - Z_B)\hat{k}\right]$$

Equation 3.36

and for ***I***.

$$(X_{S_1} - X_I)\hat{i} + (Y_{S_1} - Y_I)\hat{j} + (Z_{S_1} - Z_I)\hat{k} = K_1\left[(X_{S_4} - X_I)\hat{i} + (Y_{S_4} - Y_I)\hat{j} + (Z_{S_4} - Z_I)\hat{k}\right]$$

Equation 3.37

Note that **K** and **K_1** are constants. In this test, the magnitude of these two constants is not important, what matters is the **sign** of these constants. A **negative** constant will indicate a valid point of intersection and a **positive** constant will mean otherwise. For the valid point ***B***, the value of K in **Equation 3.36** is negative since the vector from **T_4** to ***B*** and the vector from **T_5** to ***B*** have opposite directions. This means that ***B*** is between **T_4** and **T_5** and the side **T_4T_5** of polygon **T** in **Figure 3.15** intersects the **LOI**. For the point ***I***, the value of **K_1** is positive since the vector from **S_1** to ***I*** and the vector from **S_4** to ***I*** have the same directions. This implies that point ***I*** is not located between the vertices **S_1** and **S_4** and that the side **S_1S_4** do not intersect the **LOI**.

After each point of intersection is calculated, it is subjected to the test just described. Invalid points are discarded since it is useless to store them. Valid points are stored along with the vertices of the polygon they belong to. Such identification will be useful in the following stage.

3.6.2 Algorithm for Calculating the Intersection Length

The next step is to determine which of these points make up the line of intersection if there is one. In this case the points ***D*** and ***E*** in **Figure 3.15** need to be isolated from the set containing the valid points ***B***, ***D***, ***E*** and ***G***. Bear in mind that it is not sufficient to go through the procedure discussed above in order to determine whether or not two fractures intersect. So far, only the intersection points between the **LOI** and the fractures have been calculated.

If each fracture has two intersection points with the **LOI**, three cases may be anticipated.

1. As shown in **Figure 3.15**, the two fractures overlap but neither one is completely contained in the other along the **LOI**. A special case would be a point intersection between the two fractures. See also **Figure 3.17a**.

2. The fractures intersect and one is contained completely in the other along the **LOI**. See **Figure 3.17b.**

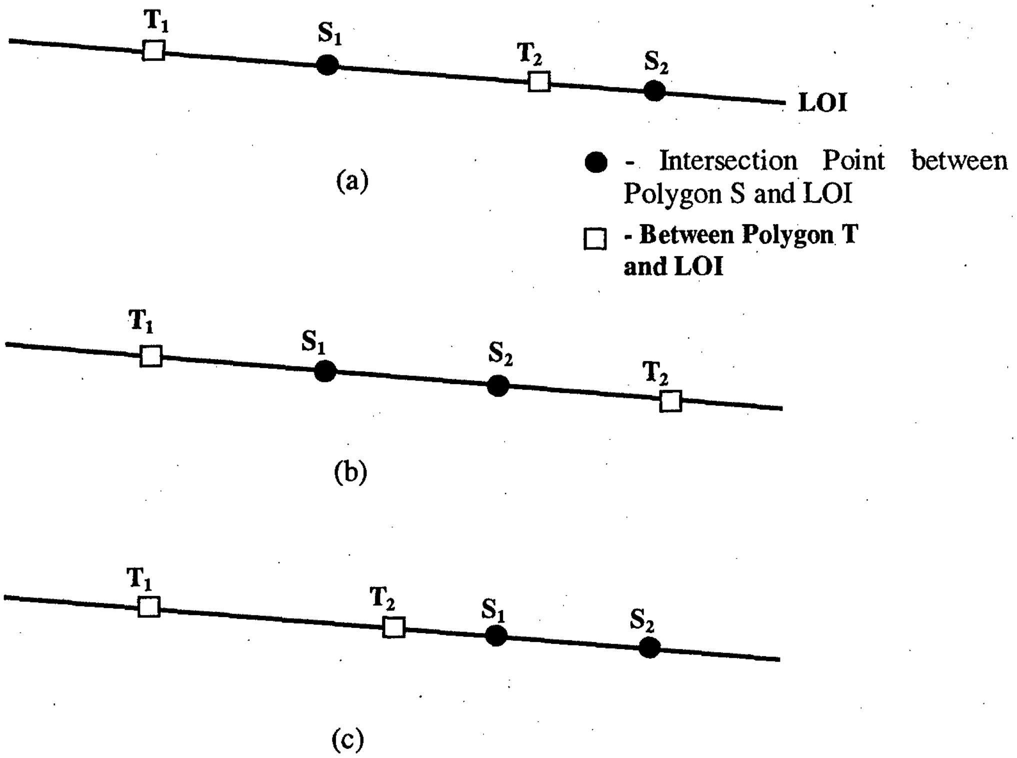
3. The fractures do not intersect at all. See **Figure 3.17c**.

At this stage, a single reference point is first chosen among the four intersection points. Any one of the four points will do. Using a check on the directions of the vectors pointing to the other three points, those three remaining points are grouped according to which side they are located with respect to the reference point. There can only be two cases for this:

1. All the other three points are located on one side of the reference point.

2. One is located on one side and the other two are on the other side.

Figure 3.17a to **Figure 3.17c** show how the four points may be arranged along the **LOI**. If any **one** of the outer points in any **one** of the three arrangements in **Figure 3.17** is chosen as the reference point, then case 1 is encountered (example: if **T_1** is chosen in **Figure 3.17a**, the other points are on one side of **T_1**). On the other hand, if any **one** of the inner two points is chosen as the reference point then case 2 is encountered (example: if S_2 is chosen in **Figure 3.17b, T_1** will be on one side while S_2 and T_2 are on the other).



T1
S1
T2
S2
LOI
(a)
● - Intersection Point between Polygon S and LOI
□ - Between Polygon T and LOI
T1
S1
S2
T2
(b)
T1
T2
S1
S2
(c)

Figure 3.17 – The three likely arrangements of intersection points on the line of intersection

The algorithm for computing the intersection length between two fractures is shown in **Figure 3.18**. For example, the point **S_1** for the case in **Figure 3.17b** is chosen as the reference point. Two of the points (**S_2** and **T_2**) are on one side of **S_1** and the third (**T_1**) is on the other. The flowchart in **Figure 3.18** can be followed up to this stage ("**two points are on one side and one is on the other side**"). Now the algorithm decides if the single point (**T_1**) belongs to the same polygon as the reference point. But **T_1** is from a different polygon so the process proceeds to the step labeled "**calculate the distances to the other two points on the other side.**" The shorter calculated distance is taken as the length of intersection. In this case, the intersection between the **LOI** and the polygon **S** is completely contained in the polygon **T** (**Figure 3.17b**). The same path along the flowchart would be taken if the case in **Figure 3.17a** were considered but the two polygons will merely overlap and the line segment of intersection (**S_1T_2**) will not be completely contained in either fracture polygon. If **T_2** in **Figure 3.17c** were taken to be the reference point, the path would change at the decision stage labeled "**is that single point from the same polygon as the reference point.**" This case would lead to the decision "yes" and the algorithm determines that the fractures do not intersect (and indeed they do not).

As another example, the point **T_1** in **Figure 3.17a** is taken as the reference point. All the other three points are on one side of **T_1**. The distances from **T_1** to each these points (**S_1**, **T_2** and **S_2**) are calculated and then it is determined if the nearest point (**S_1**) is from the same polygon as the reference point. But **S_1** is from a different polygon so the algorithm proceeds to calculate the distances from this point, **S_1**, to the other two points **T_2** and **S_2**. The shorter of the two distances from **S_1** is the length of intersection. The same would happen if point **T_1** were chosen as the reference point in **Figure 3.17b** except that the shorter distance would now be from **S_1** to **S_2**. Now if **T_1** were chosen as the reference point for the case in **Figure 3.17c**, the algorithm would flow differently since the nearest point to **T_1** would be **T_2** and there would be no intersection.

If both polygons have single-point intersections with the **LOI** (**Figure 3.19**) then only the question of whether or not they coincide needs to be answered. This means that the global coordinates of the two points need to be equal. This situation, however, may seem highly unlikely.

If only one of the polygons has a single-point intersection with the **LOI** and the other has a two-point intersection, then whether or not the single point is between these two points has to be determined. In **Figure 3.20**, it has to be checked if point **C** is within the line segment **AB**. The vector pointing from **C** to **A** should have a direction opposite that of the vector pointing from **C** to **B** for intersection to occur.

So for intersection:

$$(X_A - X_C)\hat{i} + (Y_A - Y_C)\hat{j} + (Z_A - Z_C)\hat{k} = K\left[(X_B - X_C)\hat{i} + (Y_B - Y_C)\hat{j} + (Z_B - Z_C)\hat{k}\right]$$

Equation 3.38

and **K** should be **negative**. Otherwise, **C** is outside **AB** and no intersection will occur.

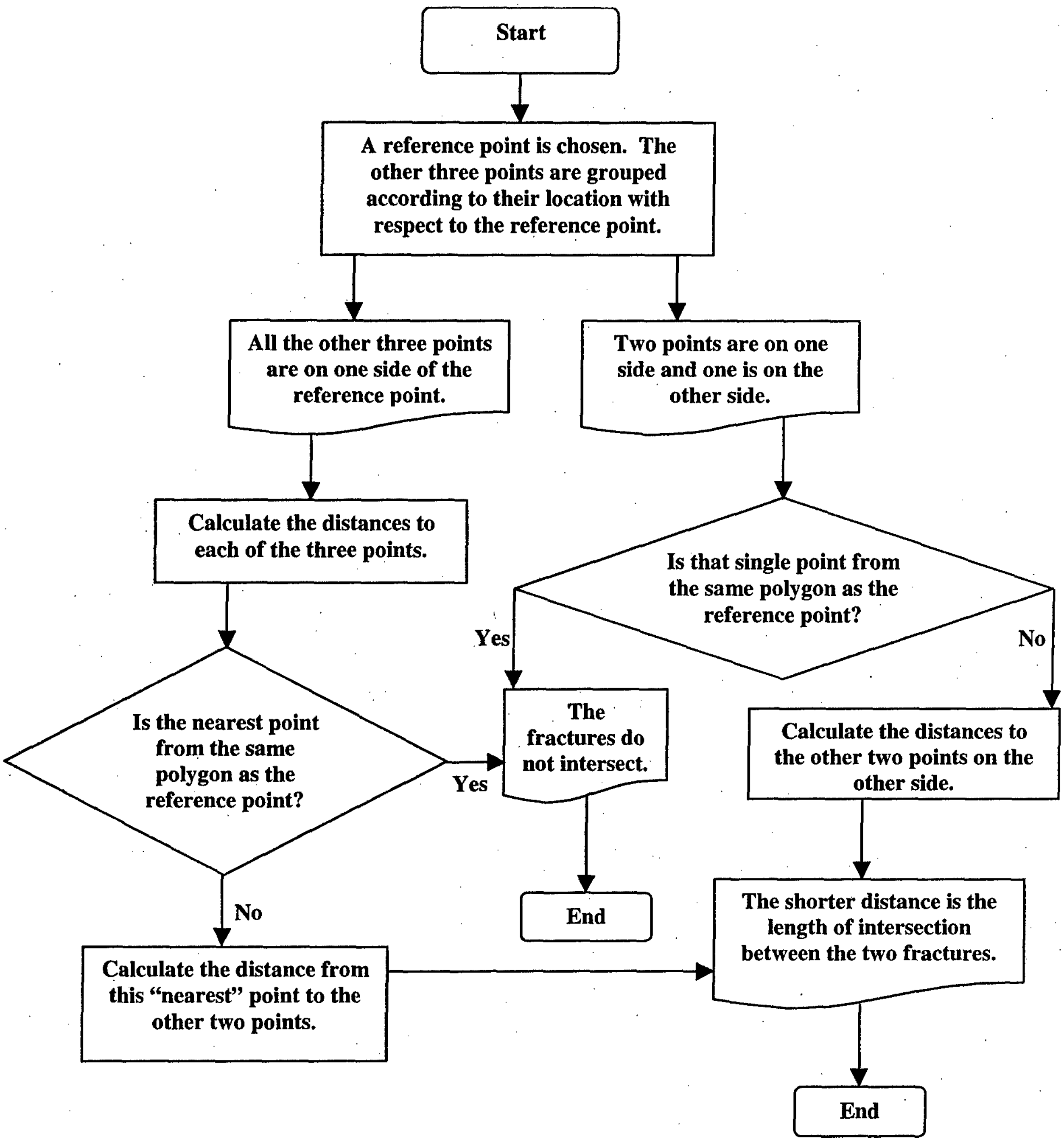

Start
A reference point is chosen. The other three points are grouped according to their location with respect to the reference point.
All the other three points are on one side of the reference point.
Two points are on one side and one is on the other side.
Calculate the distances to each of the three points.
Is that single point from the same polygon as the reference point?
Yes
No
Is the nearest point from the same polygon as the reference point?
Yes
The fractures do not intersect.
Calculate the distances to the other two points on the other side.
No
End
The shorter distance is the length of intersection between the two fractures.
Calculate the distance from this "nearest" point to the other two points.
End

Figure 3.18 – Flow chart for the algorithm for determining the line of intersection between two fractures

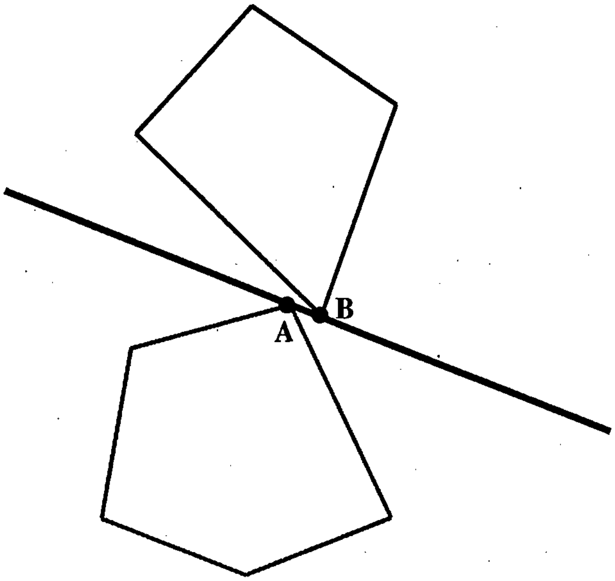


Figure 3.19 – Point intersections with the LOI, no intersection between fractures

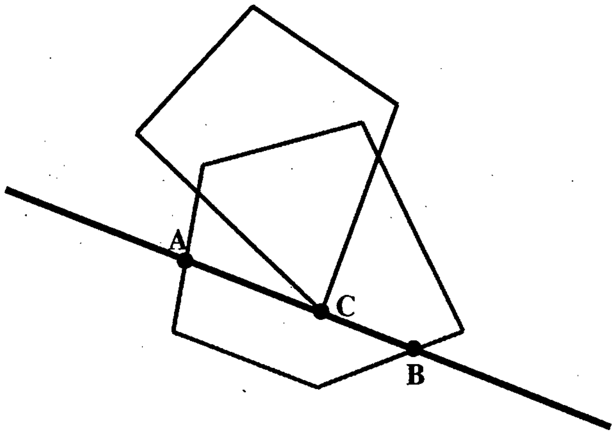


Figure 3.20 – Point intersection with the LOI, Point intersection between fractures

4 Fracture Intersection Geometry Study for the Boston Area

Meyer (1999) used the sub-networking capabilities of GEOFRAC to study the connectivity of the fracture networks in the Boston Area. Two modeling approaches were carried out. Regional modeling was performed in order to give a general idea of the appearance of the fracture networks relative to the faults in the area. Local modeling was performed to calibrate the model parameters and study the fracture network properties (sub-network size, connectivity). Through local modeling, Meyer found that sub-networks in the area exhibit more vertical connectivity in terms of physical extent compared to horizontal connectivity (i.e. sub-networks have $C_{8z} > C_{8x}$ and C_{8y}).

The modeling parameters were derived from fracture traces observed by Billings (1976) during the construction of drainage and water supply tunnels in the Boston Area. Meyer used sensitivity analyses to derive the best estimates of the fracture equivalent radius and fracture intensity (P_{32}) from tracelength and borehole spacing data, respectively. Using the best estimates of these parameters and the capability of modeling the individual fracture intersections, the goal is to study the geometry of the individual intersections.

4.1 Site Location and Geology

Figure 4.1 below illustrates the major geological features in the Boston Region. It shows the three major geological structures in the area: the Boston Basin, the Boston Platform and the Nashoba Thrust Belt.

The Boston Basin is composed of two lithologic units: the Roxbury Conglomerate and the Cambridge Argillite. The Roxbury Conglomerate is a pebble to cobble sized conglomerate interbedded with volcanic rocks (Meyer, 1999). The pebbles and cobbles range in size from 1 to 15 cm but sizes as large as 30 cm in diameter have been observed. The matrix of the typical conglomerate is a gray feldspathic sandstone (Billings, 1976). The Cambridge Slate, as the Cambridge Argillite is also known, is typically dark bluish gray or brownish gray, rather fine-grained and composed chiefly of argillaceous material (LaForge, 1932). It occupies almost all the northern part of the Boston Basin.

The Boston Platform occupies the western part of the state of Massachusetts as well as the entire state of Rhode Island (**Figure 4.1**). It consists of a late Precambrian batholithic complex and associated meta-sedimentary and meta-volcanic rocks that were intruded by plutons and covered by sediments at various times during the Paleozoic (Woodhouse et al., 1991). The composition of the late Precambrian batholithic complex ranges from quartz-rich alaskite to diorite or gabbro (Woodhouse et al., 1991).

The Nashoba Thrust Belt (shown as the Nashoba Block in **Figure 4.1**) is part of a folded, faulted, metamorphosed sequence between westward-dipping faults, the Clinton-Newbury fault zone and the Bloody Buff fault zone (Abu-Moustafa, 1976). It passes just to the northwest of Boston (**Figure 4.1**) and is composed of ferromagnesian rocks of hybrid origin into which several types of granite have been intruded (Meyer, 1999).

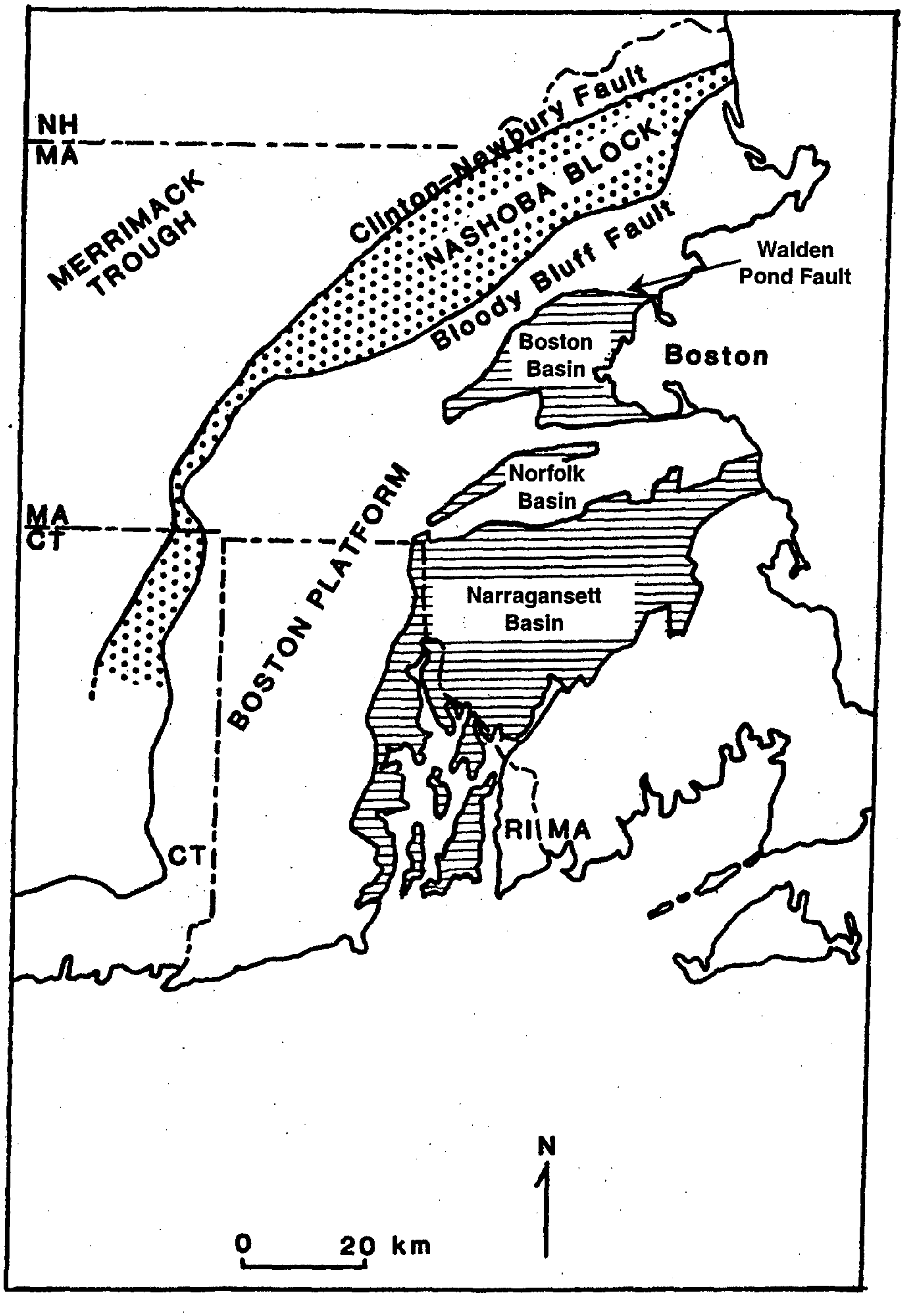


Figure 4.1 – Main geologic formations in the Boston region (from Meyer, 1999)

The next section briefly describes the regional modeling approach, and the derivation of the modeling parameters for the local modeling approach is also discussed.

4.2 Modeling Parameters for Intersection Geometry Study

Regional modeling was aimed at generating the general appearance of the fracture networks specifically in the Aberjona watershed and Superfund sites, which are located at the boundary of the Boston Basin and the Boston Platform (**Figure 4.1**). The boundary between the Boston Basin and Boston Platform is formed by the Walden Pond Fault. To the north lies another major fault, the Mystic Fault, which is part of the Bloody Buff Fault System. This represents the boundary between the Boston Platform and the Nashoba Thrust Belt. These two major discontinuities were included in the regional modeling approach. The orientations of the faults are given in **Table 4.1** below.

	Walden Pond Fault	Mystic Fault
Strike	N100°W	N135°W
Dip	55°N	55°NW

Table 4.1 – Strike and dip of major discontinuities considered in the modeling simulations in Figure 4.2

The Walden Pond Fault and the Mystic Fault are represented clearly in the regional modeling output (**Figure 4.2** and **Figure 4.3**). The fracture intensity in the Boston Basin was assumed to be twice the fracture intensity in the Boston Platform and the Nashoba Thrust Belt. This is to accommodate the observation made by Billings (1976) in the Malden tunnel, which strikes north-south, that the fracture trace spacing increases as the tunnel crosses the boundary between the Boston Basin and the Boston Platform. **Figure 4.3** demonstrates that GEOFRAC models this change in intensity quite nicely.

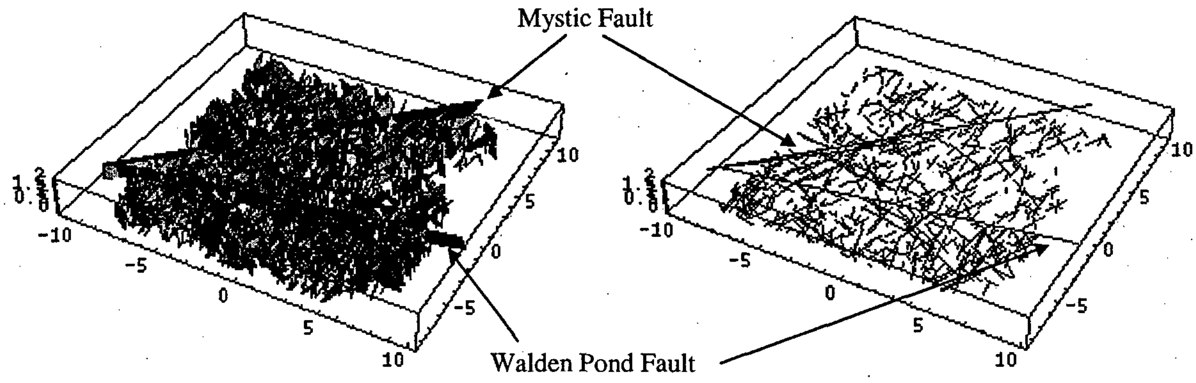


Figure 4.2 – Regional modeling output (fracture network and corresponding outcrops). The two major discontinuities shown are the Walden Pond Fault and the Mystic Fault.

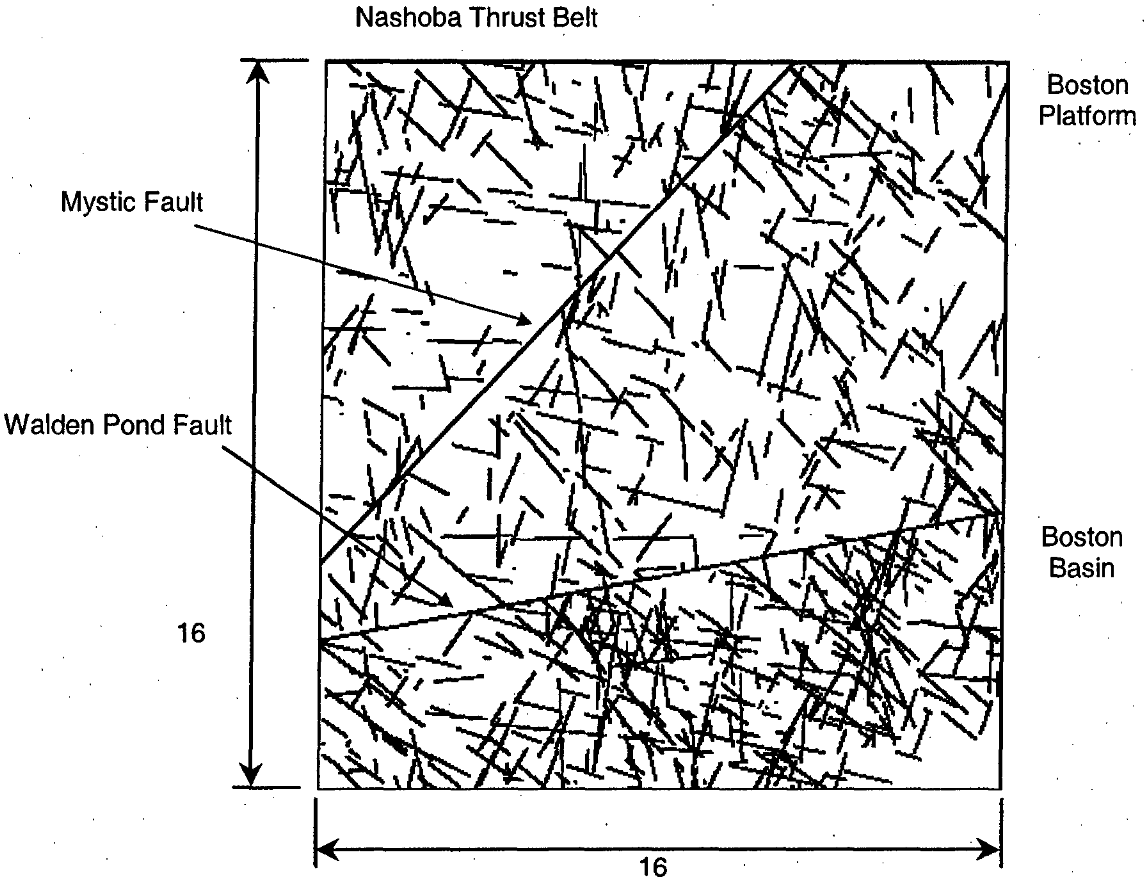


Figure 4.3 – Plan view of traces on a horizontal outcrop (from Meyer, 1999). This is not from the same simulation shown in Figure 4.2

However, we will only be concerned with the local modeling of the fracture systems in the Boston Area, which means that neither of these faults will be included in the simulations. Local modeling will focus more on an arbitrarily located volume within the Boston Basin. The next section discusses the parameters used in the local modeling and presents the results of the fracture intersection geometries and orientations.

The following paragraphs discuss the local modeling approach used by Meyer (1999) to calibrate the input parameters to be used in GEOFRAC to model the fracture networks in the Boston Area. These parameters will subsequently be used to model the fracture intersections in the same area.

The modeling parameters for the fracture formation in the Boston Area are taken from Meyer (1999). These parameters were derived from field tracelength and spacing data recorded by Billings (1976). The observations were made during the construction of several water supply and drainage tunnels in the area.

Four distinct joint sets were observed in the area. **Table 4.2** shows the strike and dip values for the joint sets observed in the Boston Area. The variations of the dip values are also given. In the model simulations, however, a constant value of ΔDip = ±19° is used for all the joint sets. This is the average of the ΔDip values for the four fracture sets.

Fracture Set	Strike	Average Dip	Δ Dip
A	N98°E	88°S	±8°
B	N50°W	80°NE	±20°
C	N170E	88°W	±18°
D	N160°W	70°W	±30°

Table 4.2 – Strike and dip of fracture sets in the Boston Area from observations by Billings (1976)

Billings observed that fracture set D is more abundant than the other three sets. To accommodate this, a larger fraction of the derived fracture intensity will be assigned to set D.

Fracture intensity and fracture size information are derived from observations of the fracture spacing, *s*, along lines or boreholes and the fracture tracelengths, *L* on outcrops in the tunnels. Observations made by Billings during the construction of tunnels in the area indicate that *s* ranges from 1cm to 15m and *L* from 5cm to 30m.

In order to come up with an estimate of the mean equivalent radius of the fractures, values of the mean and standard deviation of the tracelengths on the outcrops are needed (μ_L and σ_L). An assumption regarding the probability distribution of *L* is thus required. Meyer fitted the tracelength data to a lognormal distribution and since different combinations of the values of μ_L and σ_L could be used, a relationship between them had to be assumed in order to pin down the values. Values of $\mu_L = 1.50$ and $\sigma_L = 1.00$ are obtained. To estimate the fracture intensity, the mean spacing, *S*, is required. Meyer assumed that *s* has an exponential distribution and obtained *S* = 2.70. Fitting the data to their respective distributions is done by choosing the distribution parameters such that the probability of an observation that is smaller than the minimum value reported, $s_{\min}$ or $L_{\min}$, is equal to the probability of observing values larger than the maximum value reported, $s_{\max}$ or $L_{\max}$ (i.e. the area under the probability density function to the left of $s_{\min}$ is equal to the area to the right of $s_{\max}$, the same goes for $L_{\min}$ and $L_{\max}$).

Meyer set out to model the fracture network using parameters inferred from the interpretation of data gathered by Billings (1976). The goal was to generate a fracture network that would produce the same observations that Billings recorded during tunnel construction in the area: a mean fracture spacing of *S* = 2.70m and a mean fracture tracelength of $\mu_L = 1.50$ m.

The estimate for fracture intensity is obtained using the equation given by Dershowitz and Herda (1992).

$$P_{32} = \frac{C}{S}$$

Equation 4.1

where S is the fracture spacing. Dershowitz and Herda (1992) suggest a range of values between 1.0 and 3.0 for C. A value of 2.0 was used initially with the assumption that fracture orientations were distributed uniformly. The mean fracture size is estimated from tracelength data using the following equation given by Zhang (1999).

$$E[R_e] = \frac{128\mu_L{}^3}{6\pi^3[\mu_L{}^2 + \sigma_L{}^2]}$$

Equation 4.2

where $E[R_e]$ is the equivalent radius of the fractures, μ_L is the mean tracelength and σ_L is the standard deviation of the tracelengths. The equation above was developed for circular fractures so a correction on the input parameters is needed. The number of fractures, N, can be calculated from the size of the fractures, the volume of the modeling region, V, and the fracture intensity (P_{32}).

$$N = \frac{VP_{32}}{\pi E[R_e]^2}$$

Equation 4.3

Using **Equation 4.1** to **Equation 4.3** and the derived values of μ_L, σ_L and S, the following input parameters were obtained:

Derived from Observations			Derived Input Parameters		
μ_L	σ_L	S	$E[R_e]$	P_{32}	N
1.50	1.00	2.70	0.715	0.74	12470

Table 4.3 – Modeling parameters for the base case (from Meyer, 1999)

The fracture intensity of 0.74 is the total for the four fracture sets but it is not evenly divided among them. To account for the observation that fractures from set D are more numerous than the other sets, Meyer assumed that set D is responsible for 34% of the total intensity and sets A to C 22% each.

Meyer performed sensitivity analyses by individually changing the values of μ_L and S, holding the others to their original values. **Table 4.4** below shows the additional cases he considered and the corresponding input parameters.

Variations to Observed Values			Derived Input Parameters		
μ_L	σ_L	S	$E[R_e]$	P_{32}	N
1.80	1.00	2.70	0.946	0.74	7108
1.20	1.00	2.70	0.487	0.74	26814
1.50	1.00	2.00	0.715	1.00	16835
1.50	1.00	3.50	0.715	0.57	9620

Table 4.4 – Additional cases for sensitivity analysis (from Meyer, 1999)

Based on the results of 15 simulations for each case and upon studying the simulated values for the parameters P_{32}, $E[R_e]$, N, μ_L and S, Meyer suggested the following best estimates of the input parameters for modeling the fracture network in the Boston Area:

Best Estimates of the Observed Values			Derived Input Parameters		
μ_L	σ_L	S	$E[R_e]$	P_{32}	N
1.50	0.88	2.70	0.770	0.52	7559

Table 4.5 – Best estimates of the parameters for modeling the Boston Area fracture networks (from Meyer, 1999)

An example of the simulated fracture network and the corresponding intersections for the local modeling simulations are shown in **Figure 4.4** and **Figure 4.5**, respectively.

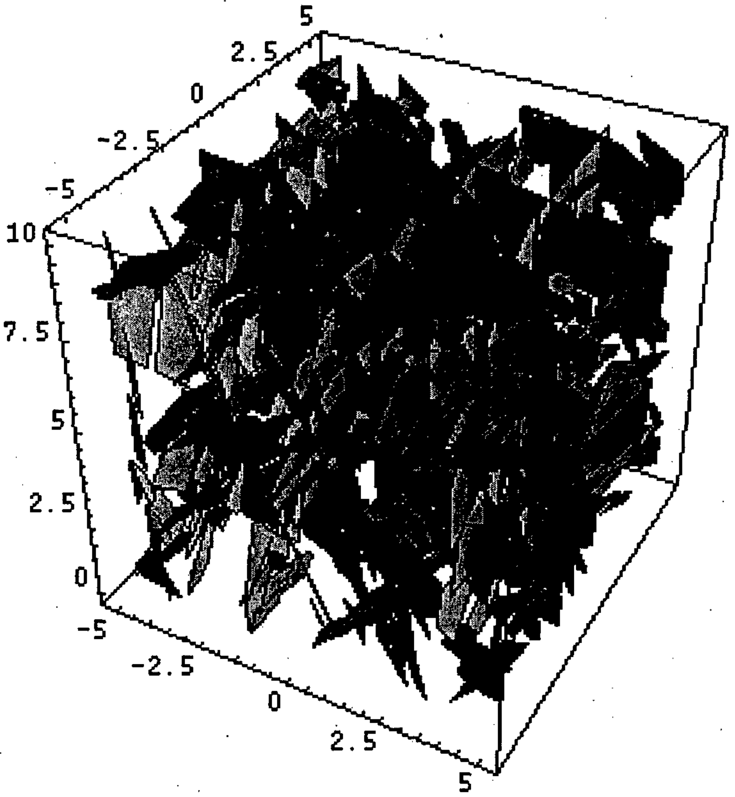


Figure 4.4 – Simulated fracture network for a 10 by 10 by 10 m modeling volume. (local modeling)

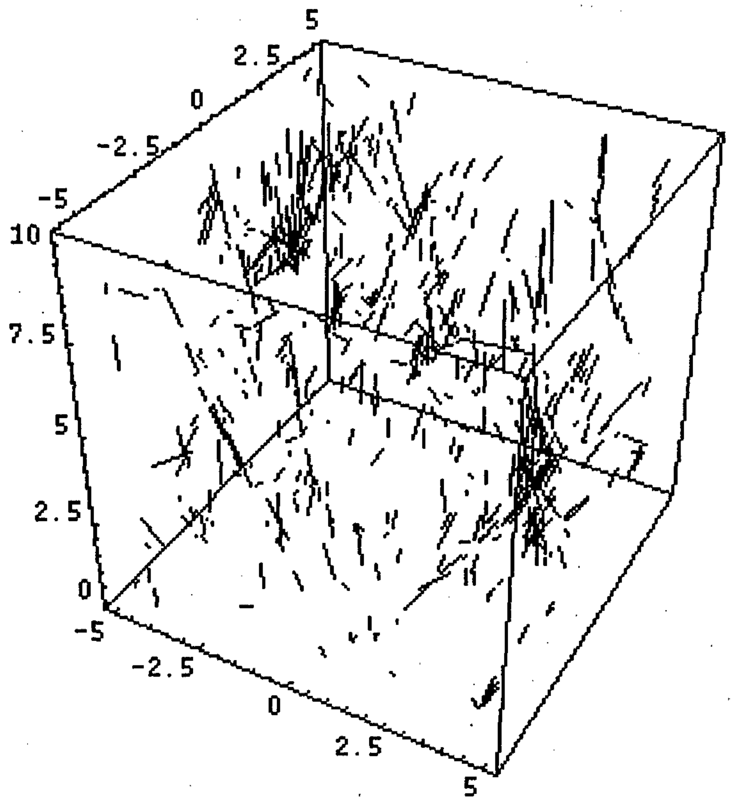


Figure 4.5 – The corresponding fracture intersections for the fracture network in Figure 4.4. (local modeling)

One can see that the current modeling capabilities not only show the geometry of the individual sub-networks but also the geometry and orientations of the individual intersections.

In the next section, the parameters given in **Table 4.5** will be used to simulate the fracture networks in the Boston Area and the geometry of the fracture intersections that result will be studied.

4.3 Results and Discussion

To study the geometry of the fracture intersections in the Boston Area, 15 simulations were run for each of five different modeling volume sizes. The modeling volumes used had edge lengths of 10, 12, 14, 15 and 20 meters. The computation time and memory requirements for larger volumes become very prohibitive and limit the size of the modeling volume that can be used.

In the following pages, graphs of the important output parameters are presented. The interpretations and explanations follow after each one. Also, the orientations of the intersections are plotted and compared with the computed values.

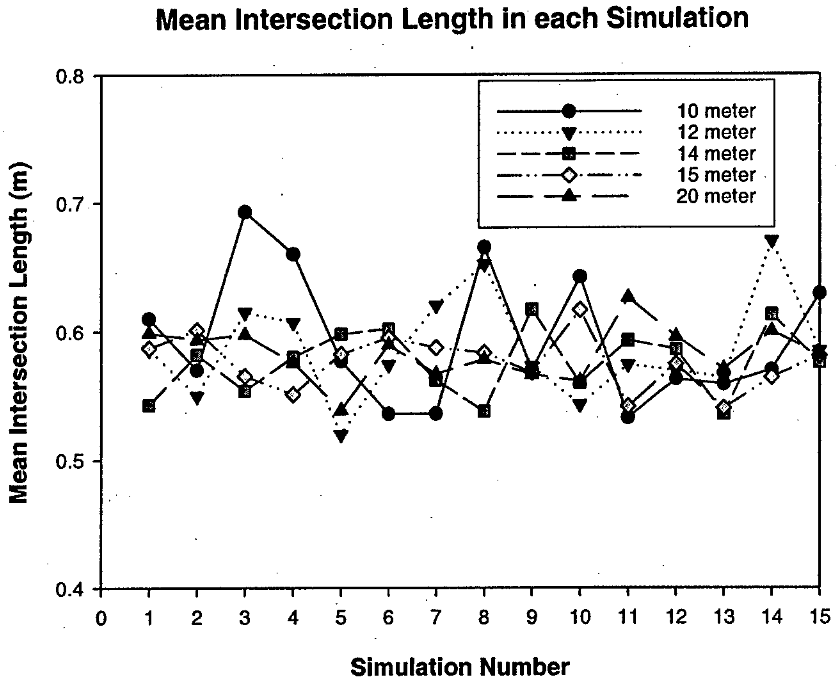


Figure 4.6 – Mean intersection length in each simulation

Mean Intersection Length in Each Simulation (Figure 4.6):

- The mean intersection length varies between 0.5m and 0.7m. The input (target) or mean fracture equivalent radius is 0.770m.

- As the modeling volume size increases, the variation of the mean intersection length from simulation to simulation becomes less although there is no significant difference in the behavior between the 15^3m^3 and the 20^3m^3 simulations.

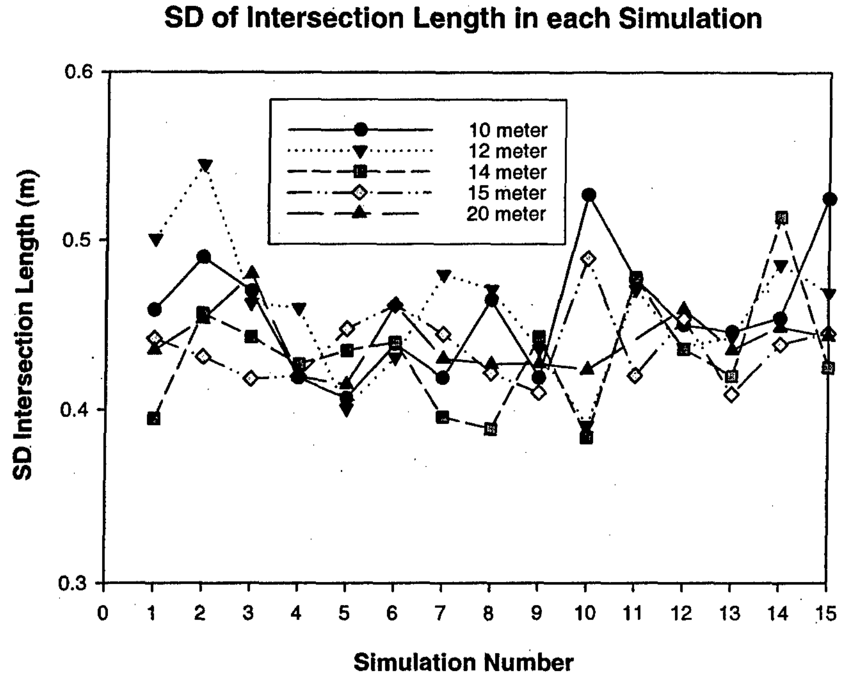


Figure 4.7 – Standard deviation of intersection length in each simulation

Standard Deviation of Intersection Length in Each Simulation (Figure 4.7):

- The standard deviations observed in all the simulations vary from about 0.39 to a maximum of about 0.55 (about 51% to 71% of the mean equivalent radius).

- The variation of the standard deviation of intersection length from simulation to simulation is largest when using the 10^3m^3 to 14^3m^3 modeling volume sizes. For the sizes 15^3m^3 and 20^3m^3, there isn't much difference in the way the standard deviation varies from simulation to simulation.

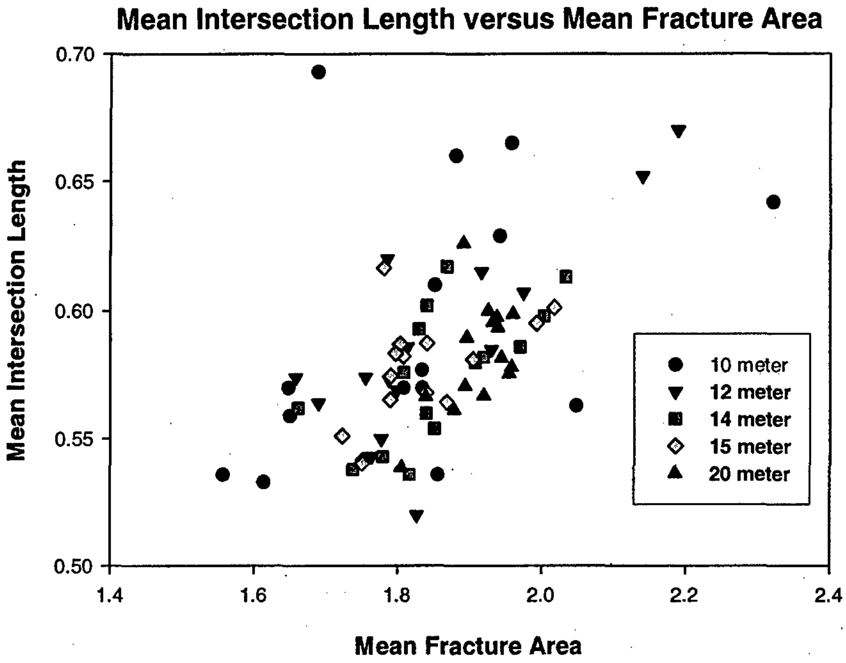


Figure 4.8 – Mean intersection length versus mean fracture area for each simulation

Mean Intersection Length versus Mean Fracture Area (Figure 4.8):

- There seems to be no specific trend that can describe the relationship between the mean intersection length and the mean fracture area. For the 10^3m^3 and 12^3m^3 modeling volumes, the data points are scattered all over. As the modeling volume size increases, the scatter of points becomes more confined. The data points for the 20^3m^3 volume show a vast improvement over the 10^3m^3 data points in terms of scatter but still do not suggest an obvious trend.

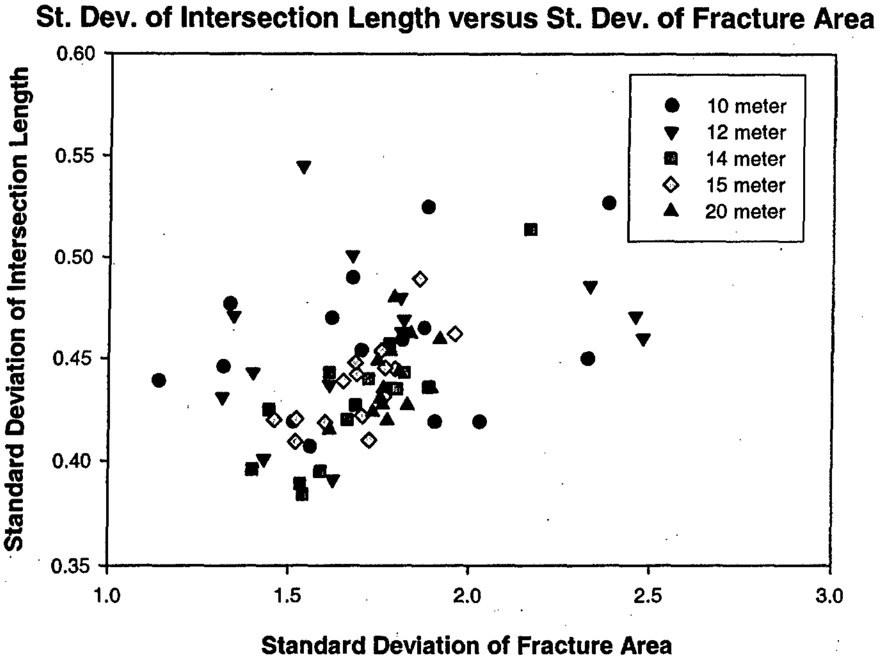


Figure 4.9 – Intersection length standard deviation versus fracture area standard deviation in each simulation

Standard Deviation of Intersection Length versus Standard Deviation of Fracture Area (Figure 4.9):

- As in **Figure 4.8**, there is no apparent trend in the relationship between the standard deviations of the intersection length and the fracture area. The same observations regarding the scatter of data points are also applicable here. The 10^3m^3 modeling volume produces the most scatter, while the 20^3m^3 modeling volume gives less scatter in the data.

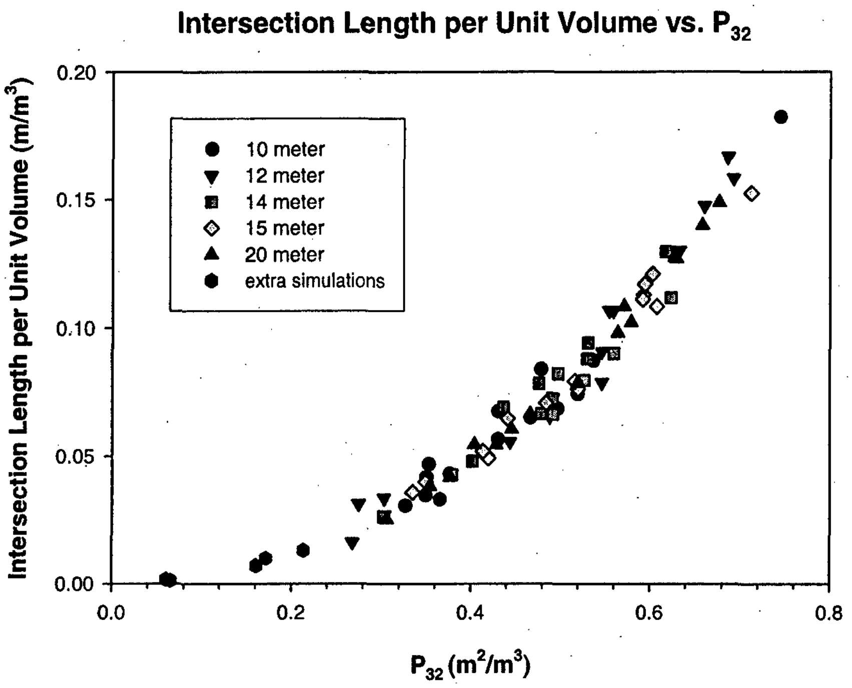


Figure 4.10 – Intersection length per unit volume versus fracture intensity. The "extra simulations" were performed to determine the fracture intensity at which there will be approximately zero intersection length per unit volume

Intersection Length per Unit Volume versus Fracture Intensity, P_{32} (Figure 4.10):

- The data for intersection length per unit volume with the corresponding P_{32} lies within a narrow band.

- The plot of the relationship is slightly curved and the slope increases with increasing P_{32}. This observation will be explained later since some of the other plots also exhibit the same behavior.

- Additional simulations were performed to determine the fracture intensity at which there will be approximately zero intersection length per unit volume. This value is seen to be about $P_{32} = 0.005$.

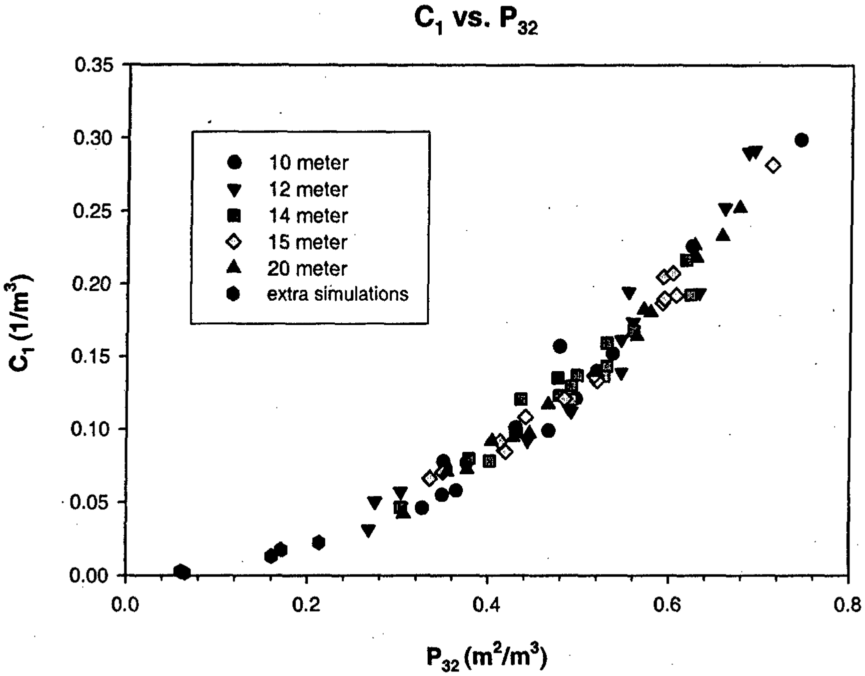


Figure 4.11 – C_1 versus P_{32}. C_1 is the number of intersections per unit volume. The "extra simulations" were performed to determine the fracture intensity at which there will be approximately no intersections

C_1 versus Fracture Intensity, P_{32} (Figure 4.11):

- The plotted data lie within a narrow band and the shape of the relationship is very similar to that between the intersection length per unit volume versus the fracture intensity (**Figure 4.10**).

- As expected, the relationship between the number of intersections per unit volume and the fracture intensity exhibits the same shape as the relationship between the intersection length per unit volume and the fracture intensity.

- According to additional simulations, the fracture intensity at which almost zero number of intersections occurs is also about $P_{32} = 0.005$. This result agrees with that obtained from **Figure 4.10**.

- The slope of the relationship is also increasing with increasing P_{32}. This can be explained by looking at the number of intersections that an additional fracture creates when added to a fracture network of a given intensity. When a single fracture is added to a network with very low intensity, it is likely that the single fracture will be isolated or will form only a few if not one intersection with the existing fractures. This accounts for the initial flatter slope observed in **Figure 4.11**. The length of the intersection formed by the newly added fracture will depend on the size distribution of the fractures and the location of the fractures. For fracture networks with higher intensities, an additional fracture will likely intersect with more than one fracture and thus form more than one intersection. This explains the steadily increasing slope as the fracture intensity increases. The same trend can be observed from the plot of intersection length per unit volume versus the fracture intensity (**Figure 4.10**). This shows that for a higher value of fracture intensity, the increase in the total intersection length due to the addition of fractures would be larger than if the fractures were added to a fracture network of lower intensity. Following this line of reasoning, it would be safe to assume the same kind of behavior for any fracture network that does not consist solely of a set of parallel fractures.

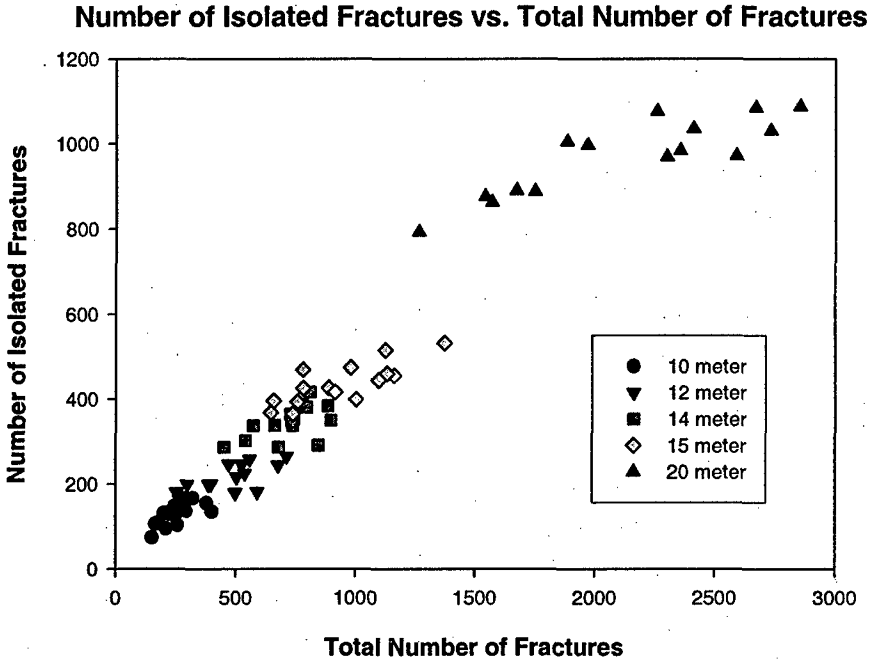


Figure 4.12 – Number of isolated fractures versus the total number of fractures

Number of Isolated Fractures versus Total Number of Fractures (Figure 4.12):

- The plot of number of isolated fractures versus the total number of fractures for each size of modeling volume shows that as the number of fractures increases there is also an initial increase in the number of isolated fractures. The number of isolated fractures then plateaus (at different levels for the different sizes of the modeling volumes). One might expect the slope of the relationship to become negative as the total number of fractures increases because as additional fractures are created, the once isolated fractures may become connected to existing sub-networks. Also, the number of isolated fractures will approach zero as the total number of fractures increases.

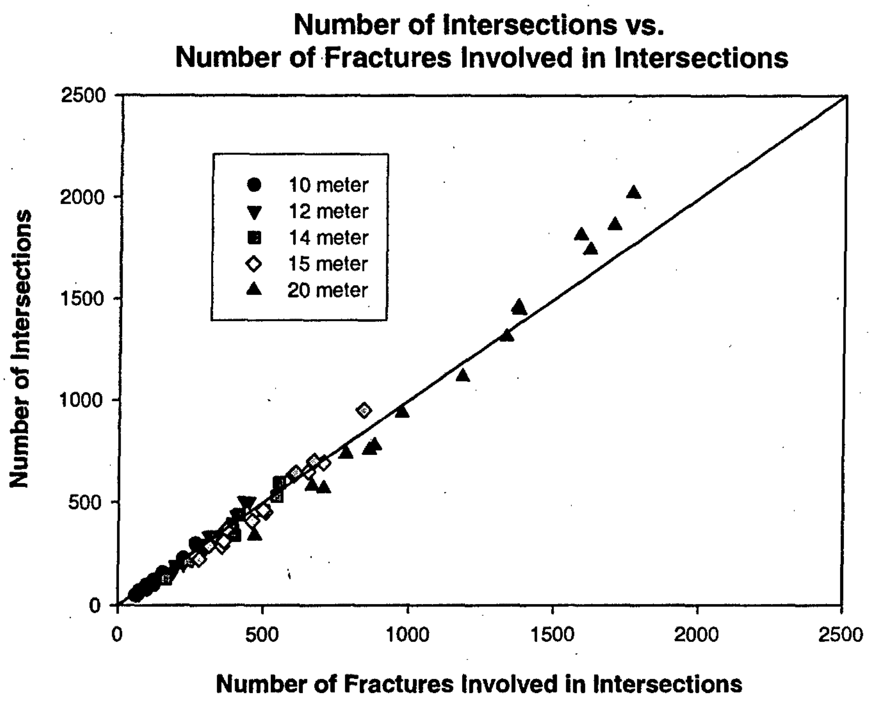


Figure 4.13 – Number of Fracture Intersections versus the number of fractures involved in the intersections (total number of fractures minus number of isolated fractures).

Number of Intersections versus the Number of Fractures Involved in the Intersections (Figure 4.13):

- The number of intersections is plotted against the number of fractures that actually create the intersections (total number minus the isolated fractures).

- For higher numbers of intersection-forming fractures at a given volume, one can observe the same trend of increasing slope as the ones seen in **Figure 4.10** and **Figure 4.11**. As the number of fractures increases, the addition of a single fracture forms more than one intersection due to the high fracture intensity.

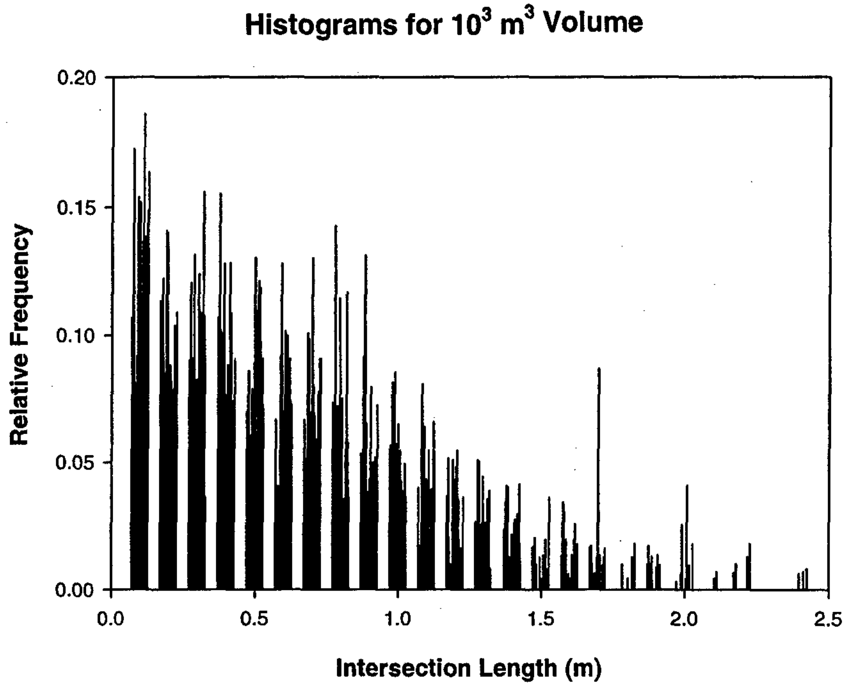


Figure 4.14 – Intersection length histograms for the 10 by 10 by 10 meter modeling volume

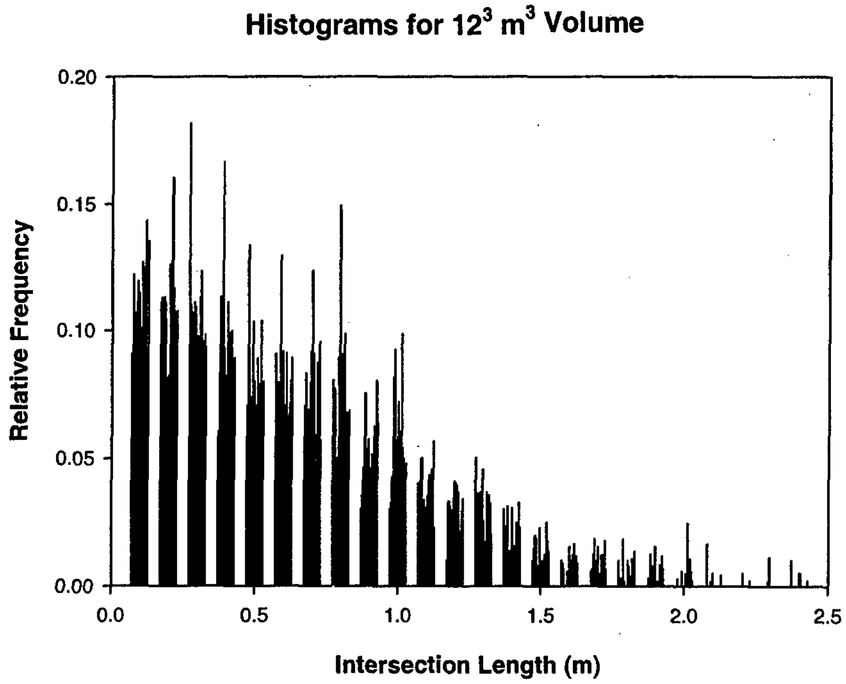


Figure 4.15 – Intersection length histograms for the 12 by 12 by 12 meter modeling volume

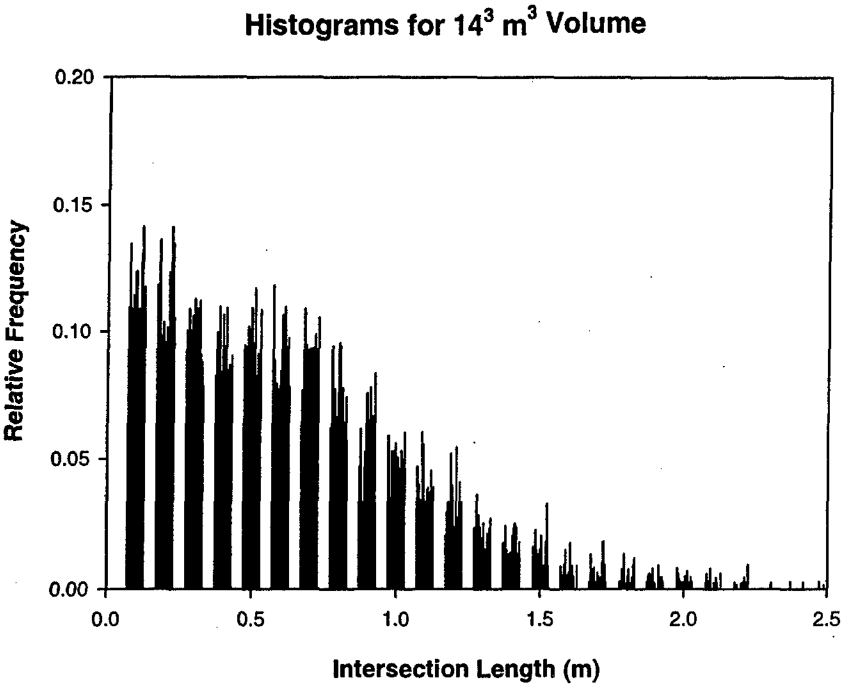


Figure 4.16 – Intersection length histograms for the 14 by 14 by 14 meter modeling volume

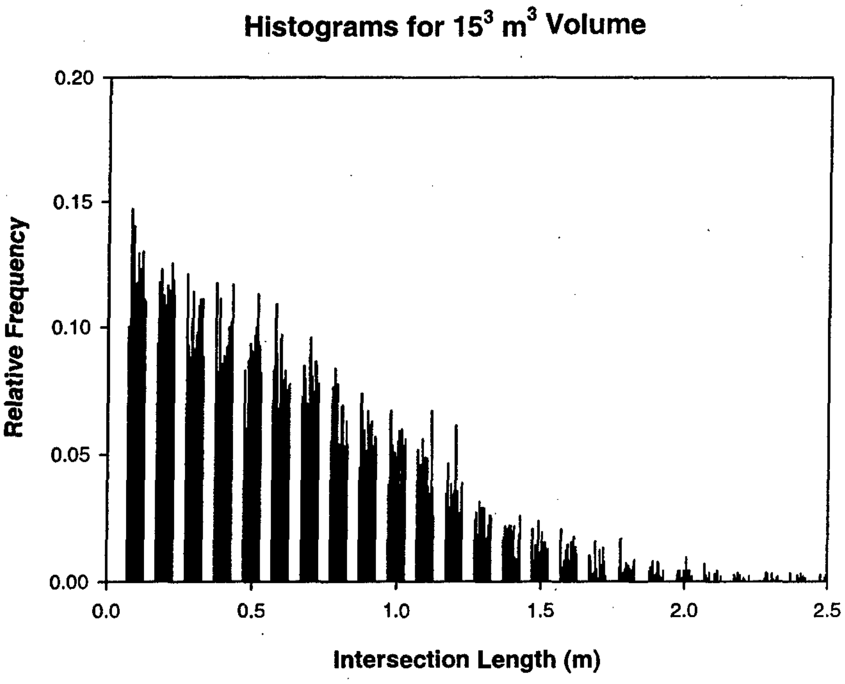


Figure 4.17 - Intersection length histograms for the 15 by 15 by 15 meter modeling volume

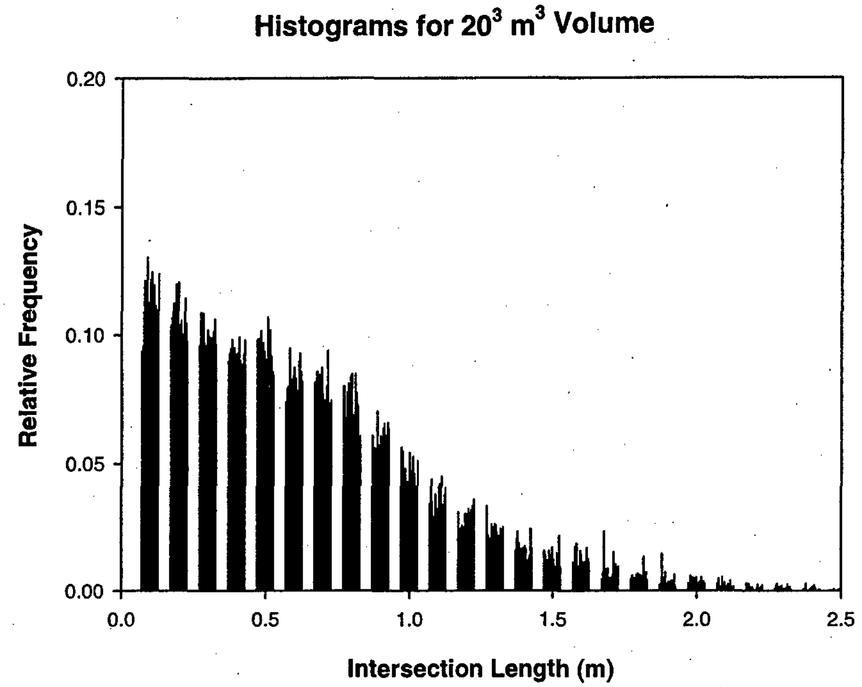


Figure 4.18 - Intersection length histograms for the 20 by 20 by 20 meter modeling volume

Intersection Length Histograms (Figure 4.14 to Figure 4.18):

- The length histograms for each size of modeling volume for each of the 15 simulations are plotted side by side in order to see the consistency of the relative frequencies as the modeling volume size changes. It can be observed that as the modeling volume increases (from 10^{3}m^3 to 20^{3}m^3), the relative frequencies for the intersection lengths become more consistent from simulation to simulation.

- As the modeling volume increases, the percentage of small intersections (10cm or less) also decreases. For the smallest modeling volume (10^{3}m^3), an average of 13.2% of all the intersection lengths is less than 10cm. For the largest modeling volume used (20^{3}m^3), an average of 11.3% of the intersections is 10cm or shorter.

- As the modeling volume size increases, the tail of the histogram becomes longer reflecting an increased occurrence of longer intersections. This is accompanied by the decrease in the percentages of smaller intersections.

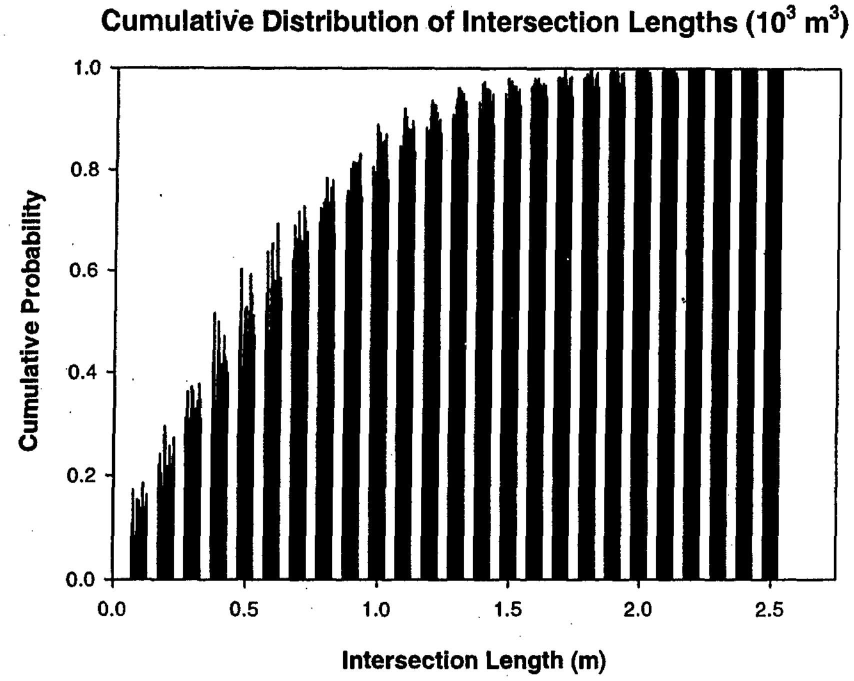


Figure 4.19 – Cumulative distribution of intersection lengths for the 10 by 10 by 10 meter modeling volume

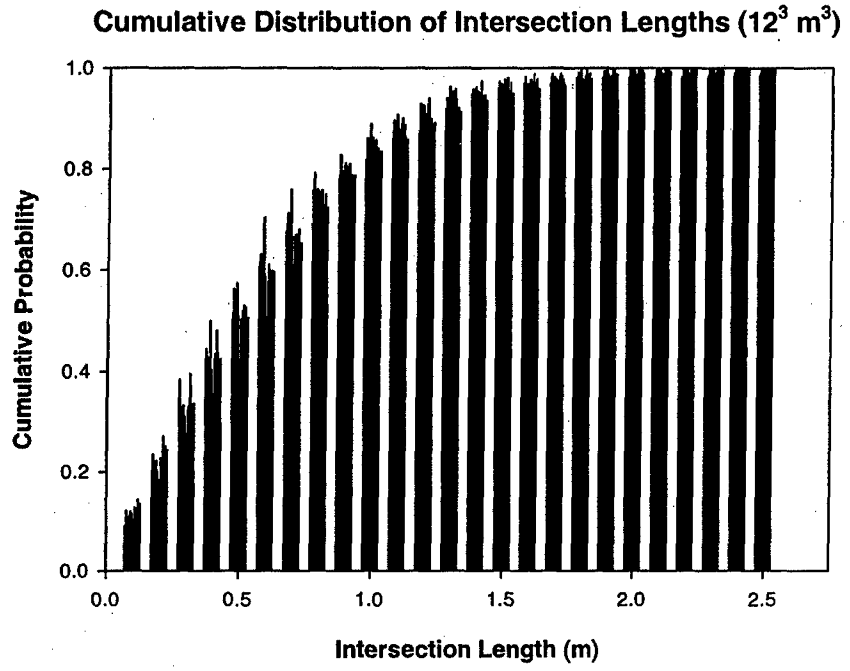


Figure 4.20 - Cumulative distribution of intersection lengths for the 12 by 12 by 12 meter modeling volume

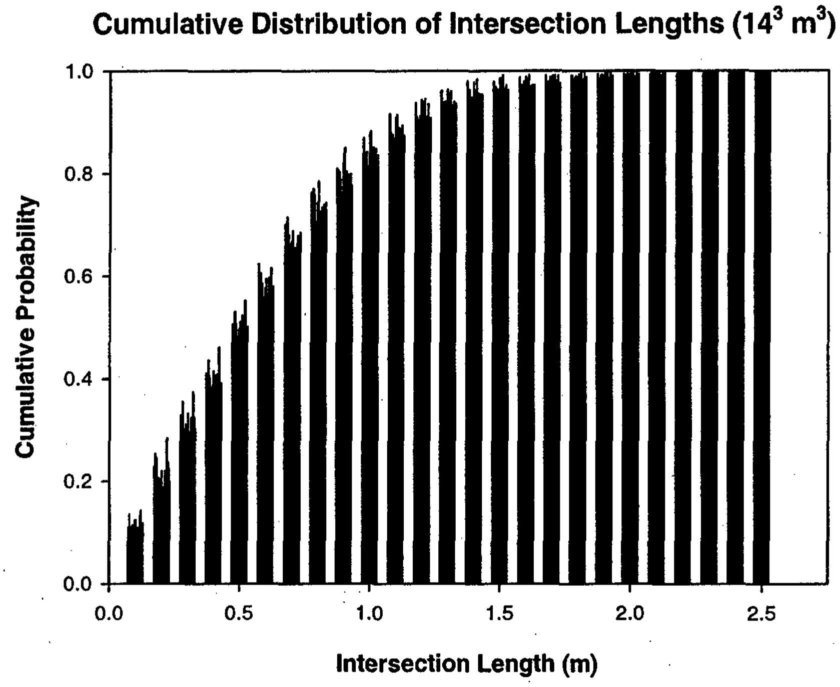


Figure 4.21 - Cumulative distribution of intersection lengths for the 14 by 14 by 14 meter modeling volume

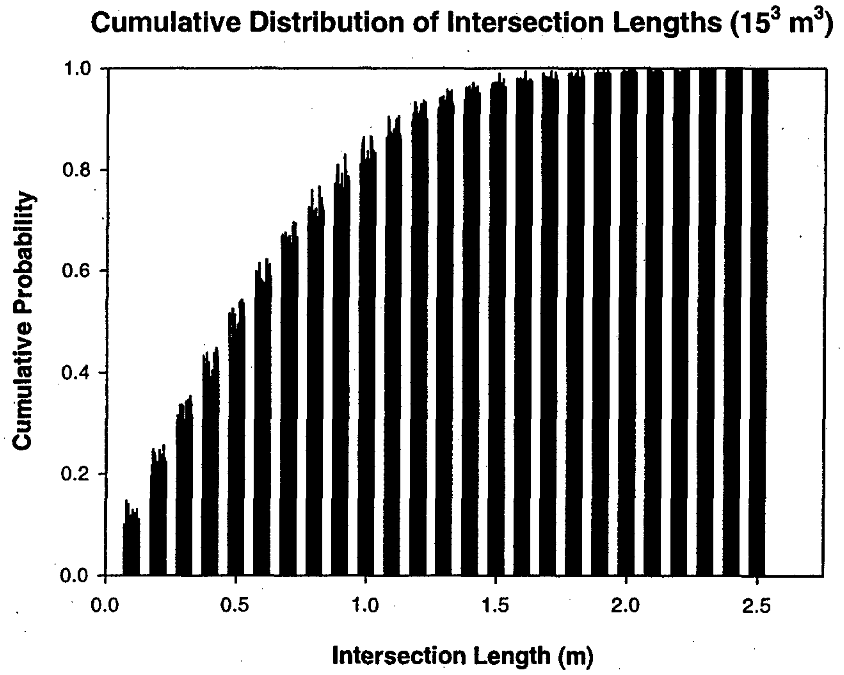


Figure 4.22 - Cumulative distribution of intersection lengths for the 15 by 15 by 15 meter modeling volume

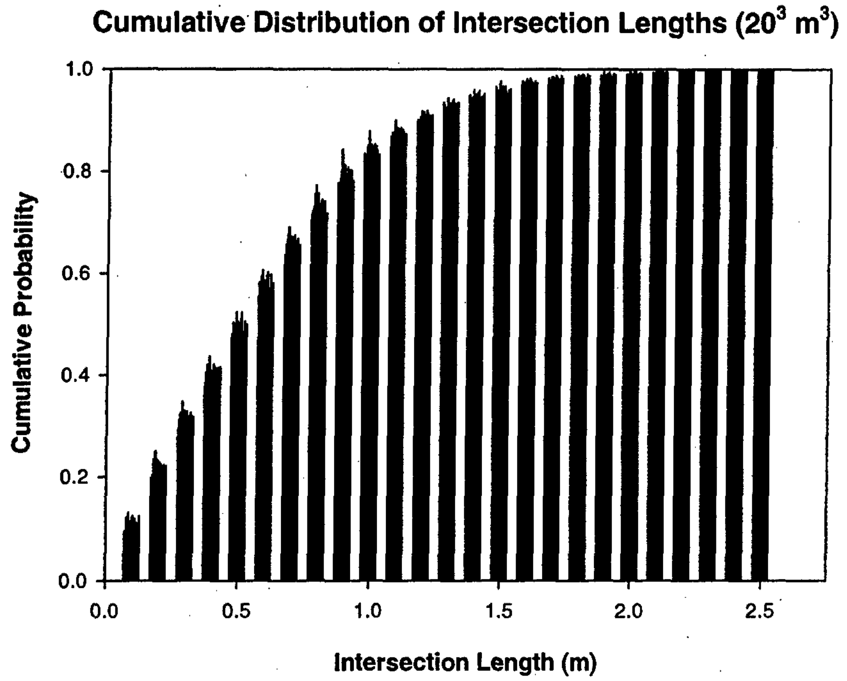


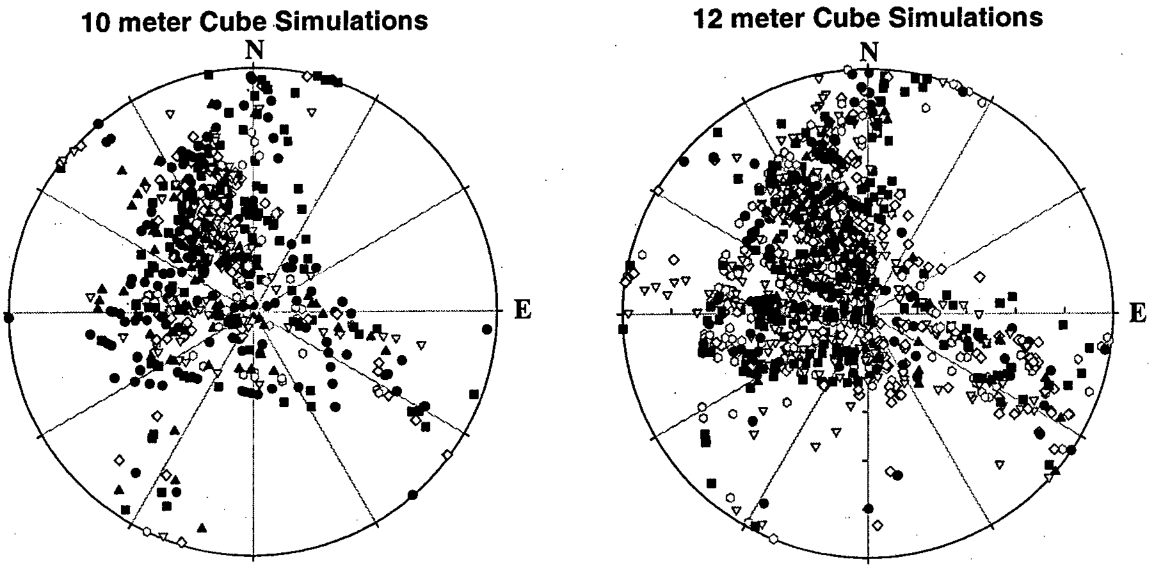
Figure 4.23 - Cumulative distribution of intersection lengths for the 20 by 20 by 20 meter modeling volume

Cumulative Relative Frequencies (Figure 4.19 to Figure 4.23):

- The cumulative distribution plots make it easier to see what percentage of the intersection lengths is equal to or smaller than a certain value. In all the plots, approximately 50% of the intersections are 0.5m or shorter.

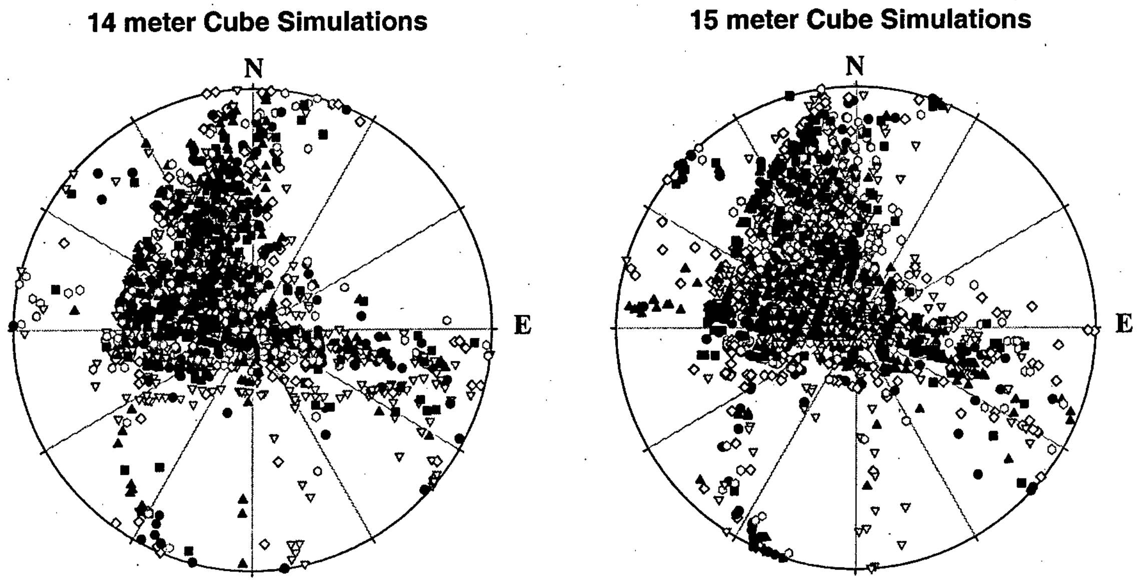
- Almost 80% of the intersections have lengths equal to or less than the mean equivalent radius of the modeled fractures (0.770m).

- About 9-12% of the intersections are 1.0m or longer.


10 meter Cube Simulations

12 meter Cube Simulations

Figure 4.24 – Lower hemisphere plots of trend and plunge of the intersection lines for the 10 m and 12 m cube simulations


14 meter Cube Simulations

15 meter Cube Simulations

Figure 4.25 – Lower hemisphere plots of trend and plunge of the intersection lines for the 14 m and 15 m cube simulations

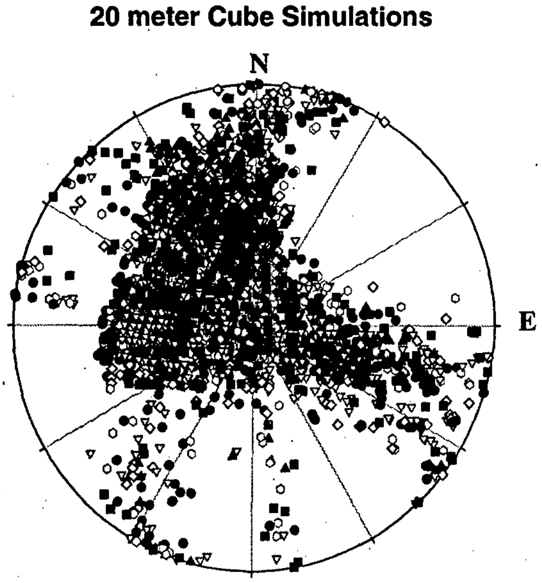


Figure 4.26 – Lower hemisphere plots of trend and plunge of the intersection lines for the 20 m cube simulation

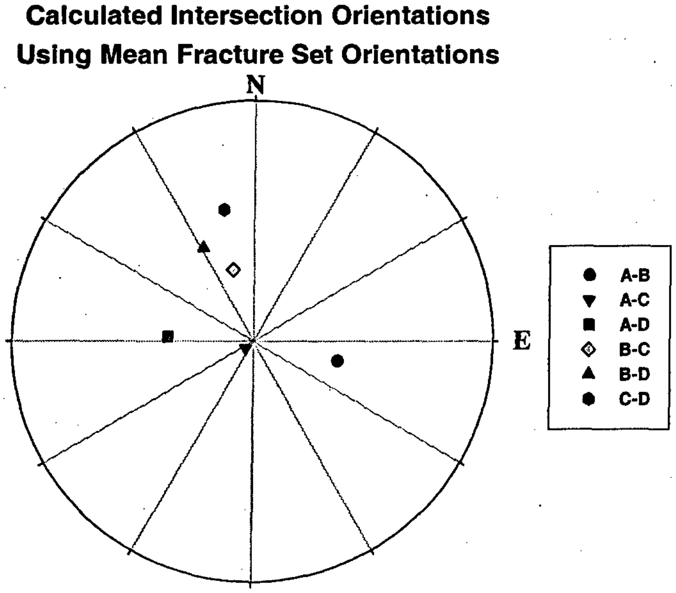


Figure 4.27 – Calculated trend and plunge of the intersections among the fracture sets A, B, C and D using only the mean orientations in Table 4.6 plotted using lower hemisphere

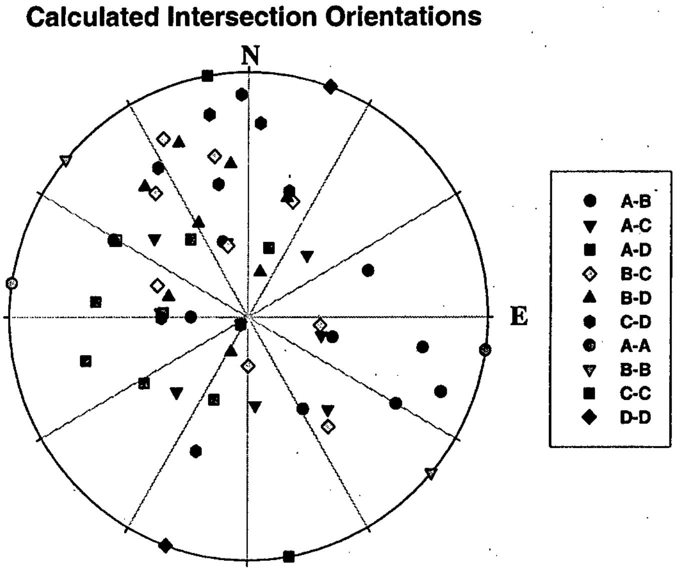


Figure 4.28 – Calculated trend and plunge of the intersections among the fracture sets A, B, C and D using their mean orientations ± the 19° assumed variation in the dip plotted using lower hemisphere (Table 4.6)

Fracture Set	Strike	Mean Dip	Mean Dip+19°	Mean Dip-19°
A	N98°E	88°S	73°N	69°S
B	N50°W	80°NE	81°SW	61°NE
C	N170°E	88°W	73°E	69°W
D	N160°W	70°W	89°W	51°W

Table 4.6 – Strike and dip values with the 19° variation. The intersections between the fractures with these orientations are plotted in Figure 4.28

Trend and Plunge of the Fracture Intersections (Figure 4.24 to Figure 4.28):

- The trend and plunge of each line of intersection is plotted in **Figure 4.24** to **Figure 4.26** to show the distribution of the orientations of the generated fracture intersections. These plots show that the majority of the intersections trend northwest.

- **Figure 4.27** plots the calculated orientations of the fracture intersections using the mean orientations of the four fracture sets. **Figure 4.28** shows the calculated orientations of the intersections using the data given in **Table 4.2.** This includes the intersection lines using the mean orientations of the fractures as well as the intersections when the ΔDip of ±19° is applied to each fracture set. All in all, there are 66 points in this plot. This gives a shape that comes close to the ones exhibited in **Figure 4.24** to **Figure 4.26.**

- Comparing **Figure 4.24** - **Figure 4.26** to **Figure 4.27** shows that the abundant northwest trending intersections are most likely formed by the intersection between the following pairs of fracture sets: A-D, B-C, B-D and C-D. So out of the six possible sets of fracture intersections in **Figure 4.27**, four of the sets trend northwest. Also contributing to the abundance of these northwest trending intersections is the higher intensity assigned to fracture set D. **Figure 4.28** shows that other pairs of intersecting fracture sets also contribute to this abundant group of intersections (A-B, A-C, A-A and B-B).

- **Figure 4.24** to **Figure 4.26** show a number of intersections that are oriented horizontally (plunge is zero). According to **Figure 4.28**, these intersections are formed by fractures belonging to the same fracture set.

The next section presents fracture connectivity as it relates to the flow behavior of the fracture network. Fracture connectivity is described by the flow dimension, which dictates the type of flow that occurs in the network. The geometric interpretation of the flow dimension is then used to develop estimation procedures involving the fracture intersection geometries.

5 Fracture Connectivity Based on Flow Behavior

In the previous sections, fracture connectivity was studied based on the geometric characteristics of the fracture intersections. In this chapter, a different approach is taken. The fracture network connectivity will be described from the standpoint of flow behavior using what is called the well-test flow dimension. First, an overview of well-testing is given. Then the method used for analyzing well-test data to obtain the flow dimension is discussed. Finally, geometric approaches to determining the flow dimension will be presented and verified using finite element flow simulations.

5.1 Field Determination

5.1.1 Hydraulic Testing in Fractured Rock

Hydraulic testing is used to characterize the flow behavior as well as the geometry of the fracture network. Single or multiple wells can be used but only single-well tests will be discussed here. The well or borehole could either be open to the atmosphere or closed off with the use of inflatable vessels called packers (see **Figure 5.1**).

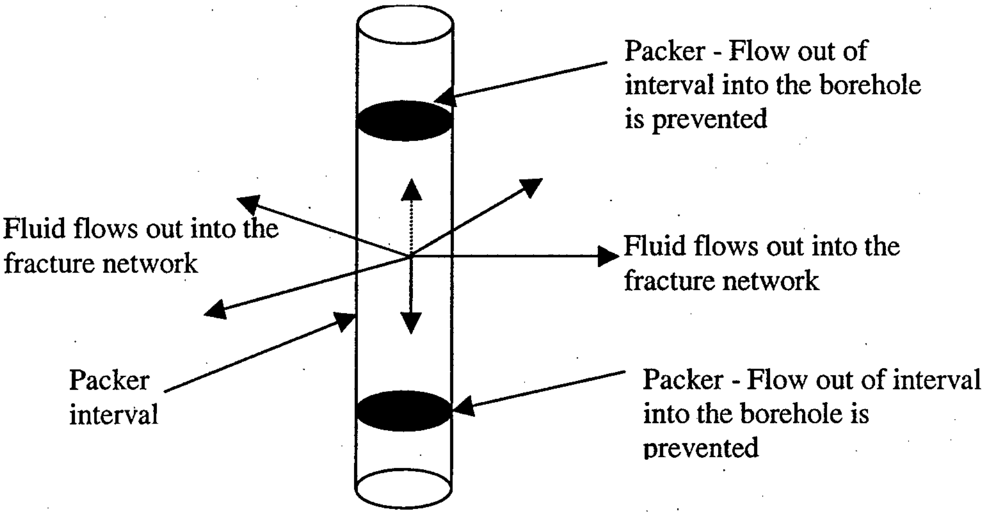

Packer - Flow out of interval into the borehole is prevented
Fluid flows out into the fracture network
Fluid flows out into the fracture network
Packer interval
Packer - Flow out of interval into the borehole is prevented

Figure 5.1 – An interval isolated using packers on each end. A submersible pump is typically used to pump the water in or out of the packer interval

To perform a hydraulic test, a well is first drilled into the formation. The test equipment is put in place then the flow in the well is allowed to reach the steady state condition. Once the steady state is reached, either constant pressure or constant discharge conditions are applied at the well. For a constant pressure test, the transient flow-rate into (or out of) the well is monitored during the test. The transient head in the well is monitored for a constant rate test. The transient rate or head is plotted versus time in a log-log plot. The resulting plot is then compared to theoretical curves for known flow dimensions in order to come up with an estimate of the flow dimension for the system. If the observed data fit the theoretical curve then the flow dimension of the curve is taken as the flow dimension of the fracture network for that particular well position. This process is called type-curve matching. The derivation of type-curves will be discussed in more detail in **Section 5.1.2**. The data for the drawdown (or pressure) and derivative of drawdown are superimposed onto the type-curves of different flow dimension n until a satisfactory fit is achieved. The comparison with the derivative of drawdown is usually done for refinement. Changing the position of the well will likely change the flow dimension such that one might come up with a flow dimension distribution for a fracture network. Altering the geometry of the well will also likely cause a change in the flow dimension since a shorter well will initially have access to fewer fractures compared to a longer well and the flow will likely take a different path into the fracture network.

Among the problems encountered in hydraulic tests is the effect of wellbore storage on the observed data. The goal is to determine the flow behavior of the fracture network and it is possible that the true flow behavior of the network may be masked by the behavior of the wellbore (Doe and Chakrabarty, 1997). Wellbore storage is defined as the volume of fluid added to or released from storage in the wellbore for a unit change in pressure (Earlougher, 1977). Wellbore storage effects can be broken down into three components: C_1, C_2 and C_3 (Doe et. al., 1987). C_1 is the change in storage due to the change in water level in the well. C_2 is the change in storage due to fluid compressibility. C_3 is the storage due to the deformation of the wellbore, the tubing and other well test equipment. C_1 is applicable only to open well tests and sometimes this may be much larger than the other two components such that those two can be neglected (Doe et. al., 1987). In most packer tests, C_1 can usually be neglected. For C_3, the contributions of the wellbore and the tubing can be computed if their elastic properties are known. Controlled lab tests may be required to determine the contribution of the packers to the wellbore effect (Doe et. al., 1987).

Another problem is caused by the disturbance of material around the well due to the drilling process. This disturbed material can manifest itself in a well test as a zone of altered permeability or in what is commonly called a skin effect. If the drilling causes an increase of permeability then there is a negative skin effect. If there is a decrease of permeability, a positive skin effect exists. This convention can be remembered using what is known as a skin factor (Rock Fractures and Fluid Flow). If there is a skin of higher permeability compared to the formation then the drawdown caused by pumping fluid out of the well will be less than when there is no skin. In this case, the skin factor is negative. If the skin has a lower permeability than the surrounding formation, the drawdown that results from pumping will be larger than when there is no skin therefore the skin factor is positive.

There are also the effects due to the outer boundary. The two basic types of outer boundaries are the no-flow (or closed) and the constant head boundaries. The no-flow boundary can be represented mathematically by $\frac{\partial h}{\partial r}(r = r_c, t) = 0$ and the constant head boundary by $h(r = r_c, t) = h_c$ where r_c indicates the location of the boundary. A typical boundary condition takes this form: $h(r \to \infty, t) = 0$. In other words, the injection or extraction does not cause a drawdown at a large distance from the well. Constant head boundaries may also indicate a connection to a reservoir that is sufficiently extensive and conductive such that the well test has no effect on its pressure while no-flow boundaries may indicate the termination of a fracture within solid rock (Doe et. al., 1987). The effect of a constant head boundary is illustrated in **Figure 5.2** for the case of injection into an aquifer. The dimensionless curve in **Figure 5.2** initially follows the curve for the condition where the constant head boundary is located at infinity and then flattens out to a constant value depending on the distance of the constant head boundary from the well. This distance is indicated by a dimensionless parameter $\mathbf{R_D} = \mathbf{R}/r_w$ where **R** is the distance of the boundary from the well and r_w is the radius of the well. The effect of a no-flow boundary is shown in **Figure 5.3** for injection into an aquifer. As in **Figure 5.2**, the curves in **Figure 5.3** initially follow that of an aquifer with infinite extent. The flow-rate subsequently drops at a time that depends on the distance of the no-flow boundary from the well indicated by $\mathbf{R_D}$. The final rates of flow in the different curves in **Figure 5.3** may reflect the rate of leakage into the surrounding low-permeability rock (Doe et. al., 1987).

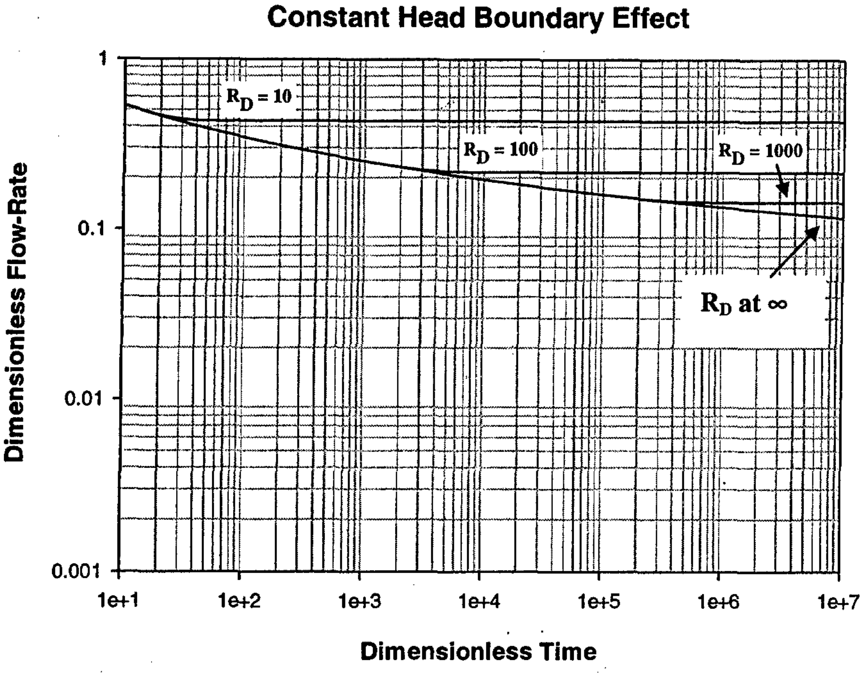


Figure 5.2 – Effects of a constant head boundary (from Doe et. al., 1987). R_D is a dimensionless parameter that represents the distance of the boundary from the point of injection (well)

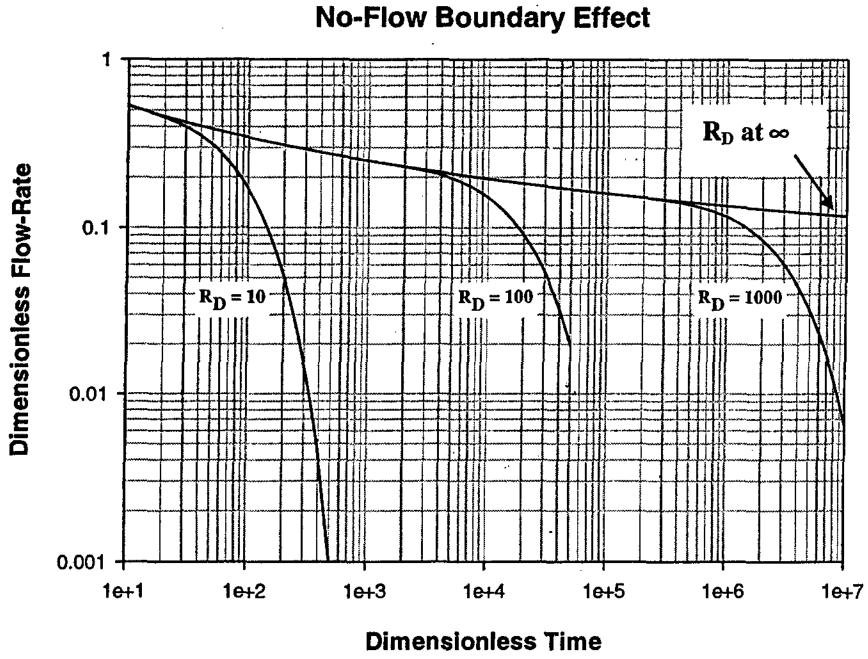


Figure 5.3 – Effect of a no-flow boundary (from Doe et. al., 1987). R_D is a dimensionless parameter that represents the distance of the boundary from the point of injection (well)

The next section takes a closer look at the physical significance of the flow dimension and the derivation of type curves.

5.1.2 Well-Test Flow Dimension and Type-Curve Matching

To better understand the concept of the well-test flow dimension, its physical significance is discussed. In a fracture network, the flow area along the flow paths may vary as the distance away from a well raised to some power. Flow dimension is defined as the power of this change **plus one**. To determine the flow area formed between two intersections lying on the same fracture, the orientations and lengths of the individual intersections must be found. The geometry of the intersections is used to calculate the flow width, which is multiplied by the fracture aperture to get the flow area. The flow properties may also change with radial distance and this effect can be combined with that of the changing flow area using the conductance, *C*. Conductance is defined as the product of the flow area and the hydraulic conductivity. So, a power change in conductance or flow area can be written as:

$$C \propto A_f \propto r^{n-1}$$

Equation 5.1

where *C* is the conductance, A_f is the flow area, *r* is the distance away from the well and *n* is the flow dimension.

Some examples are given for integer values of *n*. **Figure 5.4** shows the case for $n = 1$ and the flow area is $A_f = k_p r^0$ where k_p is the constant of proportionality. This is equivalent to a pipe with constant cross-sectional area extending from a well. A fracture that has been closed by mineralization and where a solution channel has formed can possess such a flow-path geometry. Another example for $n = 1$ would be a fracture network that propagates in one direction and with low intensity. For $n = 2$, the flow area is $A_f = k_p r$. The flow area increases linearly with radial distance away from the well. An example for this case would be a large fracture perpendicular to the well and intersecting it (**Figure 5.5**). The flow area at a radius *r* would be the product of the circumference of a circle whose radius is *r* (the distance from the well) and the fracture aperture or $A_f = 2\pi r \cdot a_p$ where a_p is the aperture (**Figure 5.5**). Assuming that the aperture is constant throughout the fracture, the relationship between A_f and *r* is linear. For a flow dimension of $n = 3$, the flow area increases as the square of the radial distance from the well. An example would be the point source shown in **Figure 5.6**. The flow is radially outward in all directions and the flow area is the surface of the sphere. Think of the change in flow area as an expanding sphere. At a distance *r* from the source, the flow area is the surface area of the sphere with radius *r*, that is, $A_f = 4\pi r^2$. Thus the flow area varies as r^2. This is a very ideal example and serves only to illustrate the different ways the flow area changes with radial distance away from the well. A fracture network extending in all three directions with high intensity may exhibit such a variation in flow area. The flow dimension can also take on values less than one and greater than three as well as non-integer or fractional values. Flow area may decrease by a power law of radial distance reflecting a flow dimension less than one (Doe, 1991). Flow in such a formation is called sublinear. On the other hand, flow area may increase with distance by a power greater than two, corresponding to a flow dimension greater than three. Such geometry supports hyperspherical flow (Doe, 1991).

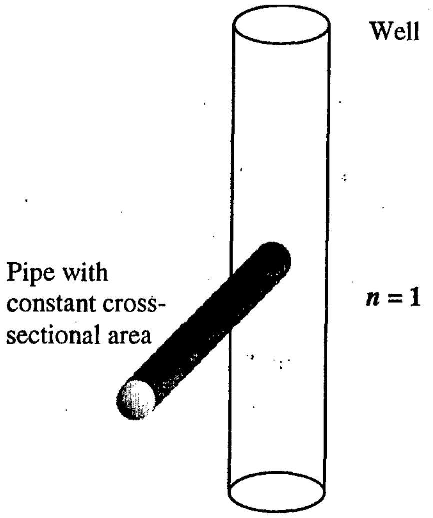


Figure 5.4 – Example for flow dimension equal to 1

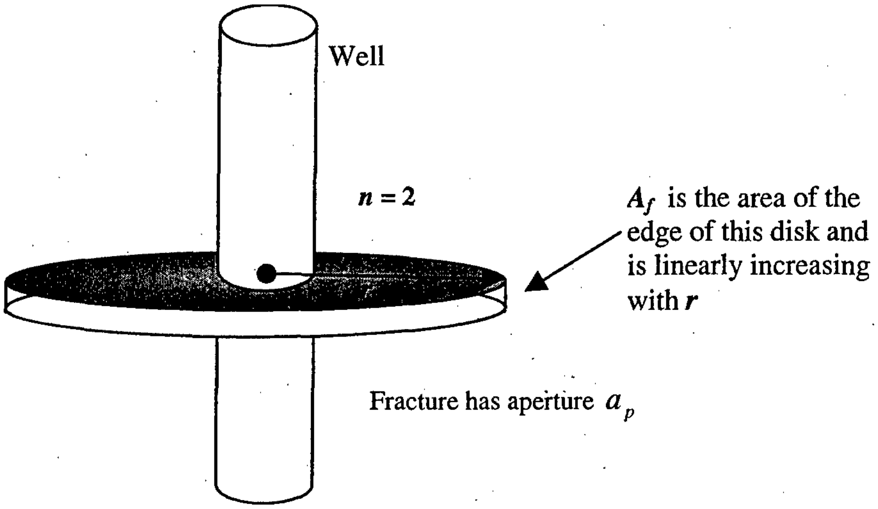

Well
$n = 2$
A_f is the area of the edge of this disk and is linearly increasing with r
Fracture has aperture a_p

Figure 5.5 – Example for flow dimension equal to 2

To understand the concept of a non-integer or fractional flow dimension, consider the two networks of conductive elements in **Figure 5.7**. **Figure 5.7a** is a regular Euclidean network while **Figure 5.7b** approximates a fractal network known as a modified Sierpinski gasket (Rock Fractures and Fluid Flow). Assuming that a well pumps from the center of each network, the flow area available at a distance *r* can be determined by drawing a circle of radius *r* and counting the conductive elements intersected by the circle. In network **(a)**, the number of conductive elements is directly proportional to the distance *r* from the well. This means that network **(a)** has a flow dimension of two. Therefore, **Figure 5.7(b)** is another representation of **Figure 5.5**. In network **(b)**, however, the flow area is proportional to $r^{0.59}$ and its flow dimension is 1.59. The fractal network in **Figure 5.7(b)** has a lower flow dimension because as *r* increases, the circle formed intersects increasingly larger regions without conductive elements. So a fractal network can be visualized as a partially connected network of conductive elements (Rock Fractures and Fluid Flow).

Now that geometric representations of the flow dimension have been presented, it is time to look at how it influences the flow behavior of the fracture network. The flow behavior of the fracture network for different values of n can be shown using type-curves. As discussed before, type-curves are a tool for analyzing well-test data in order to assess the flow and geometric properties of the fracture network. Type-curves are plots of the head or discharge at the well versus time and are derived by solving the governing equations for the flow occurring between a well and a fracture network. An example of these governing equations is given in the model proposed by Barker (1988) for hydraulic tests in fractured rock. Barker generalized the equations to include non-integer values of the flow dimension. He found this necessary because oftentimes, the use of integral values in a flow model gave results that could not satisfactorily match observed data.

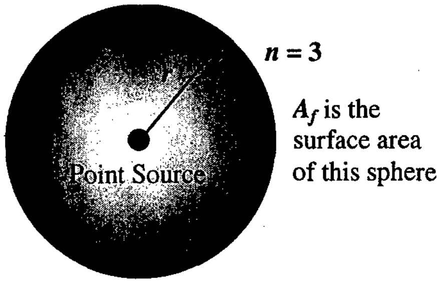


Figure 5.6 – Example for flow dimension equal to 3

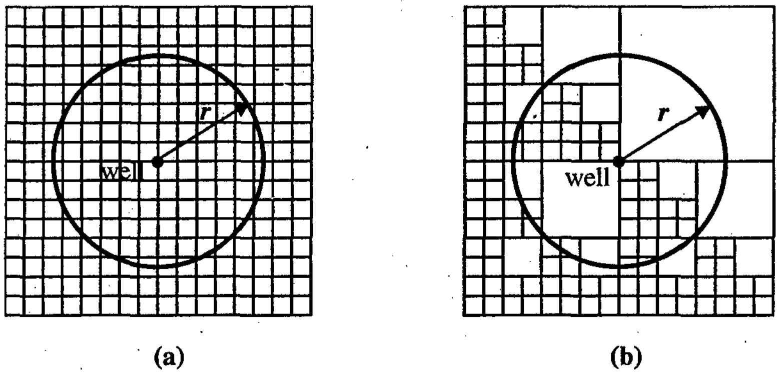


Figure 5.7 – Network (a) is a regular Euclidean network. (b) is a fractal network of flow dimension 1.59 (from Rock Fractures and Fluid Flow)

To derive the governing equations, Barker made the following assumptions:

1. Darcy's law applies throughout the system.

2. Flow is radial and n-dimensional from a single source into a homogeneous and isotropic fractured medium characterized by a hydraulic conductivity K_f and specific storage S_{sf}.

3. The source is an n-dimensional sphere (a finite cylinder in two-dimensional flow or a sphere for three-dimensional flow).

4. The source (well) has an infinitesimal skin characterized by a skin factor s_f. A head loss occurs across the surface or skin of the source and this head loss is proportional to s_f and the rate of flow through the surface.

First, the flow through the shell bounded by equipotential surfaces at r and $r+\Delta r$ is considered (shown in **Figure 5.8** for $n=2$). The region between the surfaces has a volume of $b^{3-n}\alpha_n r^{n-1}\Delta r$ (for small Δr) where α_n is the surface area of a unit sphere in n dimensions and is defined as:

$$\alpha_n = \frac{2\pi^{\frac{n}{2}}}{\Gamma(n/2)}$$

Equation 5.2

where $\Gamma(x)$ is the gamma function. The physical significance of b^{3-n} can best be represented for the case where $n=2$, it is the height of the circular cylinder representing the fractured rock (see **Figure 5.8**). For $n=3$, there seems to be no physical significance since b^{3-n} reduces to 1.

For a small change in time, Δt, the head in the shell changes an amount Δh and the volume of water entering the shell, ΔV, can be expressed as:

$$\Delta V = S_{sf} b^{3-n}\alpha_n r^{n-1}\Delta r \Delta h$$

Equation 5.3

where S_{sf}, the specific storage of the formation, is defined as the volume of water that a unit volume of formation releases from storage under a unit decline in hydraulic head (Barker, 1988). Using Darcy's law, the net volumetric rate q into the shell is given by:

$$q = K_f b^{3-n}\alpha_n (r+\Delta r)^{n-1}\frac{\partial h}{\partial r}(r+\Delta r, t) - K_f b^{3-n}\alpha_n r^{n-1}\frac{\partial h}{\partial r}(r,t)$$

Equation 5.4

where K_f is the hydraulic conductivity of the fractured medium, h is the head in the fracture formation, r is the distance from the well or source, t is the time and b and α_n are as defined before. Considering the conservation of the water in the shell ($\Delta V = q\Delta t$) and substituting **Equation 5.3** and **Equation 5.4**, we have

$$S_{sf}b^{3-n}\alpha_n r^{n-1}\Delta r\Delta h = K_f b^{3-n}\alpha_n[(r+\Delta r)^{n-1}\frac{\partial h}{\partial r}(r+\Delta r,t) - r^{n-1}\frac{\partial h}{\partial r}(r,t)]\Delta t$$

Equation 5.5

Re-arranging gives

$$S_{sf}\frac{\Delta h}{\Delta t} = \frac{K_f}{r^{n-1}}\frac{[(r+\Delta r)^{n-1}\frac{\partial h}{\partial r}(r+\Delta r,t) - r^{n-1}\frac{\partial h}{\partial r}(r,t)]}{\Delta r}$$

Equation 5.6

Taking the limits of both sides gives

$$S_{sf}\frac{\partial h}{\partial t} = \frac{K_f}{r^{n-1}}\frac{\partial}{\partial r}\left(r^{n-1}\frac{\partial h}{\partial r}\right)$$

Equation 5.7

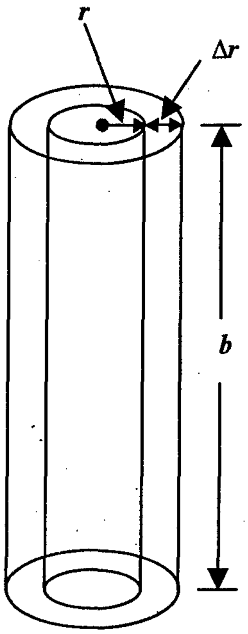


Figure 5.8 – Equipotential surfaces at r and $r+\Delta r$

The change in volume of the wellbore storage (see **Figure 5.9**) can also be related to the injection rate using Darcy's law. This gives the following equation:

$$S_w\Delta H = [Q(t) + K_f b^{3-n}\alpha_n r_w^{n-1}\frac{\partial h}{\partial t}(r_w,t)]\Delta t$$

Equation 5.8

and therefore

$$S_w \frac{\partial H}{\partial t}(t) = Q(t) + K_f b^{3-n} \alpha_n r_w^{n-1} \frac{\partial h}{\partial r}(r_w, t)$$

Equation 5.9

where S_w is the storage capacity of the source (**Figure 5.9**), *H(t)* is the head in the source (well), *Q(t)* is the injection rate into the well and r_w is the radius of the well. There is also a head loss across the boundary between the well and the fracture formation. The difference in the head in the well and the head in the fracture formation at a radial distance r_w is given by

$$H(t) = h(r_w, t) - s_f r_w \frac{\partial h}{\partial t}(r_w, t)$$

Equation 5.10

where s_f is the skin factor.

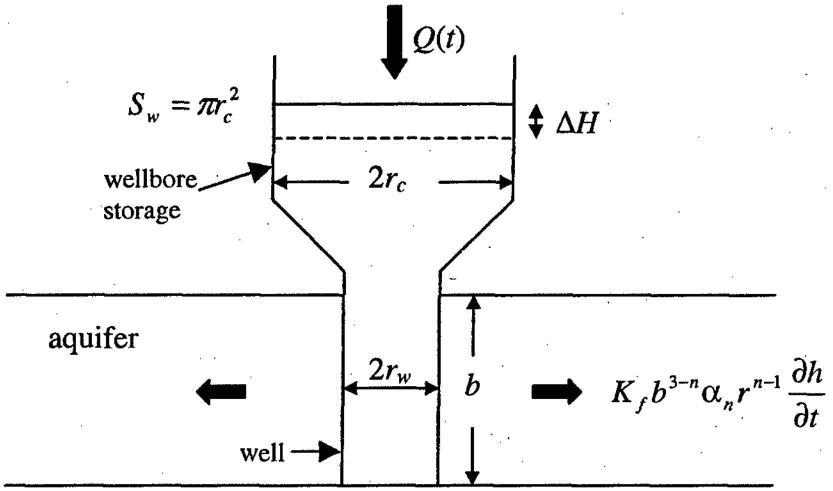


Figure 5.9 – Illustration of wellbore storage effect (from Barker, 1988)

One boundary condition is that the head at some fixed distance from the well is constant or $h(r_0, t) = h_0$ and usually takes the form of $\lim_{r \to \infty} h(r,t) = 0$. In other words, the pumping in or out of water does not cause a drawdown at distances far from the well at any time *t*.

The initial condition (i.e. just before pumping starts) states that the head in the well and the fracture network is zero or $h(r,0) = H(0) = 0$ (in equilibrium). The solution to **Equation 5.7** is then obtained using Laplace transforms and modified Bessel functions. Relationships between the Laplace transforms of the discharge, the head in the well and the head in the fracture network are derived. For a detailed derivation of these relationships, the reader is referred to Barker (1988).

The derived relationships between the Laplace transforms of the discharge, the head in the well and the head in the fracture network can further be simplified by additional conditions. For example, for a constant rate test, the condition $Q(t) = Q_0$ is applicable. For a constant head test, $H(t) = H_0$ is applied. Simplifications from well geometry include letting the radius of the well approach zero to represent a line source. Barker (1988) discusses in detail different cases that make use of such simplifications.

Doe and Chakrabarty (1996) extended Barker's model to two-zone composite formations. Two-zone formations are composed of an inner and outer zone each with its own set of flow properties and extent (example given in **Figure 5.10**). Basically, what Doe and Chakrabarty have done is to consider the skin as a zone in itself rather than use an infinitesimally thin region to represent it as Barker did.

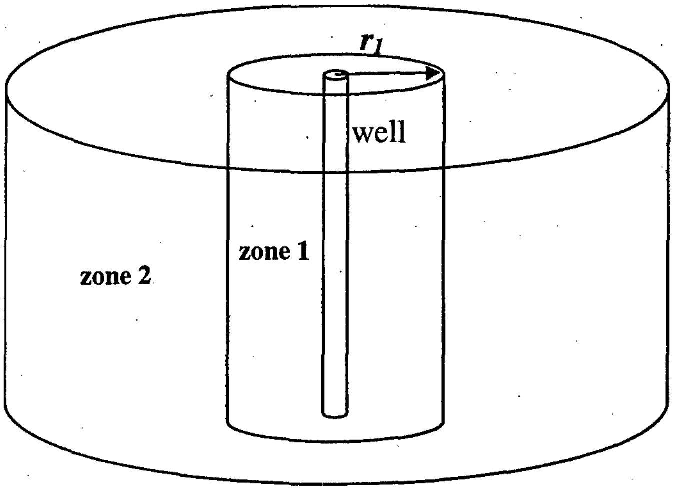


Figure 5.10 – Two-zone composite system with both inner and outer zone flow dimensions equal to 2 (Concentric Cylinders). Inner zone may have different flow properties from the outer zone.

Doe and Chakrabarty used the same governing equation (**Equation 5.7**) but applied to each of the two zones.

$$\frac{K_1}{r^{n_1-1}}\frac{\partial}{\partial r}\left(r^{n_1-1}\frac{\partial h_1}{\partial r}\right) = S_{s1}\frac{\partial h_1}{\partial t} \qquad r_w \leq r \leq r_1$$

Equation 5.11

$$\frac{K_2}{r^{n_2-1}}\frac{\partial}{\partial r}\left(r^{n_2-1}\frac{\partial h_2}{\partial r}\right) = S_{s2}\frac{\partial h_2}{\partial t} \qquad r_1 \leq r \leq \infty$$

Equation 5.12

where K_1 and K_2 are the hydraulic conductivities of the inner and outer zone, respectively. n_1 and n_2 are the flow dimension values for the inner and outer regions, respectively. The head in the inner region is h_1 and in the outer region it is h_2. S_{s1} is the specific storage of the inner region and S_{s2} is that of the outer region. Rate and head continuity should also be considered at $r = r_1$. For the rate continuity, we have

$$\left(\frac{K_1 b^{n_2-n_1}}{K_2} \frac{\alpha_{n_1}}{\alpha_{n_2}} r_1^{n_1-n_2} \right) \frac{\partial h_1}{\partial r}\bigg|_{r=r_1} = \frac{\partial h_2}{\partial r}\bigg|_{r=r_1}$$

and for the head continuity, $h_1(r = r_1, t) = h_2(r = r_1, t)$. The following conditions are used to solve the system of partial differential equations (**Equation 5.11** and **Equation 5.12**):

$$h_1(r, t = 0) = h_2(r, t = 0) = H(t = 0) = 0$$

Equation 5.13

Equation 5.11 and **Equation 5.12** are solved simultaneously using Laplace transforms and modified Bessel functions. **Figure 5.11** and **Figure 5.12** show examples of generated type-curves for a constant rate test. The derivative of the drawdown (or pressure) is included in order to refine the match between the data and the type-curves and therefore refine the estimate of the flow dimension.

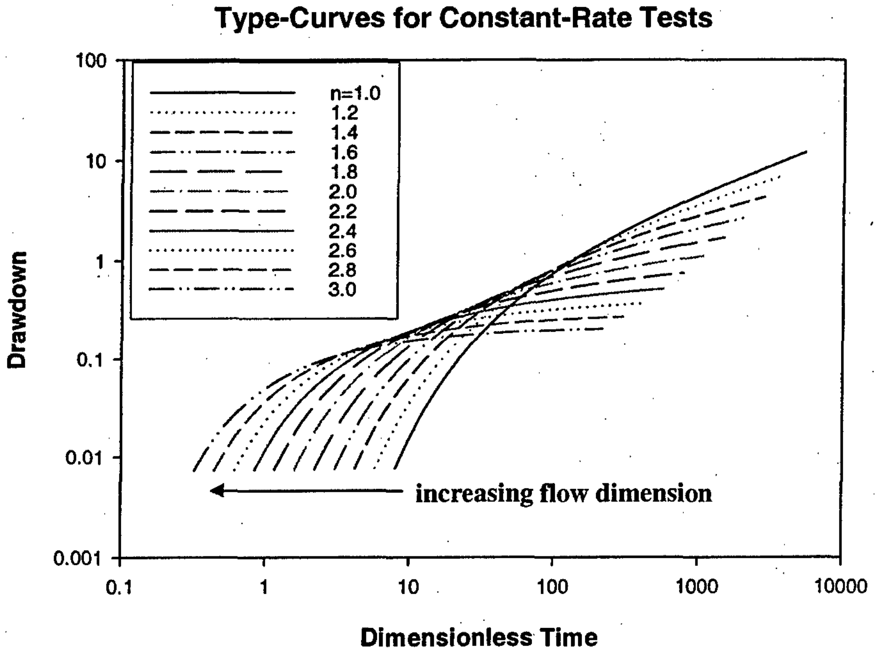


Figure 5.11 – Type-curves for constant-rate well tests (from Golder Associates Excel spreadsheet program FracDim)

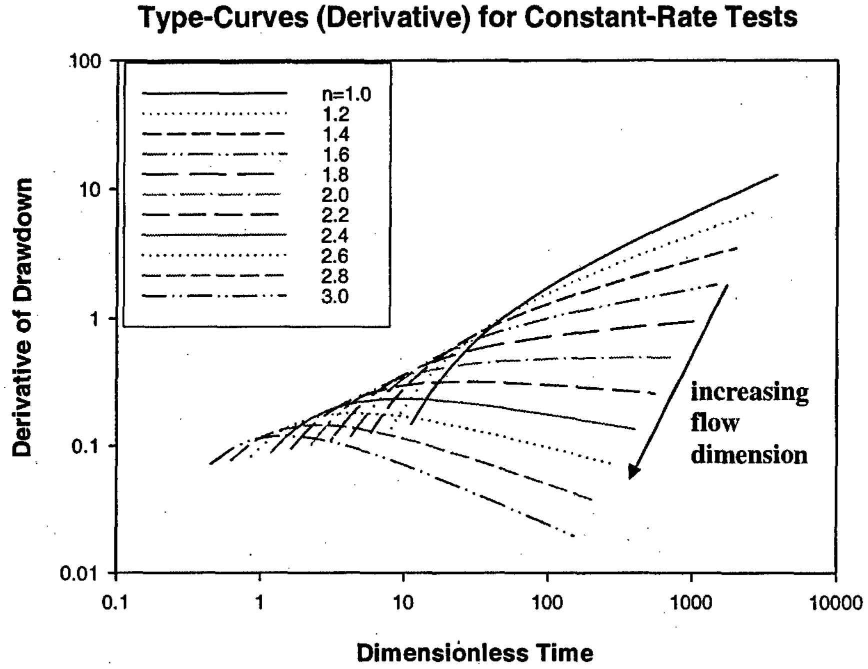


Figure 5.12 – Type-curves of the derivative of drawdown for constant-rate well-tests tests (from Golder Associates Excel spreadsheet program FracDim)

The type-curves in **Figure 5.11** and **Figure 5.12** are then used to analyze the drawdown (or pressure) data from actual or simulated well tests.

5.2 Flow Dimension Obtained Through Fracture Network Modeling and Finite Element Simulation

In order to determine the well-test flow dimension in a simulated fracture network accurately, simulated well tests need to be performed. The simulation results (e.g. drawdown at the well versus time) can then be analyzed using type-curves such as those given in **Figure 5.11** and **Figure 5.12** to obtain the flow dimension. The simulations performed in this report will mainly assume a constant rate condition at the well. Golder Associates Inc. has developed a flow modeling software called MAFIC that simulates flow in discrete fracture networks as opposed to using equivalent porous media models. MAFIC simulates flow in both the fracture network and the rock matrix. For our purposes, no flow will be allowed in the rock matrix in order to focus solely on the flow behavior of the fracture network. Three-dimensional networks of triangular finite elements are used to model the flow in the fractures.

For a nearly incompressible fluid such as water and for flow in two dimensions such as the flow in a fracture, the volume conservation equation takes the form

$$S\frac{\partial h}{\partial t} - T\overline{\nabla}^2 h = q$$

Equation 5.14

where S is the fracture storativity, h is the hydraulic head, T is the fracture transmissivity (the product of the hydraulic conductivity and the fracture aperture), q is a source or sink term, t is the time and $\overline{\nabla}^2$ is the two-dimensional Laplace operator. Storativity is defined as the volume of water that is expelled from storage per unit surface area under a unit change in head.

Equation 5.14 is solved using a Galerkin finite element solution scheme. A detailed discussion of the numerical procedure can be found in the MAFIC user manual.

Flow simulations using MAFIC involve the following steps:

1. A fracture network is generated using FracMan (Golder Associates fracture modeling software).

2. The geometry of the well (well radius, orientation, length, location) is incorporated into the generated fracture network. The outer boundaries are defined. Flow conditions (e.g. no-flow, constant rate, constant head) at the well and the outer boundaries of the modeling volume are prescribed. For the simulations in this report, the outer boundaries of the modeling volume are prescribed as constant head boundaries with $h = 0$. This is equivalent to the condition shown in **Figure 5.13** below.

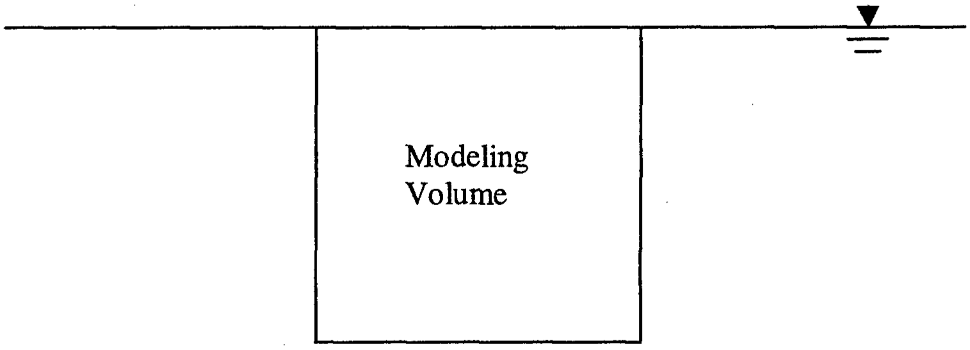


Figure 5.13 – Modeling volume submerged in water with top boundary coinciding with the water level. The modeling volume is fully saturated

At this stage, there is no flow in or out of the modeling volume since all heads are zero. If enough time is allowed after a well is drilled into the formation (the well is allowed to fill up with water), the steady state condition would also be that there is no flow in the modeling volume.

3. Using MAFIC, the steady state solution for the fracture network given the conditions in **step 2** is obtained. First, a group flux boundary condition is applied to the nodes of the elements that make up the well and then the steady state condition is solved for. Group flux boundary condition means than the assigned flux is divided among the nodes according to their respective transmissivities. This is different from the constant flux condition since the constant flux condition assigns the same flux to all the nodes in the boundary. Group flux therefore has the advantage of not overloading nodes that cannot support the assigned flow. The surface of the side of the well will be modeled using six panels with group flux (**Figure 5.14**) while the top and bottom of the well will be closed. Flux is negative for extraction and positive for injection.

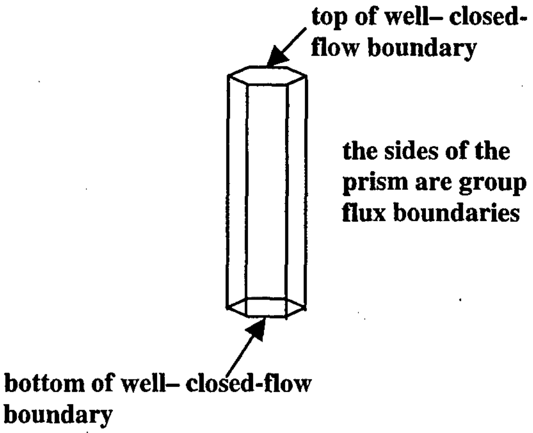

top of well– closed-flow boundary
the sides of the prism are group flux boundaries
bottom of well– closed-flow boundary

Figure 5.14 - The well is modeled as a prism with six panels and the nodes located on the panels are assigned group flux boundary conditions

4. A transient rate is applied at the well and the response (i.e. drawdown versus time) in the fracture network is recorded. Once the steady state condition has been established, a

transient rate larger than the initial group flux assigned is applied at the well. This transient rate is also assigned as a group flux boundary condition. The response is compared to type-curves (see **Figure 5.11** and **Figure 5.12**) to obtain the flow dimension.

Various file formats are required to run the flow simulations. The extensions of the required files are listed and their contents are described below. The detailed contents of these files can be found in the **Appendix**.

1. .SAB – This file contains the locations and geometry of the inner boundary (the well) and the outer boundary (boundaries of the modeling volume) as well as the boundary and initial conditions at the different parts of each boundary (constant flux, constant head, time-varying flux etc.). This file is usually called the "exploration" file (since it is also used to determine fracture spacing, locate pathways in the network, etc.).

2. .FAB – A file that contains the coordinates of the vertices of the fractures in the simulated fracture network. It also contains properties of the fractures such as transmissivity, storativity and aperture. Other fracture properties may also be included here. Fracture network simulations in FracMan produce .FDT files, which must be converted to .FAB files for use in flow simulations.

3. .MAF – The .SAB and .FAB files are combined using MeshMaster to produce .MAF files. MeshMaster converts the fracture network and the well into a finite element mesh. This file format can now be used in MAFIC. The .SAB files can be used in different fracture simulations (.FAB files) and vice versa. This feature makes it easier to apply different well and boundary conditions to a single fracture network or to apply a single set of well and boundary conditions to a number of fracture networks. The .MAF file includes the following information:

- The type of the output desired: summary or full. The summary output option is often enough.

- The number of nodes in the finite element mesh for which the head versus time values should be recorded in a separate file. The specific node numbers should also be indicated later in the file.

- The type of elements to be used in the flow simulations. There are two element types: linear and quadratic. Linear elements will be used in for the simulations in this report.

- A flag that indicates whether a restart file (restart.MAF) should be created. A restart file contains nodal flux and head values from the last time step of the current run. This file can be used as the input file for another stage of flow simulations. For example, one may desire to first simulate to a steady state condition reflecting the insertion of a well into the fracture network and allowing equilibrium and make this steady state condition the initial state from which to perform a constant rate test. The restart file in this case would include the head and flux information at all the nodes at the steady state and this, in turn, can become the starting point for another round of simulations (say, a transient rate is applied at the well).

- The type of solution required: transient or steady state. A transient solution will require the program to go through all the time steps that have been input. A steady state solution will ignore the time steps and go directly to the steady state.

- The tolerance for solution convergence (should be $\geq 1\times10^{-10}$) and the number of iterations that can be allowed in a timestep to reach convergence (should be > 0).

- The timesteps are also included. The desired type of output at the end of each timestep is indicated as well. The timesteps are ignored when steady state calculations are performed.

- The coordinates of the nodes and the initial flux and head at each node. The element numbers are also shown including the nodes that make up each element.

4. .MFT – The file format produced by editing the .MAF in EdMesh. After MAFIC is run using the .MAF file and a restart.MAF file is produced, the restart.MAF file must be modified for the next stage in the flow simulation (i.e. when going from the steady state to performing the actual well test) using the program EdMesh. In EdMesh, one can change boundary conditions for specific boundaries, enter the new timesteps for the next simulation stage and change the solution mode from steady state to transient. The .MFT file can be used directly in MAFIC to run the next stage of the flow simulation. This next stage, in turn, produces its own output file of the desired filename extension (for example: filename.OUT). The important output file contents include the heads and fluxes at all the nodes at the end of each timestep. For the purpose of monitoring the head versus time at a specific node, the user can also specify the creation of a plot.DAT file. The desired node numbers can be entered into the .MFT file so that MAFIC writes the head values at these nodes into the plot.DAT file at the end of each timestep. This is especially useful in determining the flow dimension using a constant rate well test (the drawdown vs. time is observed).

5. .SET – This file contains a record of all the changes made on the .MAF file using EdMesh. This file can then be used to run EdMesh in batch mode (i.e. there will be no need to enter the same changes all over again if one wants to modify another .MAF file). For example, the user has made changes to restart_1.MAF using EdMesh and wants to make the same changes to the file restart_2.MAF. The user does not have to type, say, all the same timesteps used to modify restart_1.MAF in order to change restart_2.MAF. The user only needs to change the target filename in the edmesh.SET file to "restart_2" and then type "ED_320K < edmesh.SET" at the command line (ED_320K is the executable).

The following sections discuss three variants of an approximate approach that estimates the flow dimension using the distance-flow width relationships in a fracture network. These will then be verified using finite element flow simulations on the same fracture networks using the same well geometry.

5.3 Flow Dimension Obtained Through Fracture Network Modeling and Distance-Conductance Relationships

In the previous approach, the determining the flow dimension involved computationally intensive flow simulations on the simulated fracture networks. Another approach, developed by Dershowitz, estimates the flow dimension directly from the geometry of the fracture intersections. First, the intersections between the fractures are determined then a graph theory search is conducted from the well out into the fracture network in order to create the path of the flow. Along the path of the flow, the change in flow area or conductance is recorded. The log of flow area or conductance is plotted versus the log of the tortuous radial distance and the flow dimension is estimated from the slope of this plot.

To explain this approach, we start with the relationship between the flow area and the radial distance from the well, which is given by $A_f = k_p r^{n-1}$ where A_f is the flow area, r is the radial distance, n is the flow dimension and k_p is a constant of proportionality. Taking the logarithm of each side gives

$$\log A_f = (n-1)\log r + \log k_p$$

Equation 5.15

This is the slope-intercept form for the equation of a line, $y = Sx + y_i$. So the slope of this line is equal to $n-1$. The flow dimension is thus

$$n = S + 1$$

Equation 5.16

The issue now becomes how these flow areas (and consequently the conductance) are defined. Dershowitz (1996) proposed an approach that converts a three-dimensional fracture network into an equivalent three-dimensional network of one-dimensional pipes. These pipes connect the midpoints of the intersections along a path and the lengths are the distances between the connected intersections. Each pipe is also assigned a conductance (or flow area) based on the lengths of the intersections or "traces" as well as the orientation of the pipe connecting them. This method of defining the flow area or conductance is described in the next section (**Section 5.3.1**). Two other approaches are also discussed: flow areas based on individual intersections and flow areas based on cubical elements with assigned flow areas.

5.3.1 Equivalent Pipes Using Pairs of Intersections

The original approach in defining the flow area given by Dershowitz calculates the flow width based on the pairs of intersections along the path of the flow. The flow width shown in **Figure 5.15** is based on the intersection lengths, L_{1i} and L_{2i}, and is given by

$$W_i = \frac{1}{2} F_i (L_{1i} + L_{2i})$$

Equation 5.17

where W_i is the flow width and F_i is a channeling factor. The flow area is W_i multiplied by the fracture aperture.

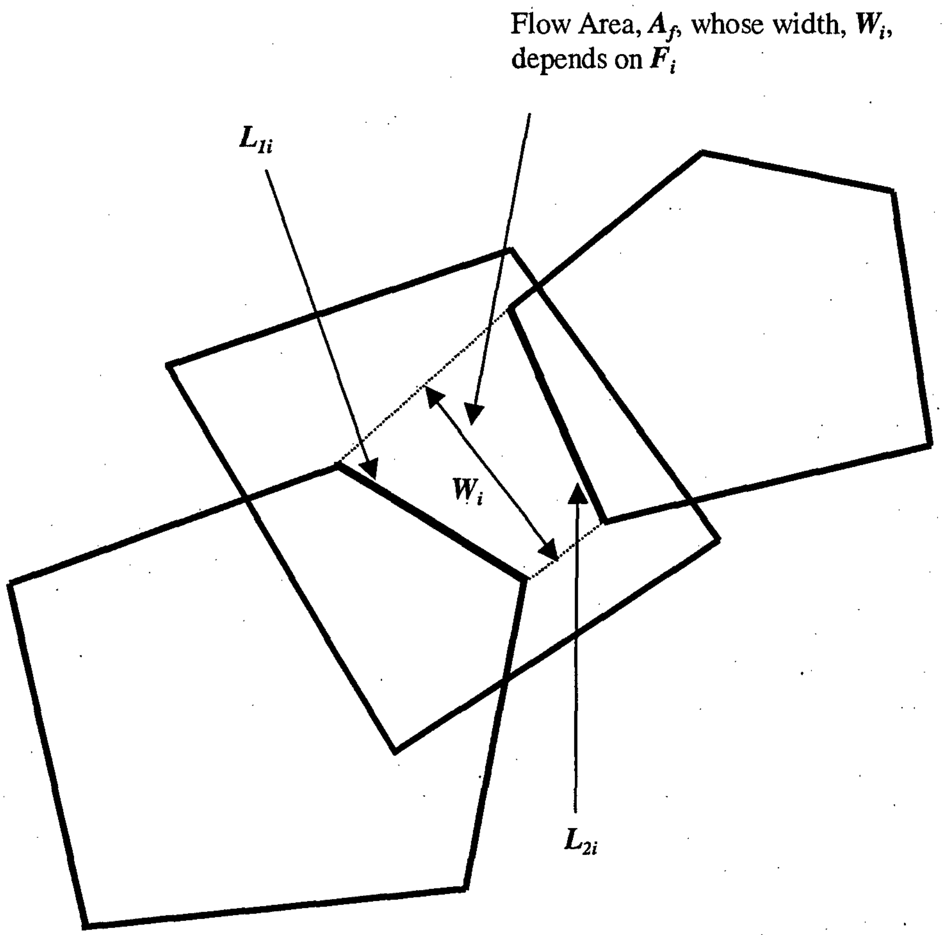


Figure 5.15 – Flow width as a function of the lengths of two adjacent intersections

The flow width at a certain distance is a function of the lengths of two consecutive intersections in a path. First, the vector connecting the midpoints of the two intersections is calculated. Then the length of the projection of each intersection onto the line perpendicular to the vector is calculated (**Figure 5.16**). The average of these two projections is the flow width at that distance.

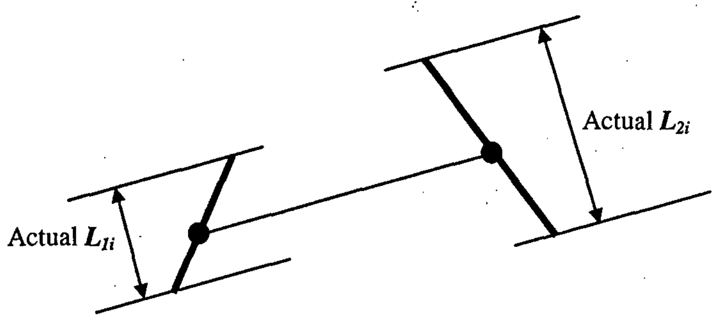


Figure 5.16 – The actual lengths used are the projections of the intersection onto a line perpendicular to the vector connecting the midpoints of the two intersections

The averaging procedure gives rise to a possibility that a very short intersection will be paired with a very long intersection thereby masking the effect of the short intersection.

In the next section, an approach that defines the flow widths based on the length of individual intersections is presented.

5.3.2 Equivalent Pipes Using Single Intersections

Another way of defining the flow width is by using the lengths of individual intersections as the flow widths themselves (**Figure 5.17**). This avoids the averaging procedure and derives the distance-conductance relationship based on the lengths of the intersections as they are. **Figure 5.17** shows that at a radial distance of d_1, the flow width is given by $W_1 = L_{1i}$. At a radial distance of $d_1 + d_2$, the flow width is $W_2 = L_{2i}$. Very small or point intersections will be represented as well as very large intersections.

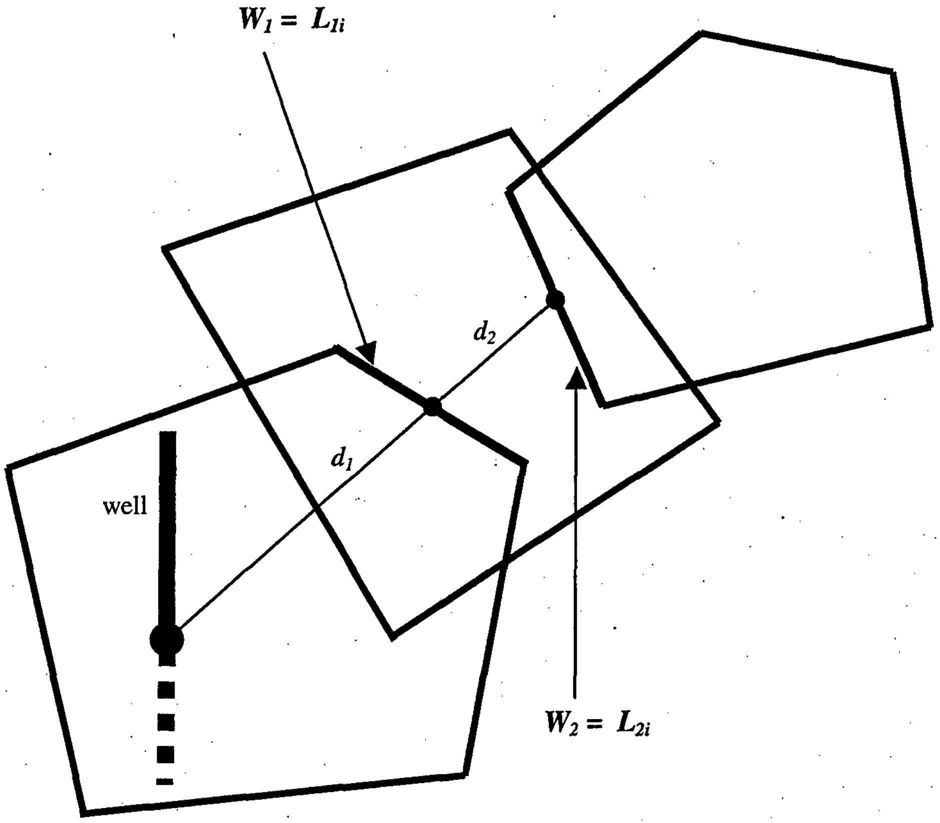


Figure 5.17 – Flow width as a function of the individual intersection lengths

The next proposed method of defining the flow width is aimed at determining how coarse the assignment of radial distance as well as flow area can be. In this approach, the modeling volume is divided into smaller cubical elements.

5.3.3 Cubical Element Approximation

The third estimating approach is aimed at determining the coarseness with which the fracture intersections are considered in order to arrive at a reasonable estimate of the flow dimension. First, the modeling volume is subdivided into smaller cubical elements (**Figure 5.18**).

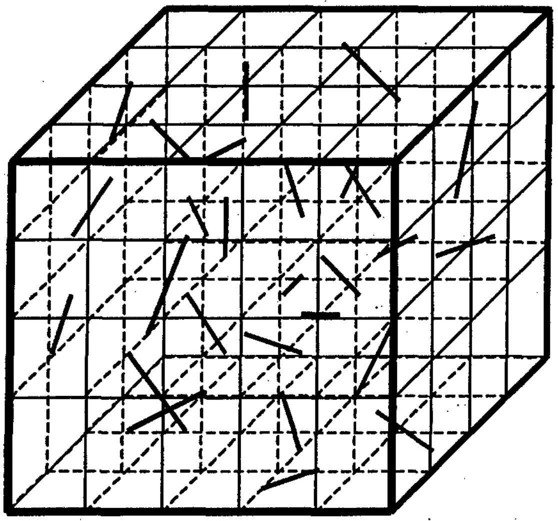


Figure 5.18 – The modeling volume is divided into smaller cubical elements

The fractures that intersect the well are located and the paths from the well into the fracture network are traced (**Figure 5.19**). These paths connect the fracture intersections to each other. The fracture intersections along the paths are then assigned to their respective cubical elements (**Figure 5.20**). A fracture intersection belongs to a cubical element if the midpoint of the intersection is located within the cubical element. Then, according to the order in which the intersections appear in the paths that were traced, the corresponding cubical elements are also arranged in an equivalent path. Along the equivalent path, the distances are composed of the distances between the centers of the cubical elements (**Figure 5.21**). The distances from the well to the first accessed cubical elements are also computed (**Figure 5.22**). Correspondingly, the flow width for a specific cubical element is the total length of the intersections assigned to it.

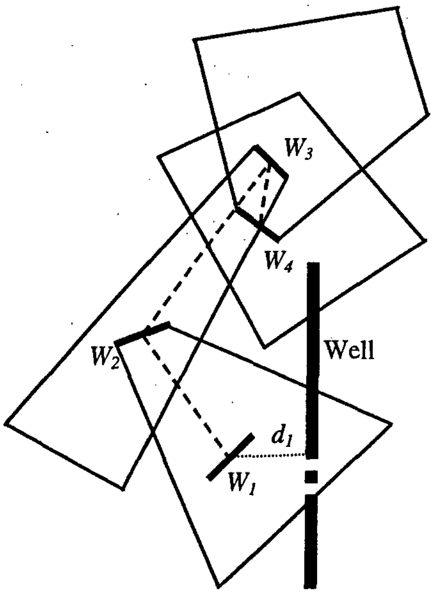


Figure 5.19 – The path from the well into the fracture network is traced and the intersections located

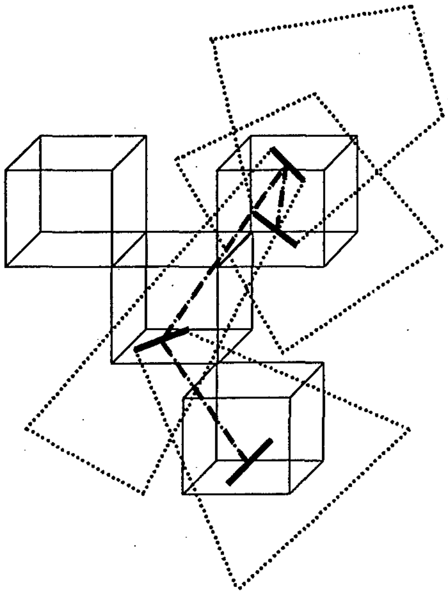


Figure 5.20 – The individual intersections are assigned to their respective cubical elements. An individual intersection is assigned to the cubical element if its midpoint is located within that element

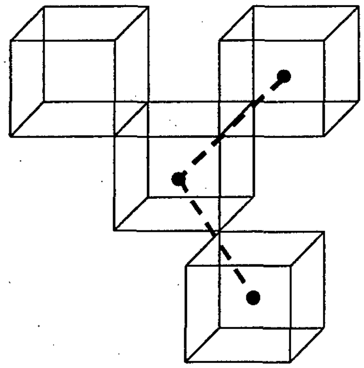


Figure 5.21 – The distances between connected cubical elements are computed. This will now represent the radial distance into the fracture network

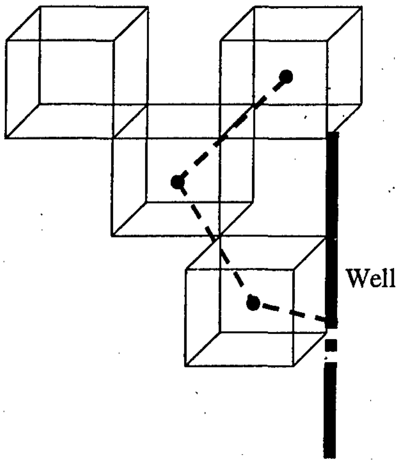


Figure 5.22 – The distance-flow width relationship is calculated and the flow dimension is derived from this relationship

The absolute values of flow dimension resulting from this approach can be expected to be higher than those from the single intersection method (**Section 5.3.2**) especially for fracture networks that have high intensity and for a coarse division of the modeling volume. When the modeling volume is divided coarsely highly tortuous paths are being replaced by straight lines (the distances between the centers of two cubical elements, for example: the path in **Figure 5.20** is replaced by **Figure 5.21**) and this is combined with a lumping together of flow widths (assigning intersections to cubical elements, see also **Figure 5.20**). These lead to a different distance-flow width relationship compared to the one derived using the single intersection method. Therefore, there will be a higher rate of change of flow width with respect to radial distance and this will lead to larger absolute values of flow dimension (since flow dimension is the slope of the log flow width-log radial distance relationship plus one). However, the flow dimension values also depend on two other things: the manner in which the flow-width is calculated for a certain interval of radial distance and the resulting behavior of the log-log plot of flow-width versus radial distance since linear regression is needed in order to estimate the flow dimension. This will become more apparent in the latter part of **Section 5.4** which presents the results for this method.

The next section shows the results of the application of the distance-conductance (or flow width) approaches presented above and their comparison with results from finite-element simulations.

5.4 Results and Discussion

In order to determine the accuracy of the different methods of estimating the flow dimension presented in **Section 5.3**, fracture systems will be generated to compare the flow dimension estimates from distance-flow width relationships derived using the original method (equivalent pipes using pairs of intersections, **Section 5.3.1**), individual intersection method (**Section 5.3.2**) and the cubical element approach (**Section 5.3.3**) to those obtained using finite-element simulations. Five fracture networks will be simulated in 200 by 200 by 200 meter modeling volumes using the FracMan modeling software from Golder Associates. The same mean fracture orientations in **Table 4.2** (page 65) will be used to generate the five fracture networks in the verification but different size and orientation distributions as well as fracture intensities will be used due to the much larger modeling volume ($200^3 m^3$ here compared to 20^3 m^3 in **Chapter 4**). Increasing the size (equivalent radius) of the fractures and decreasing the intensity will make the simulations less time consuming. The modeling parameters are given in **Table 5.1** below.

Set	Trend*	Plunge*	Number	Equivalent Radius (m)
A	8°	2°	30	40
B	220°	10°	30	40
C	80°	2°	30	40
D	110°	20°	60	40

Table 5.1 – Modeling parameters for the five fracture networks. *Mean pole orientation, trend is clockwise from North. The number of fractures from set D is assumed to be twice that of any of the other sets because set D has been observed to be the most abundant. For the pole orientations, the Fisher distribution is used for each set and the exponential distribution is assumed for the size distribution

Two well geometries will be used to obtain the distance-flow width relationships, a 10-meter long well and a 180-meter long well both having a 0.75 m radius. The location and orientation of each well are shown in **Figure 5.23** and **Figure 5.24**. The well in **Figure 5.23** can be thought of as a section that has been isolated using packers.

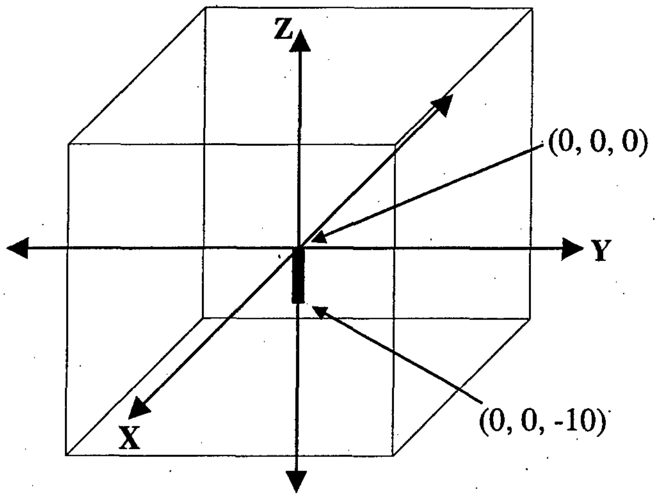


Figure 5.23 – 10 meter well location (modeling volume is 200 by 200 by 200 meters)

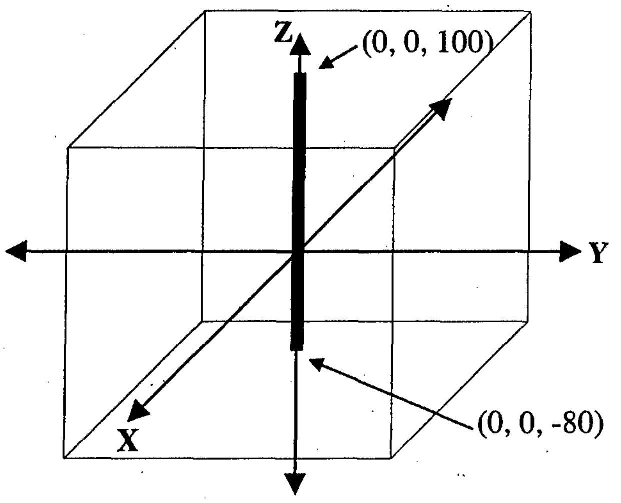


Figure 5.24 – 180 meter well location (modeling volume is 200 by 200 by 200 meters)

In the succeeding pages, plots of the distance-flow width relationships for each fracture network using each of the two well geometries are presented. The estimates of the flow dimension are also included in the plots. The label "original algorithm" refers to the method discussed in **Section 5.3.1** where equivalent pipes are created using pairs of intersections. The label "individual intersections" refers to the approach in **Section 5.3.2** where equivalent pipes are based on the lengths of single intersections.

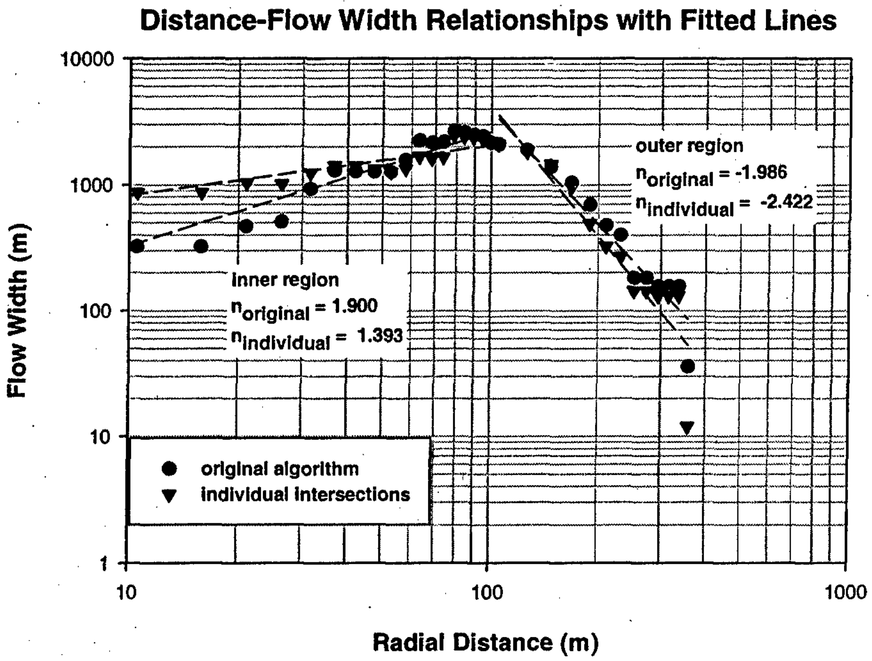


Figure 5.25 – Distance-flow width relationships for fracture network simulation 1 using a 10 meter long well

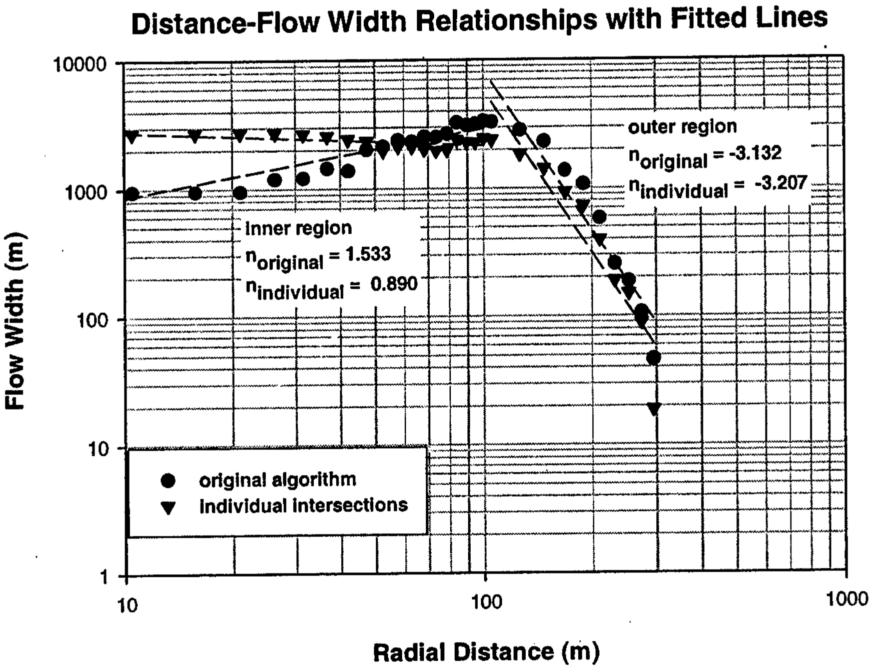


Figure 5.26 – Distance-flow width relationships for fracture network simulation 2 using a 10 meter long well

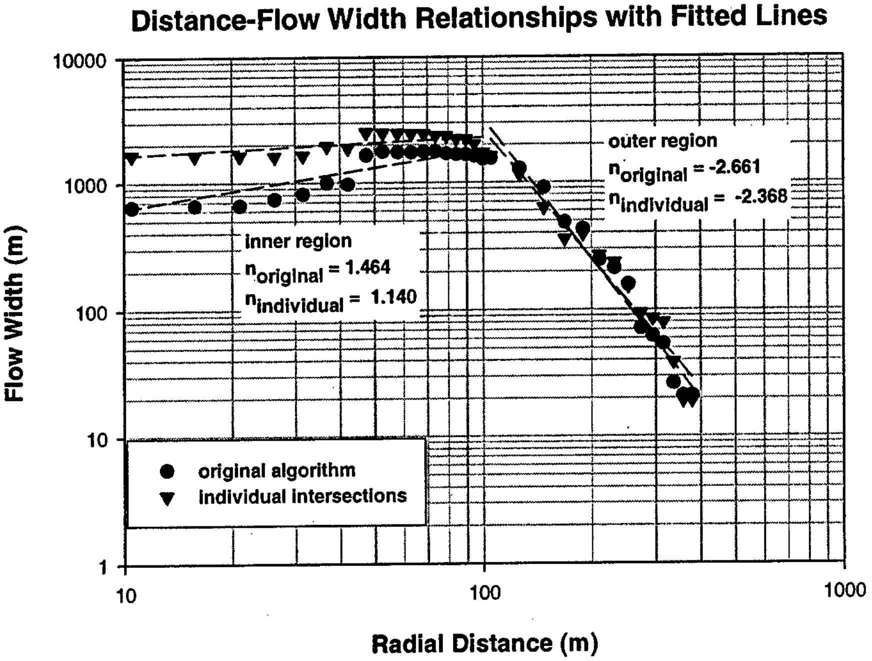


Figure 5.27 – Distance-flow width relationships for fracture network simulation 3 using a 10 meter long well

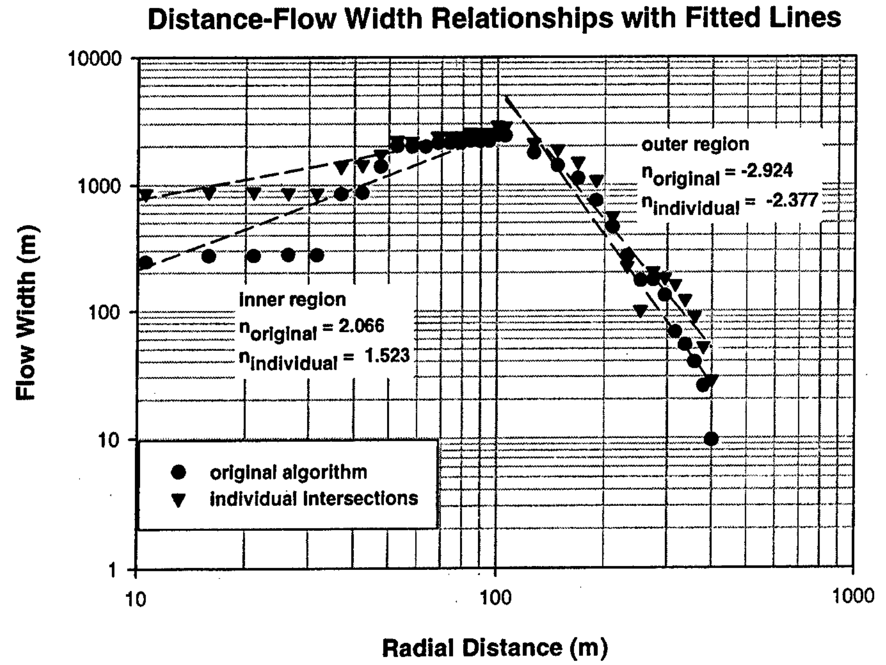


Figure 5.28 – Distance-flow width relationships for fracture network simulation 4 using a 10 meter long well

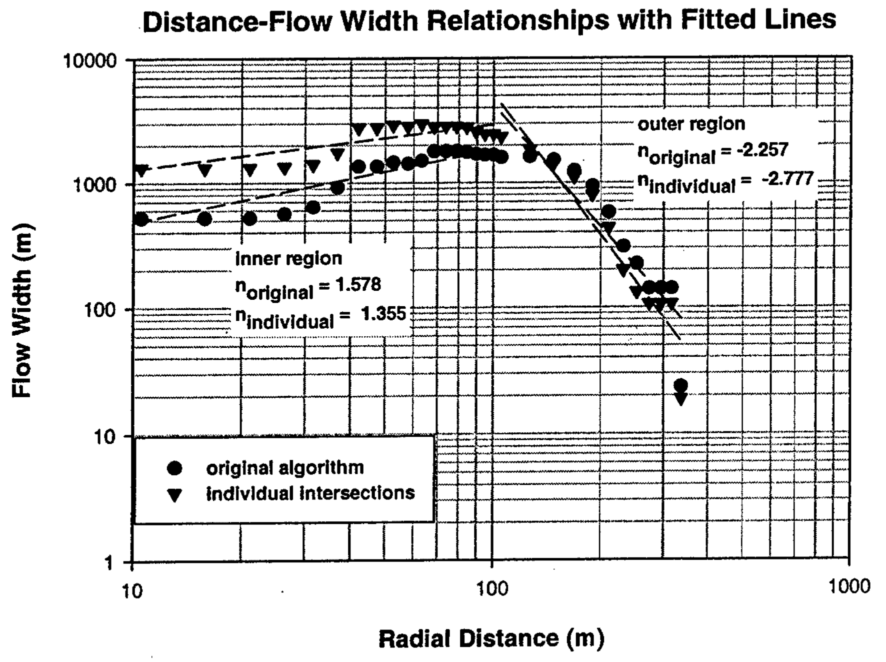


Figure 5.29 – Distance-flow width relationships for fracture network simulation 5 using a 10 meter long well

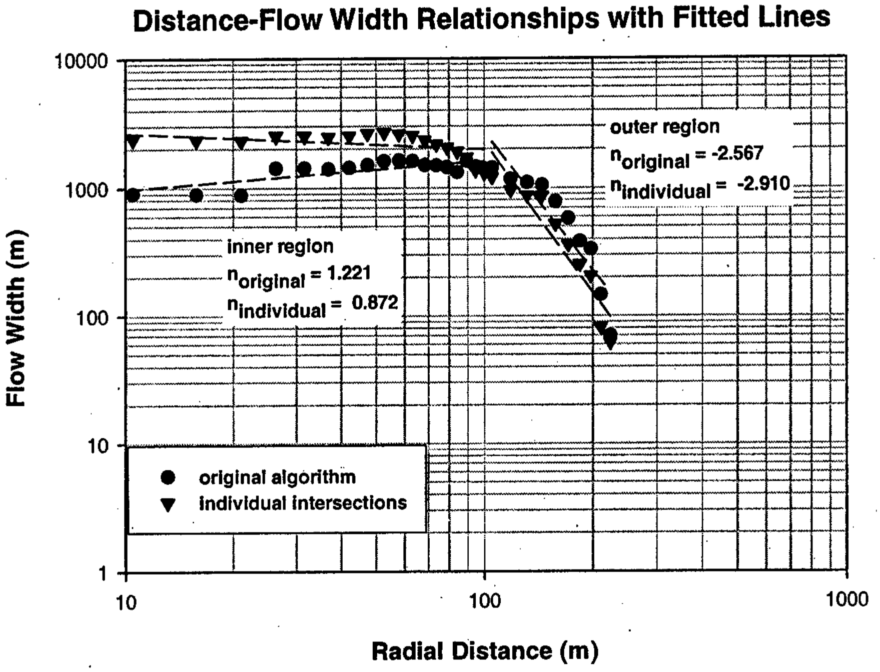


Figure 5.30 – Distance-flow width relationships for fracture network simulation 1 using a 180 meter long well

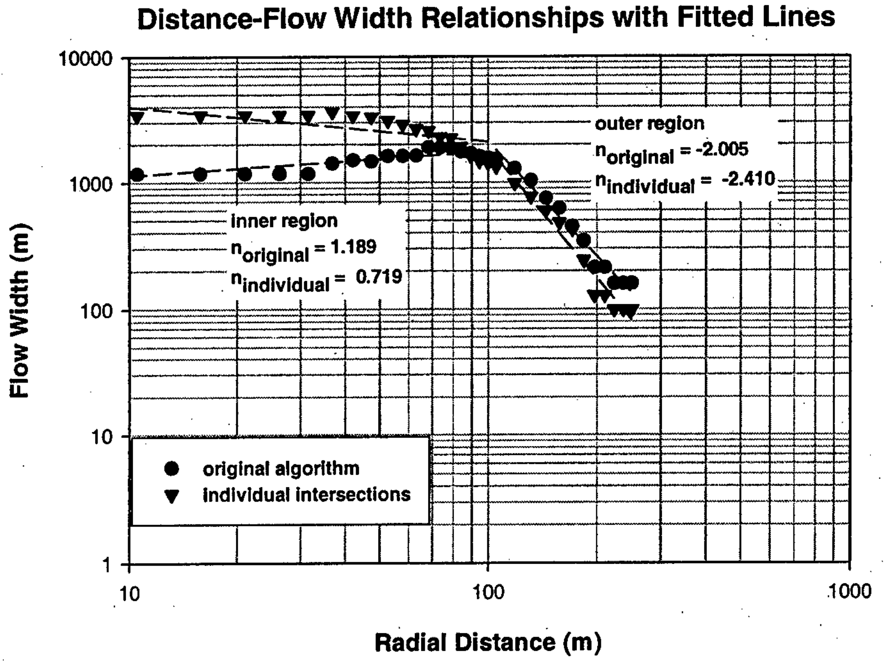


Figure 5.31 – Distance-flow width relationships for fracture network simulation 2 using a 180 meter long well

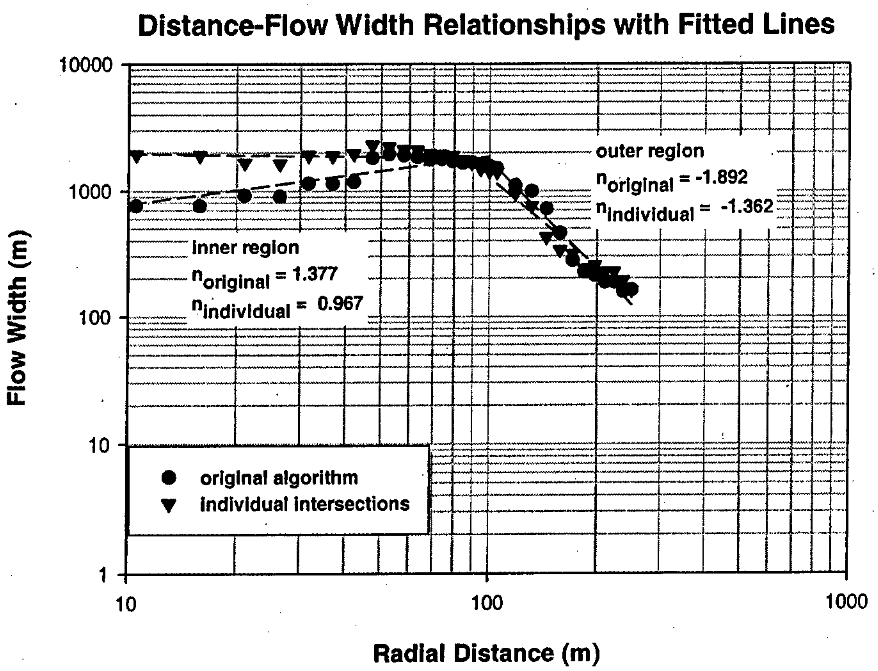


Figure 5.32 – Distance-flow width relationships for fracture network simulation 3 using a 180 meter long well

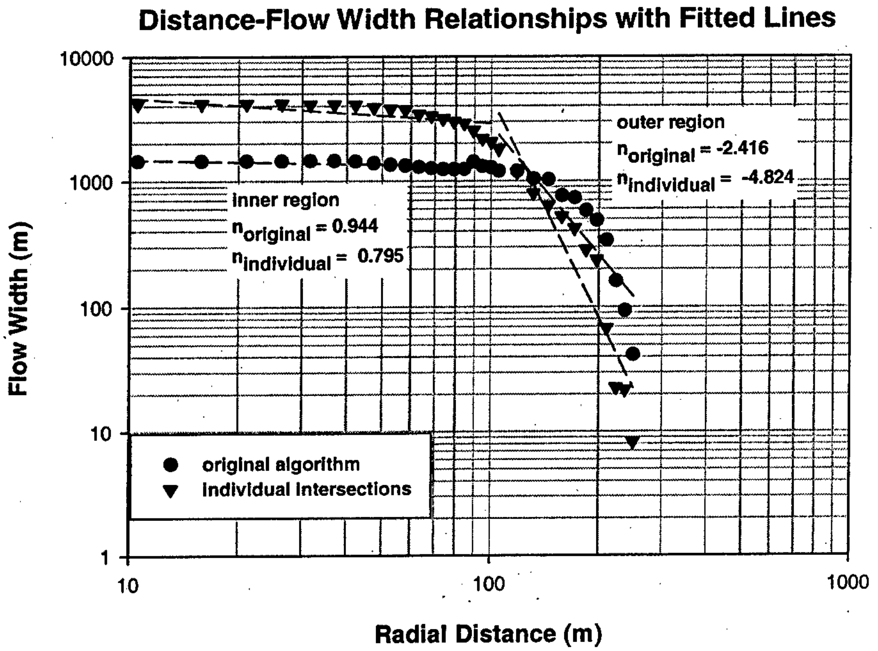


Figure 5.33 – Distance-flow width relationships for fracture network simulation 4 using a 180 meter long well

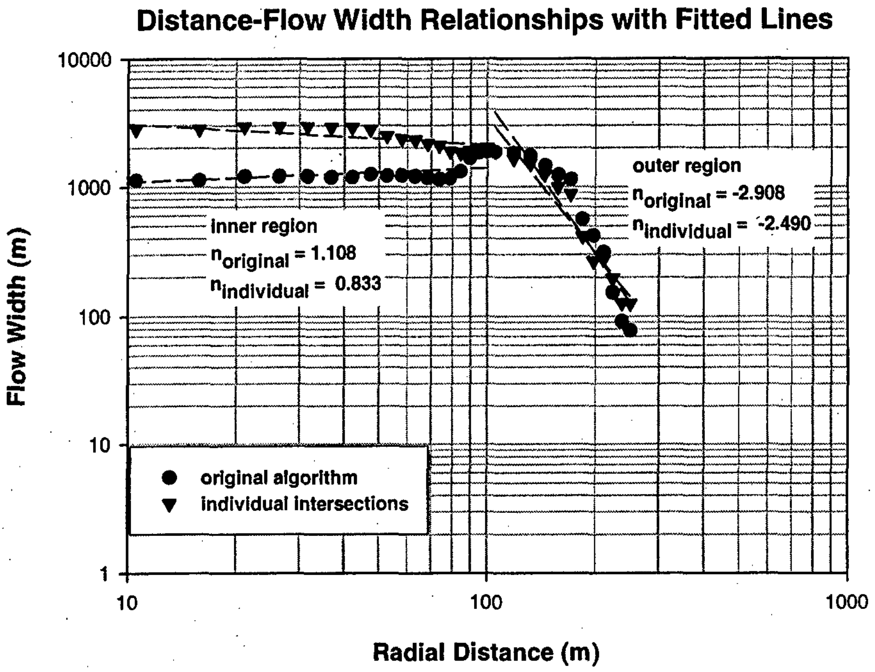


Figure 5.34 – Distance-flow width relationships for fracture network simulation 5 using a 180 meter long well

The results shown in **Figure 5.25** to **Figure 5.34** all suggest that the fracture network is made up of two regions from the standpoint of flow dimension. There is an inner region where the flow width varies from increasing to slightly decreasing with radial distance. In the outer region, the flow width always decreases with radial distance. The figures also show that the individual intersection method almost always gives a flow width that is greater than the original method. This can be expected since the original method uses the average of the projected lengths of two intersections (see **Figure 5.16**). However, at certain points on the plots the flow width from the original algorithm overtakes that of the individual intersection algorithm. For the most part of the outer region (or after the flow width from the original method overtakes that of the individual intersection method), the original method gives a larger flow width than the individual intersection algorithm. A series of intersections that decrease in length with distance from the well like that shown in **Figure 5.35** can help understand this observation. At a distance $d_1 + d_2$, the original method would give a flow width of $\frac{1}{2}(L_1 + L_2)$ (let us assume that the path is perpendicular to the intersections for simplicity). The individual intersection algorithm would give a flow width of L_2, which is smaller than $\frac{1}{2}(L_1 + L_2)$. The same can be said about the flow width at a distance $d_1 + d_2 + d_3$ because the intersection length is decreasing with distance from the well. Note that the very important restriction in this explanation is that the path is perpendicular to the intersections. If not, then the intersection lengths can be reduced substantially when using the original method and even the average of these reduced intersection lengths may not be enough to surpass the length of one of the intersections. So, the fact that the flow width using the original method exceeds that of the individual intersection method in the outer region **may** also suggest how the paths and the fracture intersections along those paths are oriented relative to each other.

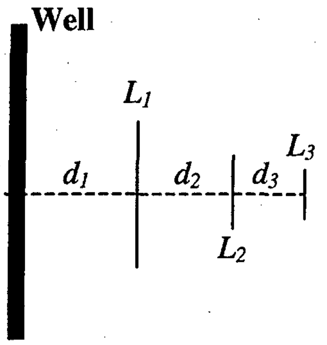


Figure 5.35 – Intersection lengths decreasing with distance from the well

For the inner region, the original algorithm always gives a higher value of flow dimension compared to the individual intersection method. Both the original and individual intersection algorithms gave larger flow dimension values for the inner region when the short (10m) well was used than when the long (180m) well was used. One can imagine, however, that for fracture networks, using a shorter well length may not always lead to a larger flow dimension since a shorter well may access a lesser number of fractures that may, in turn, connect to less extensive networks.

In the outer region, there is no consistent difference between the flow dimension values given by the two approaches. The original algorithm may give a higher flow dimension compared to the individual intersection method in one network and then give a lower value in another network. Results for the outer region were also inconsistent when comparing the short and long well flow dimension values in the sense that short well flow dimension values were not always larger than long well flow dimensions. However, radial distance-flow width relationships for the outer region consistently show the reduction of flow width with radial distance.

The results for the cubical element approximation are presented next. The degree of refinement for this approach is described in terms of the number of cubical elements the entire modeling volume is divided into or the number of boxes per edge. Six levels of coarseness for dividing the modeling volume are used in determining the distance-flow width relationships: 1000, 200, 50, 20, 10 and 5 boxes per edge. For each network and for each of the two well geometries (10 and 180 meter long wells), two flow dimension values are computed representing the flow dimension values of the inner region and outer region of the fracture network. The values of the inner and outer flow dimensions versus the level of coarseness (number of boxes per edge) are given in **Table 5.2** and **Table 5.3**. The plots of the distance-flow width relationships where these values were derived can be found in the **Appendix**.

Short Well	Network 1		Network 2		Network 3		Network 4		Network 5	
Boxes/Edge	n_{inner}	n_{outer}	n_{inner}	n_{outer}	n_{inner}	n_{outer}	n_{inner}	n_{outer}	n_{inner}	n_{outer}
1000	1.413	-1.311	0.935	-2.318	1.833	-1.531	1.478	-0.969	1.573	-1.063
200	1.434	-1.244	0.935	-2.170	1.788	-1.307	1.406	-1.238	1.570	-1.235
50	1.455	-1.207	0.879	-1.768	1.664	-1.382	1.682	-1.420	1.572	-1.403
20	1.455	-1.207	1.512	-2.231	1.192	-1.250	1.359	-1.708	1.735	-1.877
10	1.455	-1.207	0.482	-1.579	1.591	-1.170	0.929	-1.636	1.843	-1.589
5	1.455	-1.207	0.172	-0.626	1.143	-0.484	0.117	-1.152	0.605	-1.770

Table 5.2 – Flow dimension values for the inner and outer regions using the cubical element approach for each of the networks and for each level of coarseness. The short (10 m) well is used here

Long Well	Network 1		Network 2		Network 3		Network 4		Network 5	
Boxes/Edge	n_{inner}	n_{outer}	n_{inner}	n_{outer}	n_{inner}	n_{outer}	n_{inner}	n_{outer}	n_{inner}	n_{outer}
1000	1.827	0.125	0.865	-2.123	0.772	-2.304	2.616	-3.610	2.113	-1.928
200	1.768	-0.069	0.748	-2.140	0.889	-2.211	2.653	-3.552	2.126	-1.861
50	1.797	-0.105	0.677	-2.080	0.825	-2.041	2.518	-3.780	2.083	-1.918
20	1.658	-0.702	1.366	-1.503	1.046	-1.855	2.223	-3.308	2.264	-1.915
10	1.673	-0.737	2.605	-1.703	1.042	-1.415	2.161	-2.577	1.694	-2.038
5	1.750	0.111	0.274	-0.790	0.341	-0.061	2.252	-3.452	0.783	-0.500

Table 5.3 - Flow dimension values for the inner and outer regions using the cubical element approach for each of the networks and for each level of coarseness. The long (180 m) well is used here

Based on the results, one can see the inconsistency with which the cubical element method calculates the flow dimension. One would expect the flow dimension for the longer well to be less than the flow dimension using the short well for the same fracture network, a trend in the simulated fracture networks that has been observed in the results given by the original and individual intersection approaches. However, **Table 5.2** and **Table 5.3** show that this is not always the case. Another noticeable characteristic of the distance-flow width relationships using the cubical element (see plots in the **Appendix**) method that is different from the two other methods presented earlier is that the flow width does not increase or decrease in the same steady way. As a matter of fact, the flow width is shown to change abruptly with radial distance. This is due to the fact that the algorithm used to calculate the flow width at a certain radial distance for the cubical element method is different from those of the two previous ones. In all three approaches, an algorithm that calculates the paths from the well into the fracture network is employed. These paths, of course, are made up of the line segments connecting successive fracture intersections. After this step, the distances to each intersection along these paths are determined. The pieces of information are then stored as pairs of data: the length of the intersection and the radial distance to the intersection. The distance-flow width relationships are then derived from these data pairs. It is at this stage that the cubical element method differs from the other two approaches. The cubical element approach determines the total flow width at a certain interval of radial distance and not beyond that. For example, if the user specifies an interval of 10 meters, the algorithm calculates the flow width that exists between say 150 and 160 meters. If the interval is sufficiently small there is a chance that the algorithm finds very small or zero total flow width at an interval. The two previous methods calculate the total flow width at an interval and sometimes beyond that if they cannot find a flow path that is sufficiently long. For example, if along a path the algorithm cannot find a value of radial distance that falls within the 150 to 160 meter interval, it takes the first value of flow width along the same path that corresponds to a radial distance that may already be greater than 160 meters. This is why the relationship between the flow width and the radial distance using such an approach behaves better. **Figure 5.36** to **Figure 5.39** show plots of the calculated flow dimension versus the level of coarseness (i.e. how coarse the subdivision of the modeling volume is).

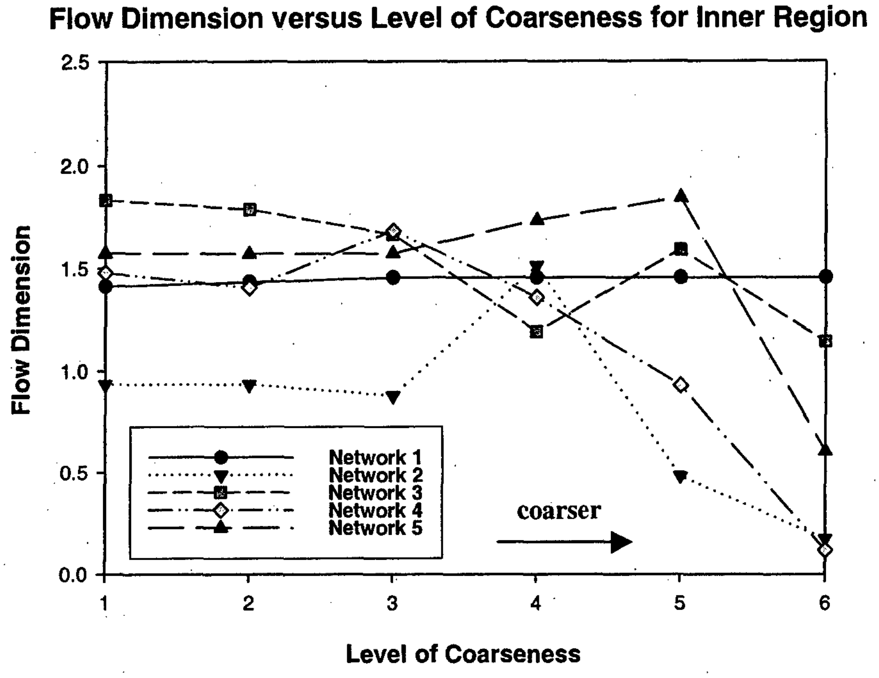


Figure 5.36 – Variation of the estimated flow dimension for different levels of coarseness in the inner region of the simulated fracture networks using the short well (10 m)

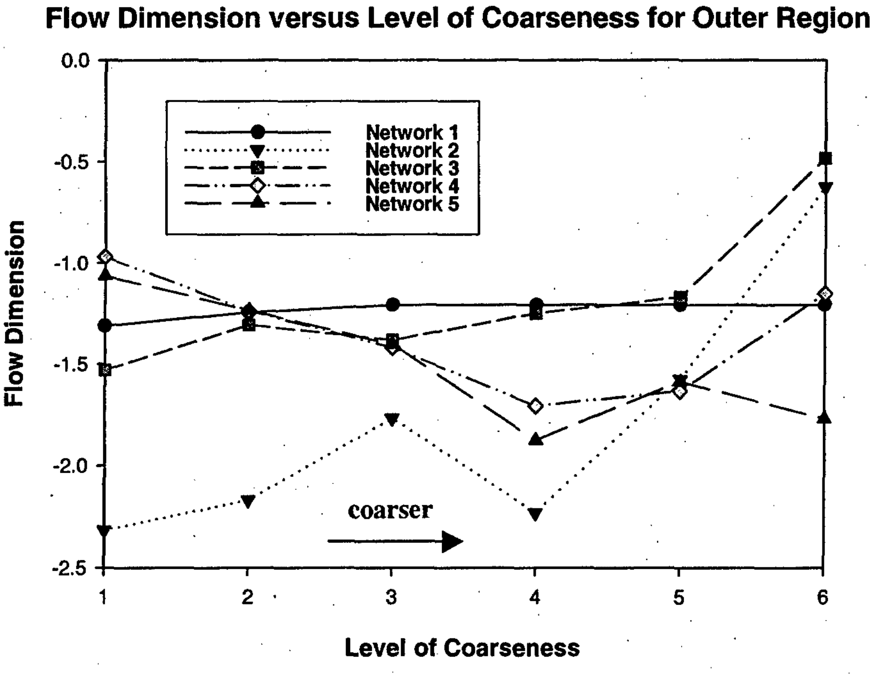


Figure 5.37 – Variation of the estimated flow dimension for different levels of coarseness in the outer region of the simulated fracture networks using the short well (10 m)

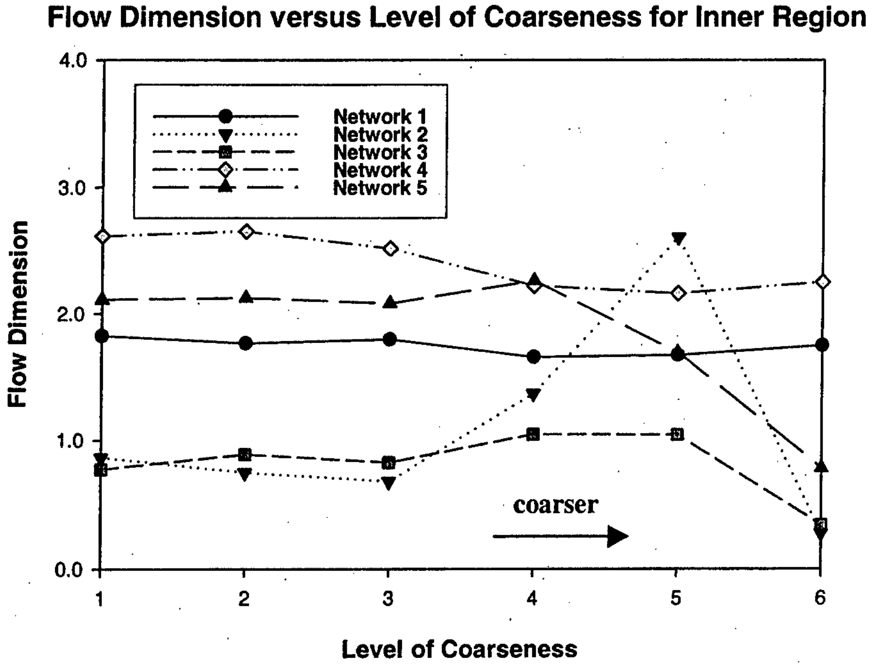


Figure 5.38 – Variation of the estimated flow dimension for different levels of coarseness in the inner region of the simulated fracture networks using the long well (180 m)

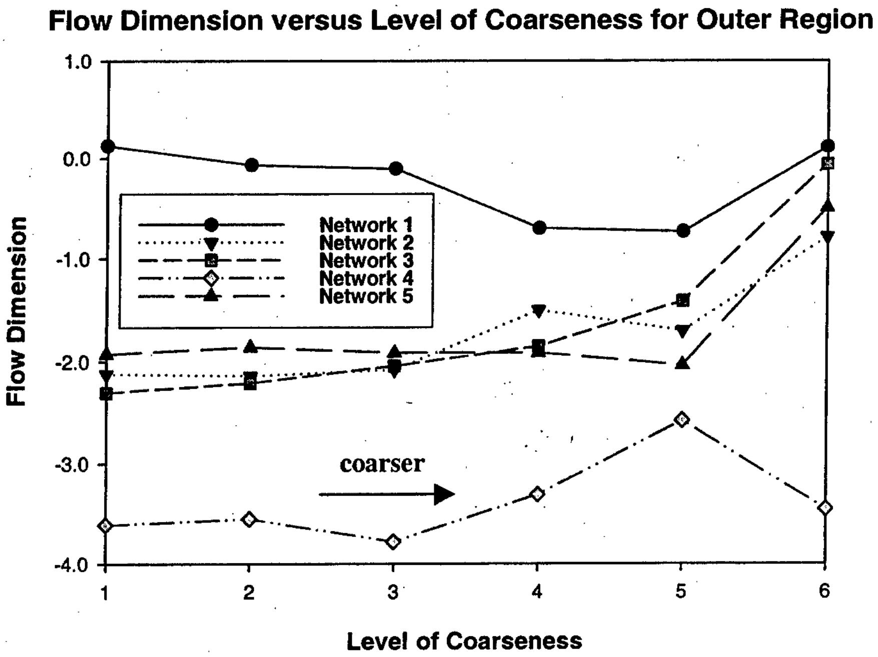


Figure 5.39 – Variation of the estimated flow dimension for different levels of coarseness in the outer region of the simulated fracture networks using the long well (180 m)

The six levels of coarseness in the plots in **Figure 5.36** to **Figure 5.39** are the following: **1** – 1000, **2** – 200, **3** – 50, **4** – 20, **5** – 10, and **6** – 5 boxes per edge. The number of boxes per edge is the number of cubical elements that can fit on one edge of the modeling volume. So for 5 boxes per edge, there will be 5^3 or 125 cubical elements in all.

The plot in **Figure 5.36** shows that the flow dimension values for the inner region generally become lower as the modeling volume division becomes coarser. However, significant increases in the flow dimension also occur between the finest and coarsest divisions. **Figure 5.36** also suggests the unpredictability in the resulting flow dimension estimate when the number of boxes per edge becomes less than 50 (level 3 in the plot). The flow dimension suddenly rises or drops after this level of coarseness. **Figure 5.38** also shows a general lowering of the flow dimension values as the coarseness increases but not to the same extent as in **Figure 5.36** when a shorter well was used. Slight increases or decreases can also be observed in **Figure 5.38** when the coarseness exceeds 50 boxes per edge.

Figure 5.37 shows no definite trend in the values of the flow dimension as the division becomes coarser. The flow dimension may end up higher or lower at 5 boxes per edge than it was at 1000 boxes per edge. In between these divisions, there is no specific trend. However, when the length of the well is increased to 180 meters (**Figure 5.39**), the flow dimension at 5 boxes per edge is always higher than the flow dimension at 1000 boxes per edge (very slightly for networks 1 and 4). Also, **Figure 5.39** suggests that a division of 50 boxes per edge can be used and a slightly larger increase or decrease in the flow dimension can be expected after that (when going from 50 to 20 boxes per edge). **Figure 5.39** also shows that the approach behaves relatively well (the flow dimension does not change much) between 1000 and 50 boxes per edge for the longer well.

At its present state, the cubical element method is not as good as either the original or the individual intersection algorithms in defining the large-scale behavior of the flow width with radial distance. It is too sensitive to sudden changes in the flow area and this can be blamed largely on the algorithm used to calculate the flow width at a given distance (as explained previously). It also has the inherent handicap of calculating the radial distance as the sum of the distances between centers of connected cubical elements. The real path in the network may actually be more tortuous than that. There also lies the problem of what coarseness to use for a fracture network. However, the capability of this method in actually estimating the flow dimension has to be evaluated using comparisons with results from finite element simulations.

Perhaps another way one could utilize the cubical elements is to change how they are used to determine the flow dimension. One could use them in the manner shown in **Figure 5.7**. The cubical elements are considered the conductive elements just as the squares were in **Figure 5.7**, and one would find out how many of them are intersected at certain distances from the source. The number of elements intersected versus the distance from the source would then give an estimate of the flow dimension. The total length of intersections in a cubical element should also be considered so that some cubical elements are more conductive than others.

In order to verify the results of the flow dimension estimates, finite element flow simulations need to be run on the same networks. The details of these simulations have already been presented in **Section 5.2**.

Before the finite element program, MAFIC, was used to determine the flow dimensions for the fracture networks, two test cases were run. The fracture pattern used for the first case was a large single fracture perpendicular to the well extending to the boundaries of the modeling volume. The flow dimension for this case is 2.0 since the flow area at a certain distance is proportional to the radial distance, *r*. The second fracture pattern used is a single fracture that is very long in one direction and narrow in the other. Such a pattern will result in a flow dimension of 1.0 since the flow area does not change with distance from the well. Several nodes at various distances from the well were chosen in each case and constant rate was applied at the well. In each case, MAFIC gave head data that produced the anticipated flow dimension.

Network	$n_{original}$	$n_{individual}$	n_{1000} bxs/edge	n_1	n_2	n_3	n_4	n_5
1	1.900	1.393	1.413	3.0	2.6	3.2	3.2	3.0
2	1.533	0.890	0.935	3.0	2.2	1.8	3.0	2.4
3	1.464	1.140	1.833	2.2	3.0	2.8	2.4	2.8
4	2.066	1.523	1.478	2.8	3.2	3.2	3.2	3.0
5	1.578	1.355	1.573	1.8	1.8	1.8	1.8	2.4

Table 5.4 – Short well flow dimension values from the three previous approaches (first three columns) compared to those from finite element simulations using nodes located not more than 10m from the well. The head values at five different nodes are observed for each network

Network	$n_{original}$	$n_{individual}$	n_{1000} bxs/edge	n_1	n_2	n_3	n_4	n_5
1	1.900	1.393	1.413	4.2	4.0	4.0	2.6	2.8
2	1.533	0.890	0.935	4.2	3.0	4.0	2.6	3.6
3	1.464	1.140	1.833	2.8	3.8	3.6	2.0	2.6
4	2.066	1.523	1.478	2.6	3.6	2.8	2.6	4.2
5	1.578	1.355	1.573	3.0	3.4	3.2	3.4	3.2

Table 5.5 – Short well flow dimension values from the three previous approaches (first three columns) compared to those from finite element simulations using nodes located more than 10m but not more than 50m from the well. The head values at five different nodes are observed for each network

Table 5.4 and **Table 5.5** show the flow dimension values from finite element simulations compared with those obtained using the original algorithm ($\mathbf{n_{original}}$, see **Section 5.3.1**), the individual intersection approach ($\mathbf{n_{individual}}$, see **Section 5.3.2**) and the cubical element approach for 1000 boxes per edge ($\mathbf{n_{1000\ bxs/edge}}$, see **Section 5.3.3**). Since constant flow rate well tests were simulated, the head values at each timestep were observed at five different nodes chosen from each network ($\mathbf{n_1}$ to $\mathbf{n_5}$). It can be observed that the flow dimension values from the finite element flow simulations are always larger than those predicted using any of the three approximate methods. For the short well case, the approximate method that gives the flow dimension values closest to the finite element simulations is the original algorithm. The original approach gives flow dimension values that are, on the average, about 33% lower than the finite element value for nodes 10 meters from the well or closer. The individual intersection approach gives flow dimension values that are 50% lower than the finite element results while the cubical element method gives values that are about 42% lower. So for the short well, the original approach provided the results that were closest to the finite element flow dimension values. It has also been observed that when nodes that are located between 10 and 50 meters from the well are used for monitoring the head, the resulting flow dimensions are higher than when the nodes are located within 10 meters of the well (compare **Table 5.4** and **Table 5.5**). On the average, the flow dimension values derived from nodes in this region are 28% higher than the flow dimension values for nodes within 10 meters of the well. Note that the locations of the nodes chosen have been spread out with respect to radial and vertical distance from the well in order to make the flow dimension results as representative of the region as possible.

Network	$n_{original}$	$n_{individual}$	n_{1000} bxs/edge	n_1	n_2	n_3	n_4	n_5
1	1.221	0.872	1.827	2.0	2.2	2.8	2.0	2.2
2	1.189	0.719	0.865	2.0	2.0	2.0	2.0	2.0
3	1.377	0.967	0.772	2.0	3.2	2.2	2.0	2.2
4	0.944	0.795	2.616	2.6	2.8	2.8	2.8	2.8
5	1.108	0.833	2.113	x	2.6	2.0	x	2.0

Table 5.6 – Long well flow dimension values from the three previous approaches (first three columns) compared to those from finite element simulations using nodes located not more than 10m from the well. The head values at five different nodes are observed for each network. Cells marked with "x" indicate cases where there is not enough head data to obtain a type-curve fit

Network	$n_{original}$	$n_{individual}$	n_{1000} bxs/edge	n_1	n_2	n_3	n_4	n_5
1	1.221	0.872	1.827	3.6	3.6	4.0	3.8	3.8
2	1.189	0.719	0.865	3.8	3.6	3.6	3.6	3.6
3	1.377	0.967	0.772	3.6	2.8	3.2	2.8	3.2
4	0.944	0.795	2.616	3.8	3.0	3.2	3.0	3.0
5	1.108	0.833	2.113	2.4	3.0	2.8	4.0	1.8

Table 5.7 - Long well flow dimension values from the three previous approaches (first three columns) compared to those from finite element simulations using nodes located more than 10m but not more than 50m from the well. The head values at five different nodes are observed for each network.

For the long well case (**Table 5.6** and **Table 5.7**), the cubical element gives results closest to those of the finite element simulations. For nodes that are at most 10 meters from the well, the cubical element approach gives flow dimension values that are 29% lower than finite element simulation values. The original and individual intersection algorithms give flow dimension values that are 47% and 63% lower than the finite element results, respectively. As in the short well case, the flow dimension values from the finite element simulations increase when nodes located between 10 and 50 meters from the well are used to observe the head. On average, the flow dimension values are 45% larger in this interval than they are in the 10 meter region.

The finite element results almost always show a lower flow dimension for the long well case compared to the short well. This trend is also reflected in both the original algorithm and the individual intersection algorithm. The cubical element method is erratic in this respect since the consistency of the results depends largely on how the modeling volume is divided. Overall, the three approximate approaches underestimate the connectivity of the fracture networks based on the flow dimension. For the short well, it is best to use the original approach, which defines the flow width as a function of the average of two consecutive intersection lengths. However, one must anticipate that this will be lower than the finite element flow dimension. For the long well, the cubical element method performs well compared to the two other approaches. However, this approach did not perform consistently as evidenced by the results for networks 2 and 3 in **Table 5.6** and **Table 5.7** where it grossly underestimated the finite element flow dimension.

6 Summary and Conclusions

The development of a simple algorithm for calculating the fracture intersections was presented and implemented in the fracture modeling program GEOFRAC. The program was then used to model the fracture networks in the Boston Area in order to study the geometry of individual fracture intersections. For convenience, the pertinent plots from **Chapter 4** are repeated near the end of this section. The simulations showed that the mean fracture intersection length varies between 0.5m and 0.7m with the mean equivalent fracture radius equal to 0.770m (**Figure 4.6**). The results do not suggest a trend between the mean intersection length and the mean fracture area and this also appears to be true between the standard deviations of intersection length and fracture area (**Figure 4.8** and **Figure 4.9**). However, a distinct relationship between the intersection length per unit volume and the fracture intensity was observed (**Figure 4.10**). The slope of the relationship increases with increasing fracture intensity, P_{32} (m^2/m^3). The same trend is observed between C_1 (the number of fracture intersections per unit volume) and the fracture intensity, P_{32} (**Figure 4.11**). The results also show that at fracture intensities of about 0.005 or lower, no intersections occur among the simulated fractures (**Figure 4.10** and **Figure 4.11**). The number of isolated fractures was also observed to increase initially as the total number of fractures increases, it then plateaus (**Figure 4.12**) and can be expected to decrease as the total number of fractures increases further. The plots of the relative frequencies of intersection lengths show that there is a low occurrence of large intersections (>2.0 meters) and that the frequency of such intersections increases with the size of the modeling volume (**Figure 4.14** to **Figure 4.18**). The occurrence of intersections that are 10 cm or shorter range from 11% of the total number of intersections for the largest modeling volume to 13% for the smallest modeling volume. The decrease in the relative frequency of the small intersections is accompanied by an increase in the relative frequency of the large intersections. As the size of the modeling volume increases, the relative frequencies for a specific value of intersection length do not change much from one simulation to another. The calculated orientations of the intersections between the fracture sets (A to D) using mean their mean orientations predict that a majority of these intersections will have a northwest trend (**Figure 4.27**). The northwest trending intersections were predicted to range from horizontal to almost vertical (**Figure 4.28**). Plots of the trend and plunge of the simulated intersections show the same trend (**Figure 4.24** to **Figure 4.26**) for all modeling volumes. The shapes of the simulated orientation distributions (**Figure 4.24** to **Figure 4.26**) also agree with the general shape given by the calculated intersection orientations among the fracture sets (**Figure 4.27** to **Figure 4.28**).

The simulations show that the algorithm for calculating the fracture intersections gives GEOFRAC the capability of modeling individual fracture intersections. This capability enables the user to study the length distribution of the intersections as well as their orientation distribution. The intersection data can also be used to assess the connectivity versus fracture intensity of the fracture sets (**Figure 4.10** and **Figure 4.11**).

Fracture connectivity was also described using flow behavior through the network by means of a parameter called the well-test flow dimension. Fracture networks were generated and the flow was simulated in each of them using two well geometries (short and long). The flow dimension values were then estimated using the distance-flow width relationships of the fracture networks. Three approaches for the calculation of the flow width were presented and are enumerated here for convenience: 1) The original algorithm derives the flow width as a function of the average of two consecutive intersections. 2) The individual intersection method assumes that the flow width is the length of an individual intersection. 3) The cubical element method subdivides the modeling volume into smaller boxes each of which is assigned a flow width value based on the total length of intersections whose midpoints are located within them. The resulting flow dimension values were compared to those from finite element flow simulations. In the following, the observations for the approximate methods are summarized first then the results of the comparison will follow. Results for the cubical element method will be discussed separately from the original and individual intersection methods.

Both the original and individual methods show that each simulated fracture network is made up of two regions. There is an inner region where the flow area increases (and in some cases, slightly decreases) with radial distance and an outer region where the flow area decreases rapidly with radial distance (**Figure 5.25** to **Figure 5.34**, pages 112-117). The original approach gave larger flow dimension values compared to the individual intersection approach for the inner region (**Figure 5.25** to **Figure 5.34**). Higher flow dimension values for the inner region were also observed in both these methods when the short well was used compared to when the long well was used. In the outer region, the approximate methods did not show a consistent difference between the two methods but both consistently reflect the rapid decrease in flow area with radial distance that occurs there.

Unlike the two other approximate approaches, the cubical element method does not consistently show that using the short well will result in a higher flow dimension compared to using the long well (**Table 5.2** and **Table 5.3**, page 119). One can also see that the flow width-radial distance relationship from the cubical element method behaves differently from the other two approaches. This behavior is due to the way it calculates the total flow width at an interval. The flow dimension results were presented versus the level of coarseness; a description of how coarse the modeling volume is subdivided expressed as the number of boxes per edge (**Figure 5.36** to **Figure 5.39**, pages 121-122). The results revealed that for the inner region (for both short and long wells), the calculated flow dimension becomes lower as the model subdivision becomes coarser. The flow dimension values for the inner region suddenly rise or drop when the number of boxes per edge falls below 50 for the short well (**Figure 5.36**). The same behavior is also apparent using the long well at the same point although the rise or drop is not as sharp as in the short well (**Figure 5.38**). In both cases, the flow dimension for the coarsest subdivision, 5 boxes per edge, is always lower than that of the finest subdivision, 1000 boxes per edge (**Figure 5.36** and **Figure 5.38**). The flow dimension results for the outer region using the short well do not indicate a specific trend with the coarseness of the subdivision (**Figure 5.37**). The flow dimension at the coarsest subdivision is not always higher or lower than the flow dimension at the finest subdivision. For the long well, however, the results show that flow dimension for the coarsest subdivision is always higher than the flow dimension for the finest subdivision (**Figure 5.39**). Also, the method appears to behave well between 1000 and 50 boxes per edge for the case of the long well (**Figure 5.39**).

The results from the approximate approaches were compared to finite element flow simulation results to determine how each method performed in estimating the flow dimension. The head values at different times were monitored at different nodes in the fracture network. These head data are then plotted against type-curves in order to obtain the flow dimension. In the case of the short well, the approximate method that gave flow dimension values closest to those of finite element flow simulations is the original algorithm (**Table 5.4** and **Table 5.5**, page 125). On the average, the original method produced flow dimension values that were 33% lower than the finite element values. The individual intersection and cubical element methods gave flow dimensions that were, on the average, 50% and 42% lower than the finite element results, respectively. On the other hand, the long well results showed that among the three approaches, the cubical element method values came closest to the finite element simulation flow dimension values (**Table 5.6** and **Table 5.7**, page 126). Cubical element flow dimension values were, on average, 29% lower than the finite element results. Note, however, that in two of the fracture networks the cubical element method gave very poor estimates of the flow dimension values (**Table 5.6** and **Table 5.7**).

The finite element flow simulations indicated that long well flow dimension values were lower than their short well counterparts for the fracture networks used. This same trend is evident in the results from both the original and individual intersection algorithms (**Table 5.4** to **Table 5.7**). The cubical element method is not consistent in this respect. In some cases, the long well flow dimension is lower than the short well flow dimension. In the other cases, it is the opposite.

The results show that there is not a single approximate method that performed the best in each of the simulated fracture networks. However, dividing the results into short and long well revealed that in each case, one approximate method outperforms the other two. The original method is most suitable for short wells and the cubical element method for long wells. Common among the approximate methods is that they all underestimate the flow dimension, and therefore the connectivity, when compared to the finite element simulation results.

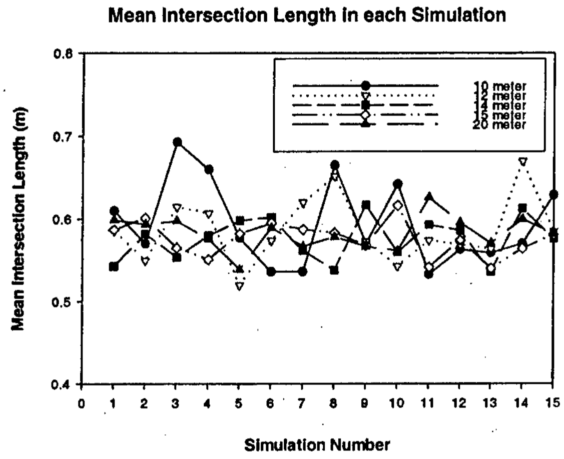


Figure 4.6

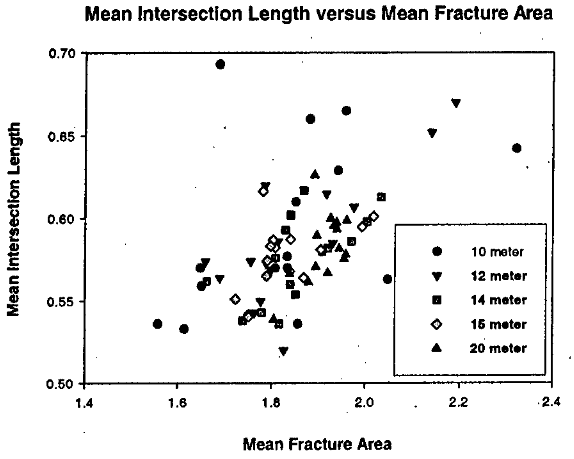


Figure 4.8

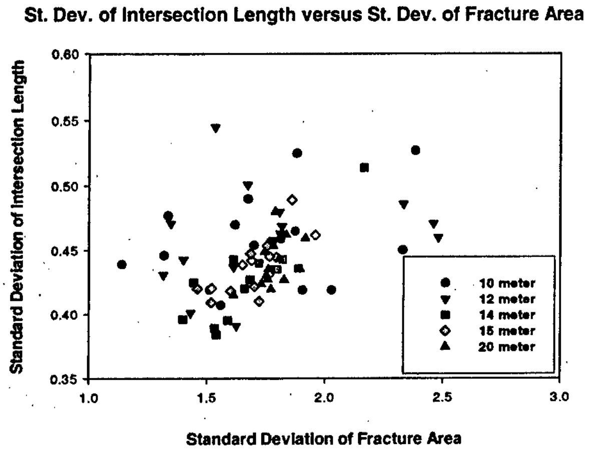


Figure 4.9

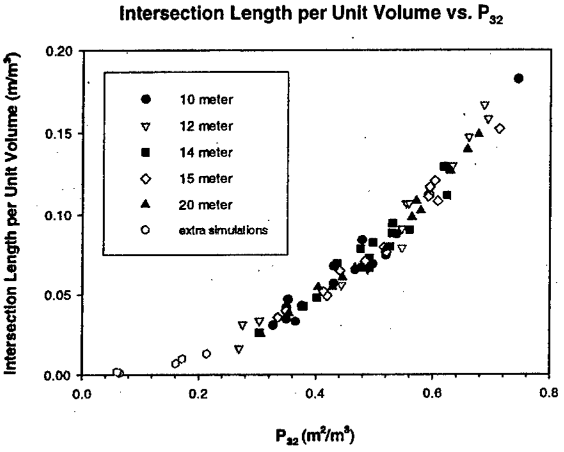


Figure 4.10

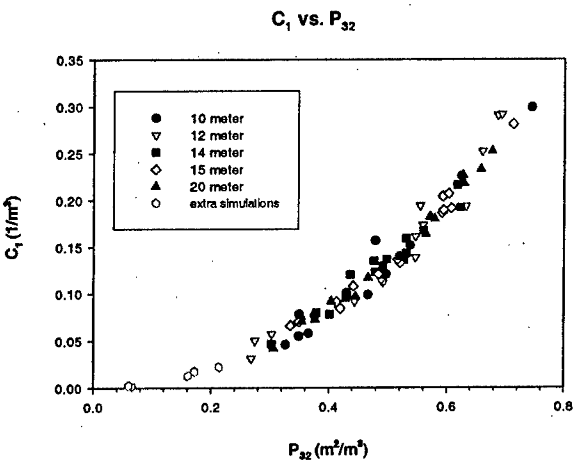


Figure 4.11

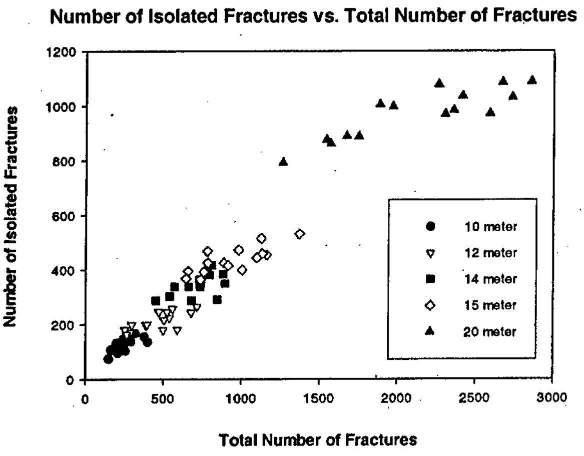


Figure 4.12

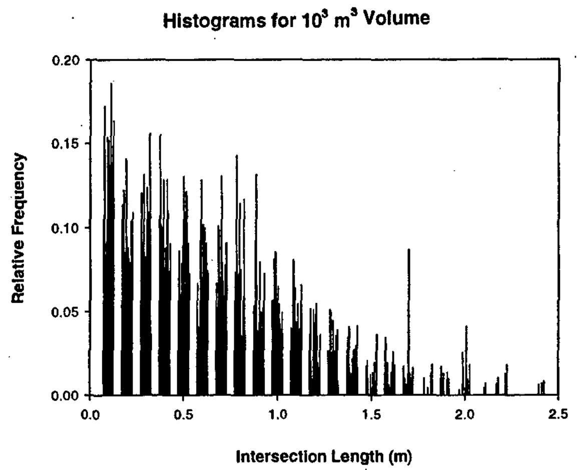


Figure 4.14

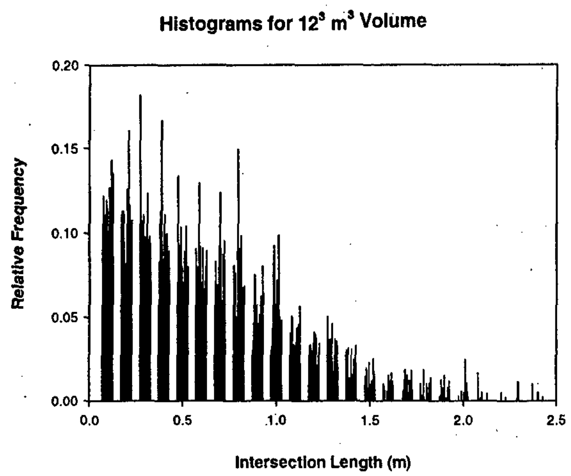


Figure 4.15

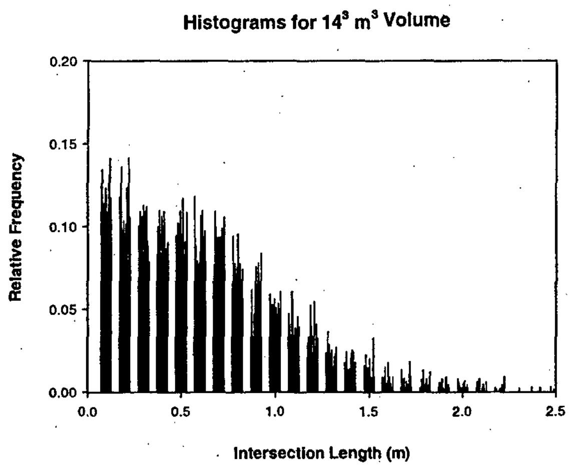


Figure 4.16

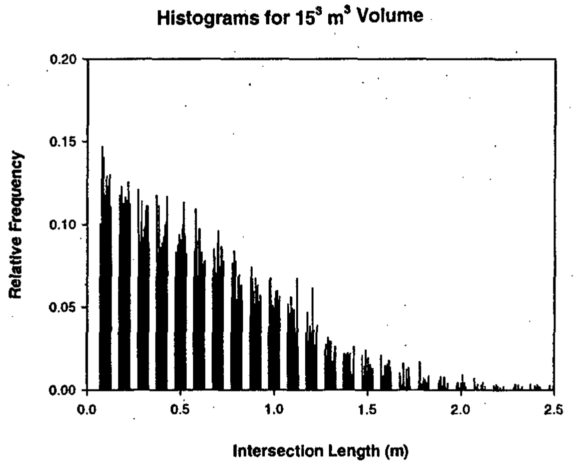


Figure 4.17

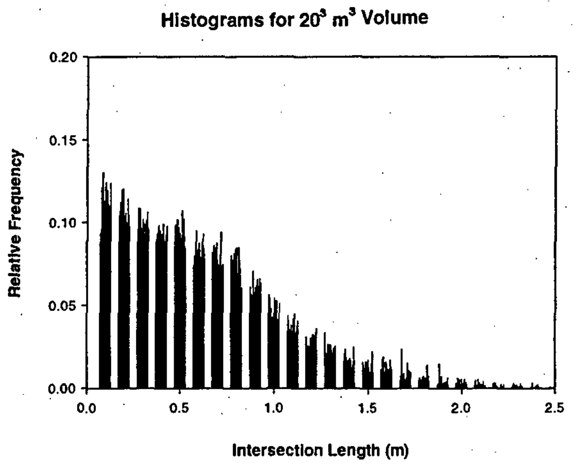
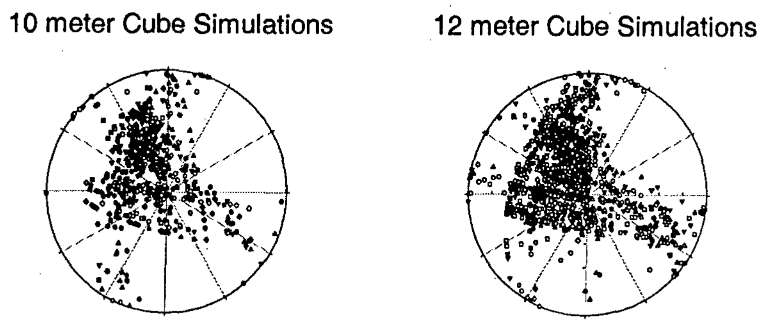


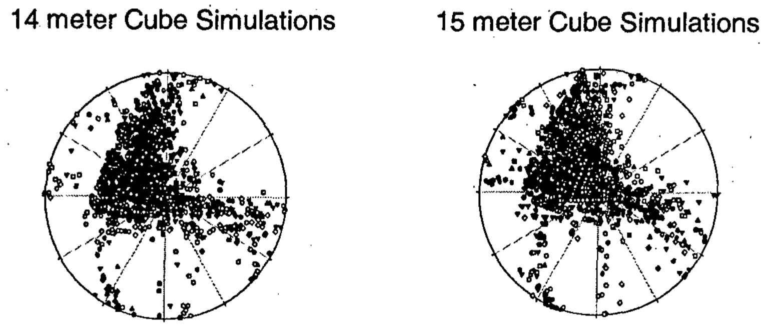
Figure 4.18



10 meter Cube Simulations

12 meter Cube Simulations

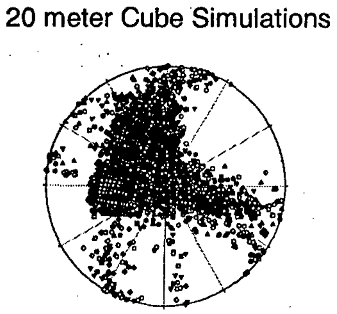
Figure 4.24



14 meter Cube Simulations

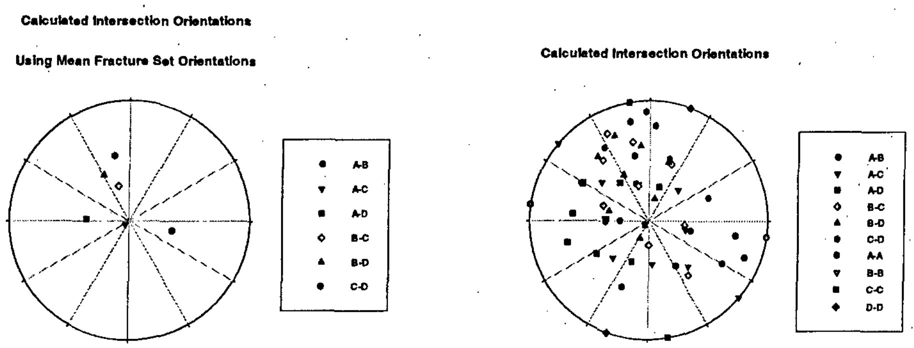
15 meter Cube Simulations

Figure 4.25



20 meter Cube Simulations

Figure 4.26



Calculated Intersection Orientations

Using Mean Fracture Set Orientations

Calculated Intersection Orientations

Figure 4.27 and Figure 4.28

References

Asmar, N., Partial Differential Equations and Boundary Value Problems, Prentice-Hall, 2000.

Bangoy, L.M., P. Bidaux, C. Drogue, R. Plegat and S. Pistre, 1992. A New Method of Characterizing Fissured Media by Pumping Tests with Observation Wells. Journal of Hydrology, 138(1992):77-88.

Barker, J.A., 1988. A Generalized Radial Flow Model for Hydraulic Tests in Fractured Rock. Water Resources Research, 24(10):1796-1804.

Billings, M.P., Structural Geology, Prentice Hall, 1954.

Billings, M.P., 1976. Geology of the Boston Basin. Geological Society of America, Memoir 146, 5-30.

Dershowitz, W.S., Rock Joint Systems. PhD Thesis, Massachusetts Institute of Technology, 1984.

Dershowitz, W.S., and T. Doe, 1997. Analysis of Heterogeneously Connected Rock Masses by Forward Modeling of Fractional Dimension Flow Behavior. International Journal of Rock Mechanics and Mining Sciences, 34(3-4):607-613

Dershowitz, W.S. and C. Fidelibus, 1999. Derivation of Equivalent Pipe Network Analogues for Three-Dimensional Discrete Fracture Networks by the Boundary Element Method. Water Resources Research, 35(9):2685-2691.

Dershowitz, W.S., T. Foxford, and T. Doe. Research Report on Fracture Data Analysis Technology. Golder Associates Inc., March 9, 1998.

Dershowitz, W.S., H.H. Herda, 1992. Interpretation of Fracture Spacing and Intensity. Proceedings of the 33rd US Symposium on Rock Mechanics, Balkema, Rotterdam. 757-766.

Doe, T.W., 1991. Fractional Dimension Analysis of Constant-Pressure Well Tests, SPE Paper #22702, Society of Petroleum Engineers Annual Meeting, Dallas, TX.

Doe, T.W. and C. Chakrabarty, 1997. Analysis of Well Tests in Two-Zone Composite Systems with Different Spatial Dimensions. Water Resources Research (in press).

Doe, T.W. and P. Wallman, 1995. Hydraulic Characterization of Fracture Geometry for Discrete Fracture Modeling. Proceedings, 8th International Congress on Rock Mechanics. Tokyo, Japan, pp. 767-772

Earlougher, R.C. Jr., 1977. Advances in Well-Test Analysis. Monograph Volume 5. Society of Petroleum Engineers, American Institute of Mining, Metallurgical and Petroleum Engineers. Dallas, Texas.

Ehlig-Economides, C.A., Well Test Analysis for Wells Produced at Constant Pressure. PhD Thesis, Stanford University, 1979.

Freeze, R.A. and J.A. Cherry, Groundwater, Prentice Hall, 1979.

Hatcher, R. D., Structural Geology, Principles, Concepts and Problems, Prentice Hall, 1995.

Ivanova, V.M., Geologic and stochastic modeling of fracture systems in rocks. PhD Thesis, Massachusetts Institute of Technology, 1998.

LaForge, L., 1932. Geology of the Boston Area, Massachusetts. US Geological Survey Bulletin, No. 839.

Leveinen, J., 2000. Composite Model with Fractional Flow Dimensions for Well Test Analysis in Fractured Rocks. Journal of Hydrology, 234(2000):116-141.

Meyer, T., Geologic stochastic modeling of rock fracture systems related to crustal faults. Master of Science Thesis, Massachusetts Institute of Technology, 1999.

Novakowski, K.S., 1989. A Composite Analytical Model for Analysis of Pumping Tests Affected by Well Bore Storage and Finite Skin Thickness. Water Resources Research, 25(9):1937-1946.

Rock Fractures and Fluid Flow: Contemporary Understanding and Applications. Committee on Fracture Characterization and Fluid Flow ... [et. al.]. National Academy Press, 1996.

Stehfest, H., Algorithm 368 – Numerical Inversion of Laplace Transforms, Communications of the ACM, 13(1):47-49, January 1970.

Woodhouse, D. *et al.*, 1991. Geology of Boston, Massachusetts, United States of America. Bulletin of the Association of Engineering Geologists, 28(4):375-512.

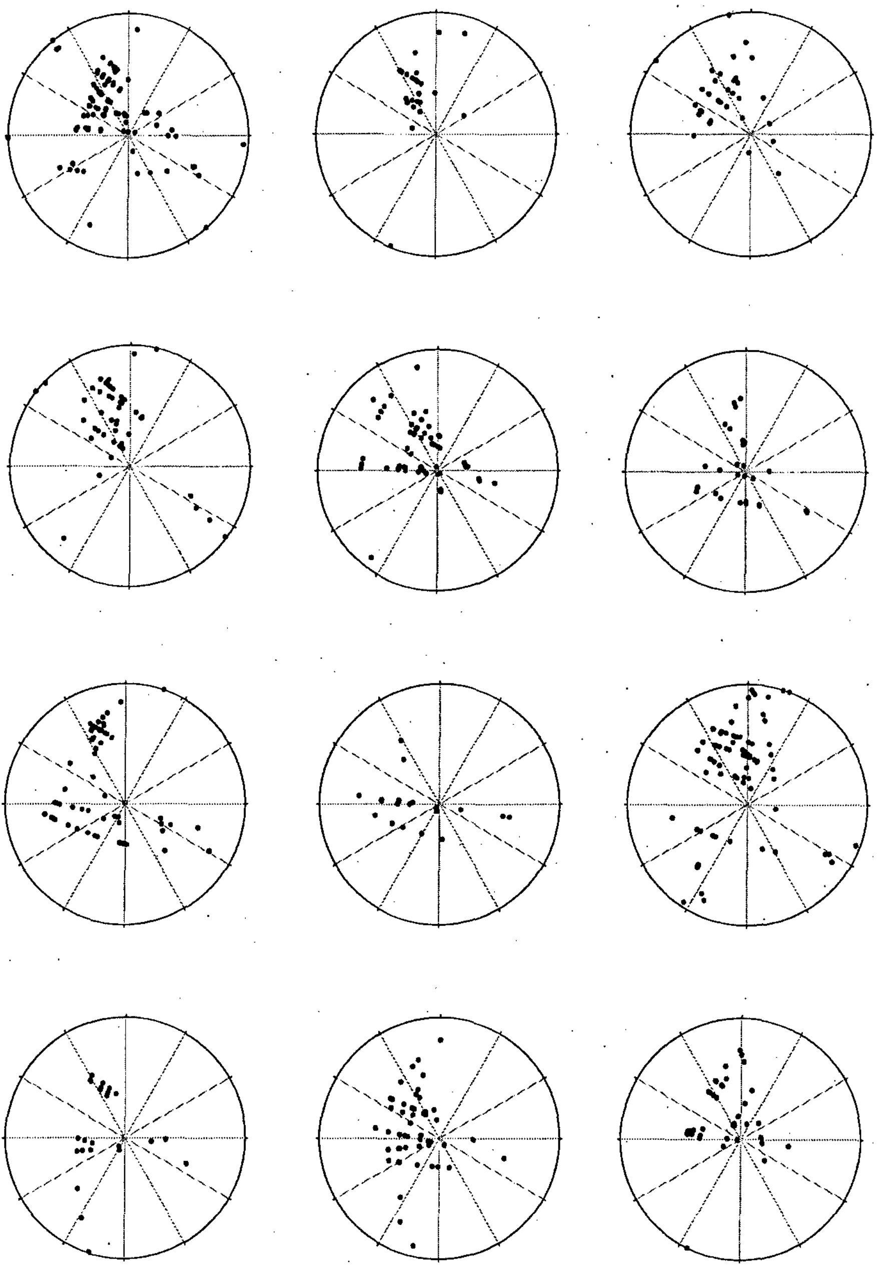
Zhang, L. and H.H. Einstein, 1998. Estimating the Mean Tracelength of Rock Discontinuities. Rock Mechanics and Rock Engineering, 34(4).

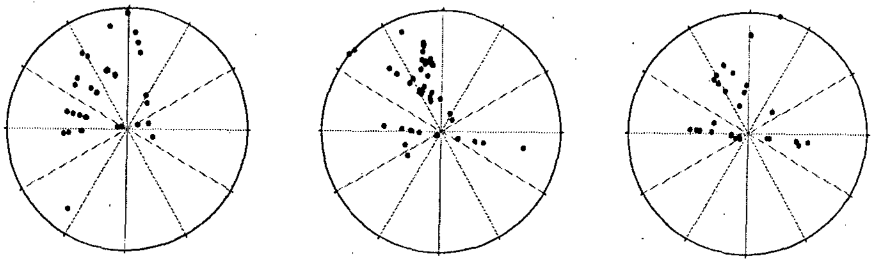
Zhang, L., Analysis and Design of Drilled Shafts in Rock. PhD Thesis, Massachusetts Institute of Technology, 1999.

Appendix

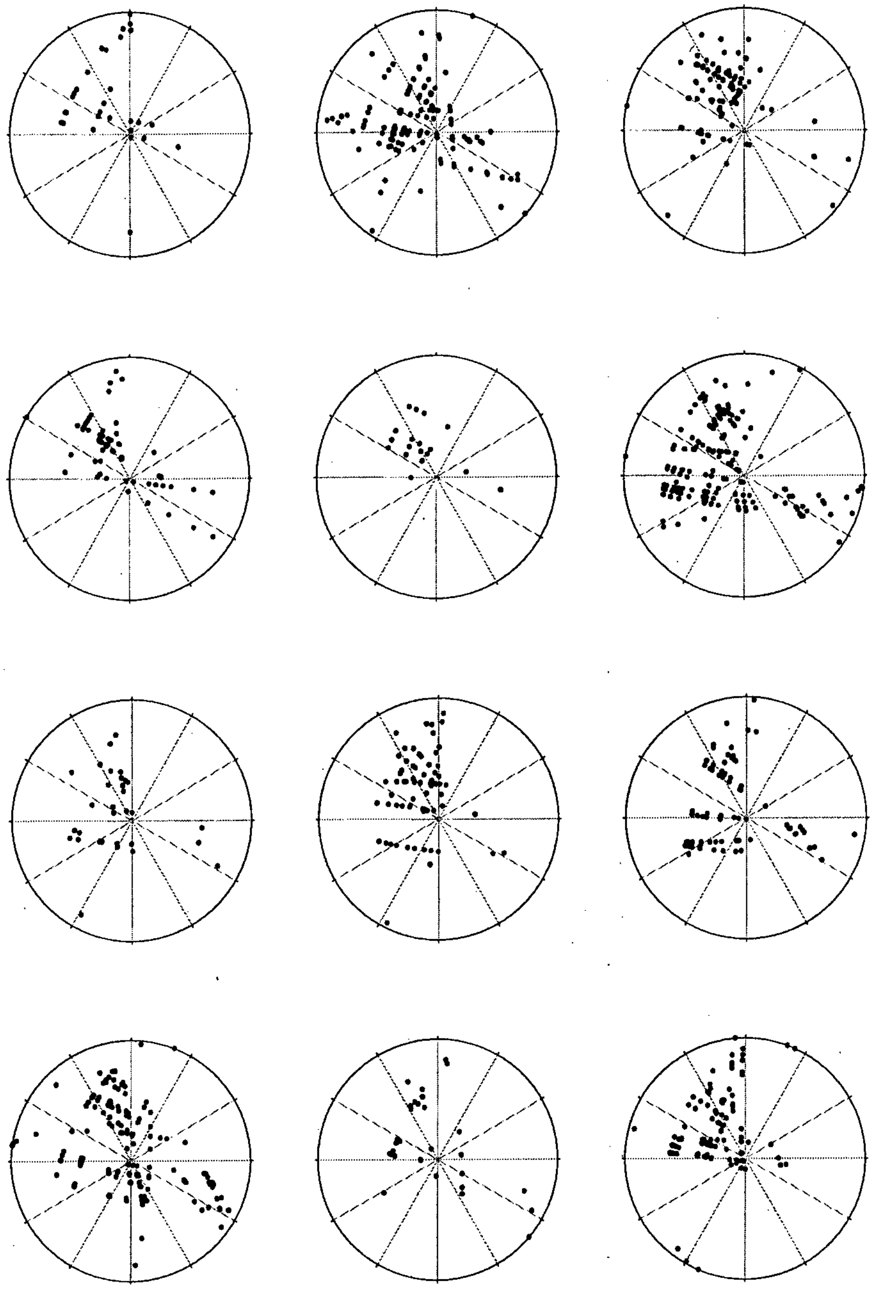
Intersection Trend and Plunge Data

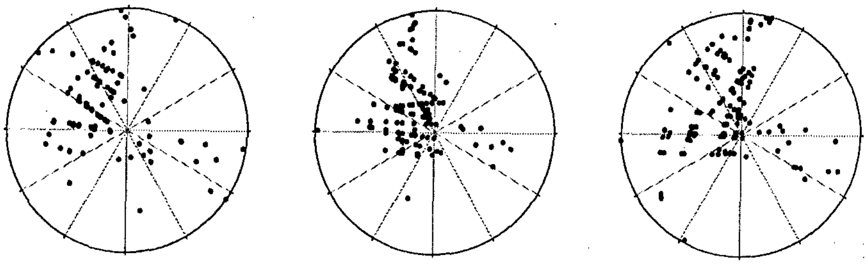
10^3 m^3 Cube Simulations



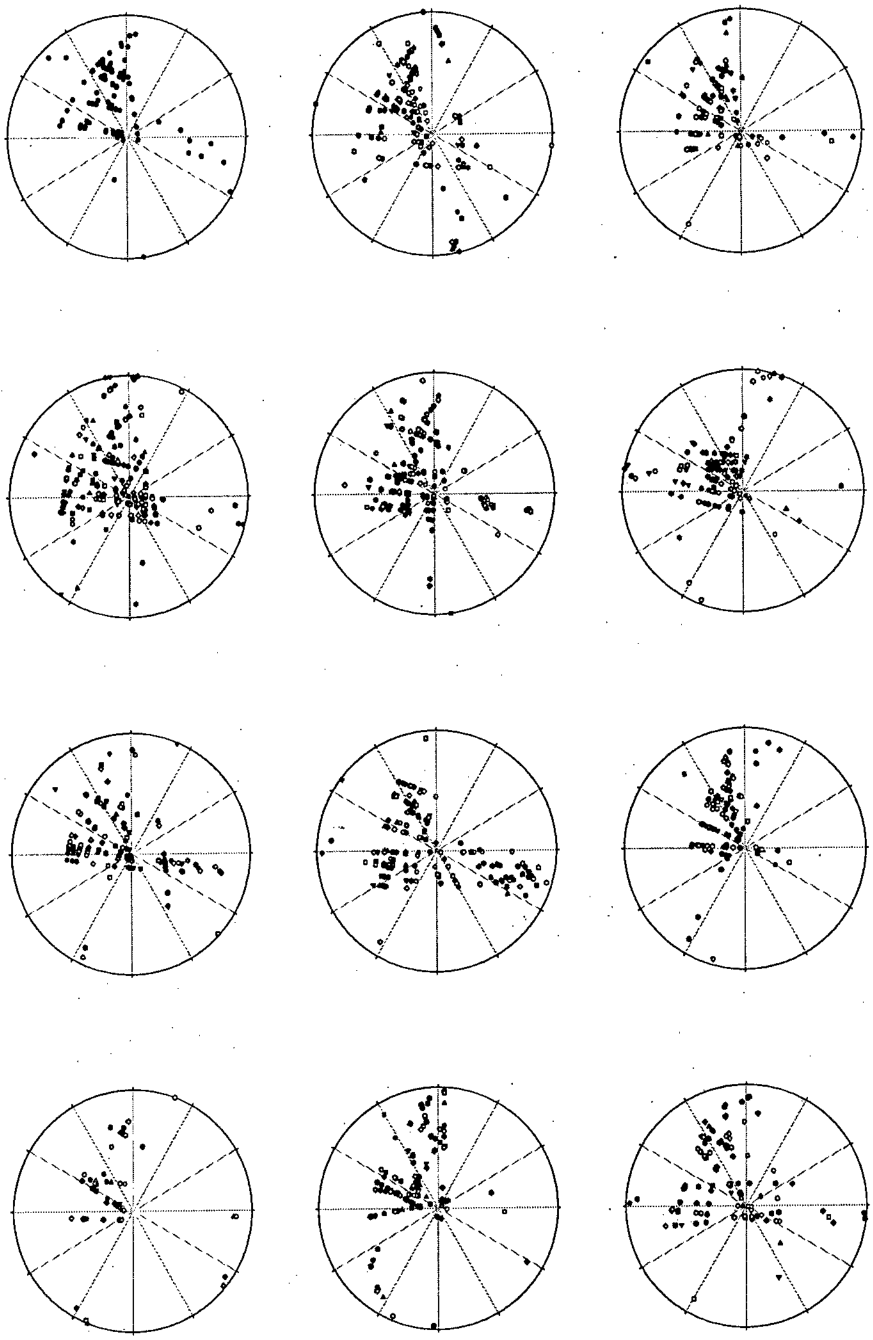


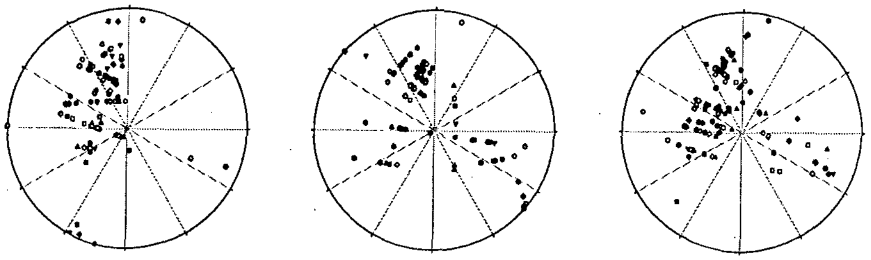
12^3 m^3 Cube Simulations



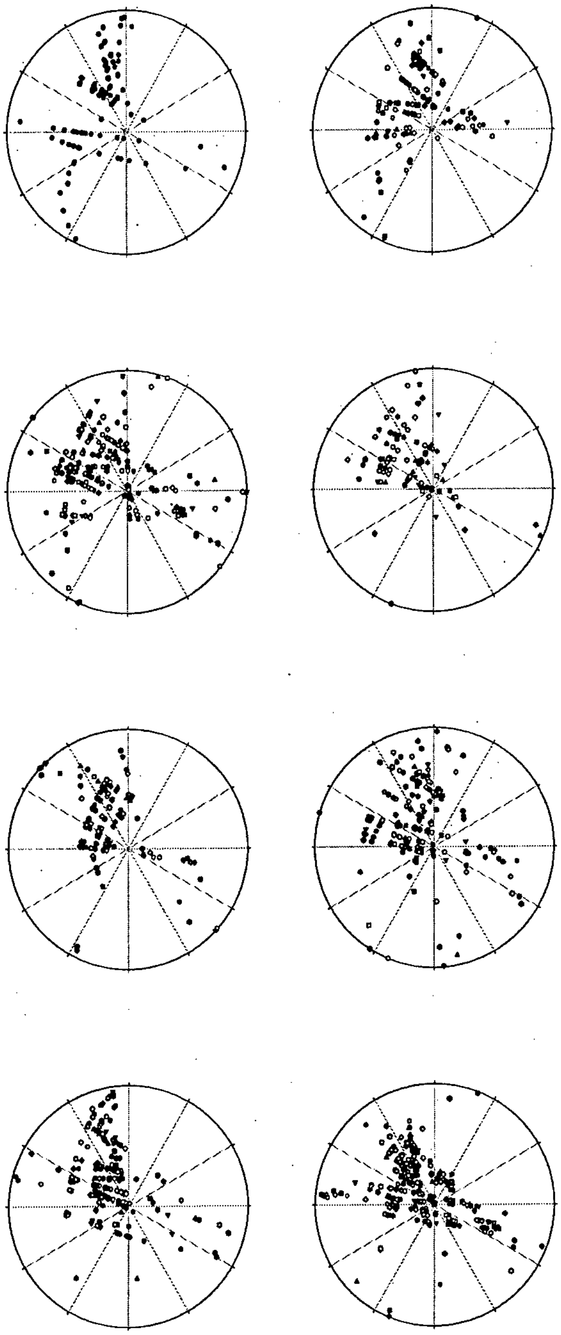


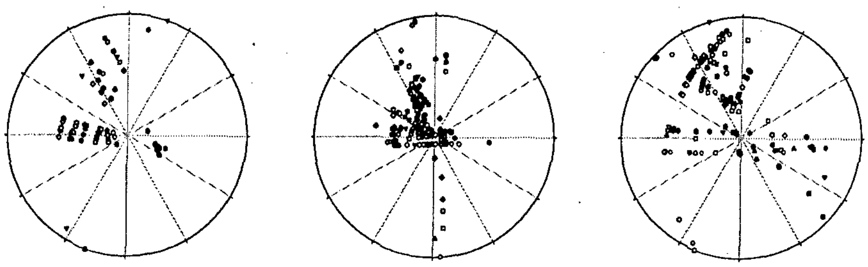
14^3 m^3 Cube Simulations



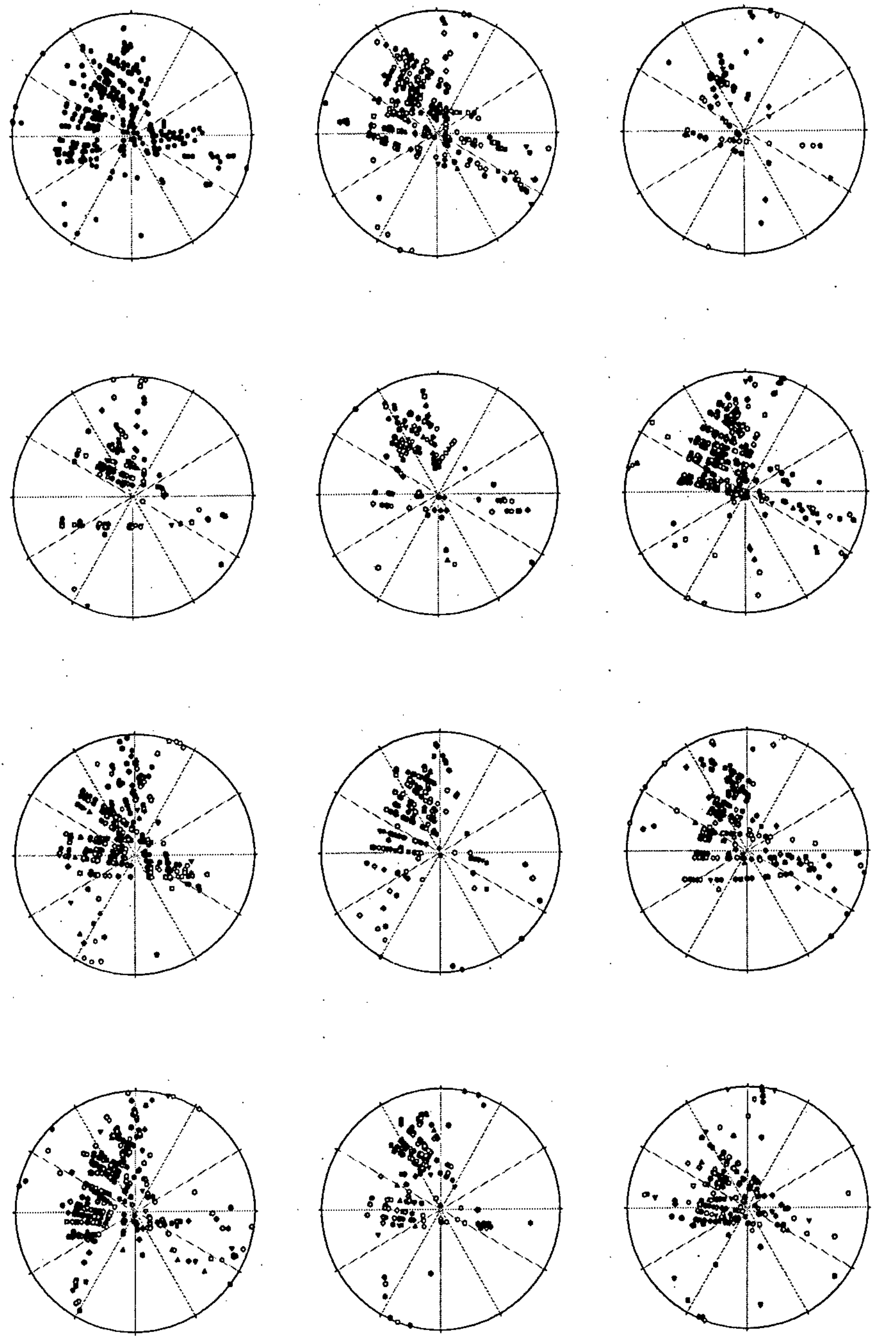


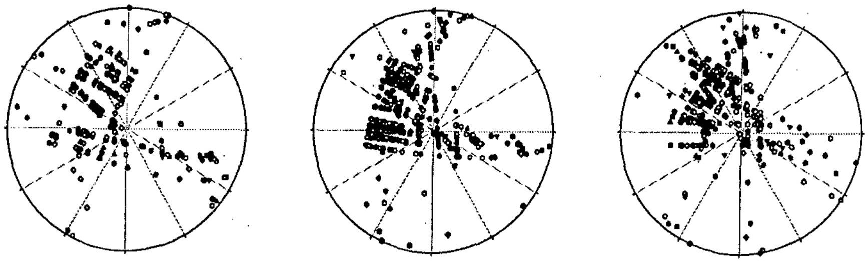
15^3 m^3 Cube Simulations





20^3 m^3 Cube Simulations





GEOFRAC Source Code

Header Files

borehole.h

```
// *******************************************************************
// *                          G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                     *
// *******************************************************************

#ifndef _BOREHOLE_H
#define _BOREHOLE_H
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "line.h"
#include "point.h"
#include "stat.h"
#include "polygon.h"
extern ofstream out;

class Node_frac
{
 public:

double X, Y, Z;
double t;                 // from parametric equation of a line
                         // for a line segment: 0<= t <=length
Polygon* fracture;
Node_frac* next_frac;

friend ostream& operator<< (ostream& o, Node_frac& nf)
{
o<<nf.X<<" "<<nf.Y<<" "<<nf.Z<<" "<<nf.t<<" "<<nf.fracture-
>Pole.theta*180./PI-90.<<" "<<nf.fracture->Pole.theta*180./PI<<"
"<<nf.fracture->dip*180./PI<<" "<<*nf.fracture;
};

};



class Borehole
{
 public:
Line log;
Node_frac* head_frac;
int Nfrac;
Borehole (Line& L)                                    // default constructor
  {
log = L;
head_frac = 0;
Nfrac = 0;
};

void if_exists_add_intersection (Polygon& );
friend ostream& operator<< (ostream& , Borehole&);
void print();
Stat find_mean_sd_spacing ();

~Borehole () {};                                    // default destructor

};

#endif _BOREHOLE_H
```

box.h

```
// ********************************************************************
// *                          G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                  (contact einstein@mit.edu)                      *
// ********************************************************************

#ifndef _BOX_H
#define _BOX_H
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include "polar.h"
#include "cartesian.h"
#include "point.h"
#include "plane.h"
#include "polygon.h"

#define TRUE 1
#define FALSE 0
#define boolian int

// Class Box is used to define a subvolume of the modelling volume where the
// actual parameters of the fracture network (intensity, ...) can be computed.
// Eight points define the corners of the box. Corner[0] to Corner[3] define
// the top of the box and have to be input anti-clockwise. Corner[4] to
// Corner[7] define to bottom of the box and have to be input anti-clockwise
// Corner[0] has to be connected to Corner[4], Corner[1] to Corner[5], aso..

class Box
{
public:
 Point Corner[8];
 double volume;
 Plane Pl_045;
 Plane Pl_051;
 Plane Pl_156;
 Plane Pl_162;
 Plane Pl_632;
 Plane Pl_673;
 Plane Pl_703;
 Plane Pl_740;
 Plane Pl_547;
 Plane Pl_576;
 Plane Pl_130;
 Plane Pl_123;


Box (void); //Default constructor
Box (Point& P0, Point& P1, Point& P2, Point& P3, Point& P4, Point& P5, Point&
P6, Point& P7);    // Constructor with 8 corners

~Box () {};           //Default destructor

int is_point_inside(Point& P);
boolian if_box_mark_polygon(Polygon&, double*);

};


#endif _BOX_H
```

cartesian.h

```
// *******************************************************************
// *                             G E O F R A C                       *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _CARTESIAN_H
#define _CARTESIAN_H

#include <math.h>
#include <iostream.h>
#include <fstream.h>
#define UNIT_VECTOR Cartesian;
#define VECTOR Cartesian
#define FALSE 0
#define TRUE 1
#define boolian int
#include "polar.h"



class Cartesian
{
 public:
    double X;
    double Y;
    double Z;
    Cartesian (void)
        {X=0.0;
         Y=0.0;
         Z=0.0;
        };
    Cartesian (double x,double y,double z)
        {X = x;
         Y = y;
         Z = z;
        };

    friend class Polar;
   Cartesian (Polar& pole)
    {  X=sin(pole.phi)*sin(pole.theta);
       Y=sin(pole.phi)*cos(pole.theta);
       Z=cos(pole.phi);
    };

   // NEWLY ADDED MEMBER!!  AUG. 17, 2000
   Cartesian& operator/ (Cartesian& c)
     {
       Cartesian *temp=new Cartesian;
       *temp=*this;
```

```
          temp->X/=c.X;
          temp->Y/=c.Y;
          temp->Z/=c.Z;
          return (*temp);
        };

    // NEWLY ADDED MEMBER!! OCT. 8, 2000
    // Overload the operator/=
    Cartesian& operator/=(const Cartesian &v)
    {
      this->X/=v.X;
      this->Y/=v.Y;
      this->Z/=v.Z;
      return (*this);
    };

    Cartesian& operator = (Cartesian  c)
         {X = c.X;
          Y = c.Y;
          Z = c.Z;
          return (*this);
         };
    boolian operator== (Cartesian& c)
      {
        if (X==c.X && Y==c.Y && Z==c.Z)
            return (TRUE);
            return (FALSE);
      };

    double operator* (Cartesian& c)       // Dot product
      {double product = X*c.X+Y*c.Y+Z*c.Z;
       return product;
      };

      Cartesian cross (Cartesian& c)      // Cross product: LEFT handed system
      {double xx = Y*c.Z - Z*c.Y;
       double yy = Z*c.X - X*c.Z;
       double zz = X*c.Y - Y*c.X;
       Cartesian* C = new Cartesian (xx, yy, zz);
       return *C;
      };

   friend ostream& operator<< (ostream& o, Cartesian& c)
    {
         return o <<c.X<<" "<<c.Y<<" "<<c.Z<<endl;
    };

  Polar convert_to_polar ();
  Cartesian local_coordinates (Polar&);
  Cartesian global_coordinates (Polar&);

 };


#endif _CARTESIAN_H
```

cell.h

```
// *******************************************************************
// *                            G E O F R A C                        *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _CELL_H
#define _CELL_H
#include <iostream.h>
#include <fstream.h>
#include <math.h>



class Cell
{
 public:

Cell* next_cell;
double Z;
double value;
double GR;

Cell (double z, double V) {
Z = z;
value = V;
next_cell = 0; };

Cell (double z, double V, double gamaray) {
Z = z;
value = V;
GR = gamaray;
next_cell = 0; };

};



class Column
{
 public:
double X, Y;
Cell* head_cell;
int Ncells;

Column ()   // default constructor
  {
head_cell = 0;
Ncells = 0;
};
void add_cell (Cell);
friend ostream& operator<< (ostream& , Column&);


};

#endif _CELL_H
```

circle.h

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998   *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _CIRCLE_H
#define _CIRCLE_H
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include "polar.h"
#include "cartesian.h"
#include "point.h"
#include "plane.h"

#define TRUE 1
#define FALSE 0
#define boolian int

// Class Circle defines a circle on a plane. It is primarily used as a window
for
// fracture traces sampling in order to infer the fracture size distribution.
// An object circle is defined by its center (Point), its radius (double)
// and the supporting plane (Plane).

class Circle
{
public:
 double radius;
 Point center;
 Plane support;


Circle (void); //Default constructor

Circle (double& R, Point& C, Plane& Pl);   // Constructor with radius, center
and
                                           // supporting plane

~Circle () {};         //Default destructor

int is_point_inside(Point& P);

};


#endif _CIRCLE_H
```

cubic.h

```
// *********************************************************************
// *                              G E O F R A C                        *
// *    Copyright Massachusetts Institute of Technology 1995-1998      *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein           *
// *                                                                   *
// *          Don't use or modify without written permission           *
// *                    (contact einstein@mit.edu)                     *
// *********************************************************************

#ifndef _CUBIC_H
#define _CUBIC_H
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include "polar.h"
#include "cartesian.h"
#include "point.h"

#define TRUE 1
#define FALSE 0
#define boolian int


// Class of cubic surface is defined by equation
// z = Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2 + Fxy + Gy^2 + Hx + Iy +J;
// This class is used for fold surfaces.

class Cubic
{
public:
double A, B, C, D, E, F, G, H, I, J;

Cubic()                          //Default constructor
  { A=0.; B=0.; C=0.; D=0.; E=0.; F=0.; G=0.; H=0.; I=0.; J=0.;};

Cubic (double a, double b, double c, double d, double e, double f, double g,
double h, double i, double j)
  { A=a; B=b; C=c; D=d; E=e; F=f; G=g; H=h; I=i; J=j;};

friend ostream& operator<< (ostream& o , Cubic& c)
 {
  o <<c.A<<" "<<c.B<<" "<<c.C<<" "<<c.D<<" "<<c.E<<" "<<c.F<<" "<<c.G<<"
"<<c.H<<" "<<c.I<<" "<<c.J<<endl;
  return o;
};

Cubic& operator= (Cubic c)
 {
  A=c.A; B=c.B; C=c.C; D=c.D; E=c.E; F=c.F; G=c.G; H=c.H; I=c.I; J=c.J;
  return (*this);
};
~Cubic () {};                  //Default destructor

Cartesian normal_at_point(Point&);
Polar polar_normal_at_point (Point&);
Polar strike_dip_at_point (Point&);

};


#endif _CUBIC_H
```

line.h

```
// ******************************************************************
// *                          G E O F R A C                         *
// *    Copyright Massachusetts Institute of Technology 1995-1998   *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein        *
// *                                                                *
// *          Don't use or modify without written permission        *
// *                   (contact einstein@mit.edu)                   *
// ******************************************************************

#ifndef _LINE_H
#define _LINE_H

#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "cartesian.h"
#include "point.h"
#include "plane.h"

#define TRUE 1
#define FALSE 0


// Class Line represents a LINE SEGMENT between 2 3D points P1(x1, y1, z1)
// and P2(x2, y2, z2). The line vector is (x2-x1, y2-y1, z2-z1).
// Parametric equation is t=(X-x1)/(x2-x1)=(Y-y1)/(y2-y1)=(Z-z1)/(z2-z1)
// for point P(X,Y,Z) on the line.
// If t=[0,1] P is on the segment P1P2 (for t=0, P=P1; for t=1 P=P2).
// If t=(-infinity, +infinity), point (X, Y, Z) is on the infinite 3D line.

class Line
{
 public:
  Point end1, end2;
  Cartesian vector;
  double length;

Line (void)                      //default constructor
  {
     end1.X=0.0;                 //line segment from (0,0,0) to (1,1,1)
     end1.Y=0.0;
     end1.Z=0.0;
     end2.X=1.0;
     end2.Y=1.0;
     end2.Z=1.0;
     vector.X =1.0; vector.Y =1.0; vector.Z =1.0;
     length = 0.;
   };

Line (Point& p1, Point& p2)             //constructor from 2 points;
   {
     end1 = p1;
     end2 = p2;
     vector.X = end2.X - end1.X;
     vector.Y = end2.Y - end1.Y;
     vector.Z = end2.Z - end1.Z;
     length = sqrt(vector.X*vector.X+vector.Y*vector.Y+vector.Z*vector.Z);
     vector.X/=length; vector.Y/=length; vector.Z/=length;
   };

Line (Point& p, Cartesian& c, double l)
  {  end1.X=p.X;                    // constructor for line trough point P
     end1.Y=p.Y;                    // parallel to a vector c and with length l
     end1.Z=p.Z;
     double cl = sqrt (c.X*c.X + c.Y*c.Y + c.Z*c.Z);
     if (c.X || c.Y || c.Z)
       {vector.X=c.X/cl; vector.Y=c.Y/cl; vector.Z=c.Z/cl;};
     end2.X=end1.X+l*vector.X;
     end2.Y=end1.Y+l*vector.Y;
     end2.Z=end1.Z+l*vector.Z;
     length = l;
   };

Line (double angle, double D, double R, Point& C);
                         // Constructor for a 2D line with equation
                         // X*cos(angle) + Y*sin(angle) = D intersecting
                         // the  circle with radius R and center C


~Line () {};                                   //destructor

boolian intersect (Line& );   //if a 2D line intersects another 2D line
Point intersection_with_line(Line& );
boolian is_point_on_line (Point&);
double angle_with_line (Line& L)      // angle between two lines in 3D
  {double angle = acos (vector*L.vector);
  return angle;};
boolian if_intersects_plane (Plane&);
Point intersection_with_plane (Plane&);
void global_coordinates (Polar&);
void local_coordinates (Polar&);
double shortest_dist_to_point(Point&);
Point intersection_from_point(Point&);

//NEWLY ADDED MEMBER JUN 13, 2001
// calculates the plunge of the line
 double compute_plunge();

//NEWLY ADDED MEMBER JUN 13, 2001
// calculates the trend of the line
 double compute_trend();

// NEWLY ADDED MEMBER. AUG 18, 2000
/*bool is_point_an_endpoint_of_line (Point&);*/
// NEWLY ADDED MEMBER. AUG 18, 2000
Point& midpt_of_the_line ();

friend ostream& operator<< (ostream& o , Line& l)
 {
  o <<l.end1.X<<" "<<l.end1.Y<<" "<<l.end1.Z<<endl;
```

```
    o <<l.end2.X<<" "<<l.end2.Y<<" "<<l.end2.Z<<endl;
    o <<"vector "<<l.vector<<" length "<<l.length<<endl;
    return o;
  };

  void print();

  boolian operator==(Line& l)
    {
       if ((end1 == l.end1 && end2==l.end2)||(end2==l.end1 && end1==l.end2))
            return (TRUE);
            return (FALSE);
    };

  // NEWLY ADDED OPERATOR OVERLOAD. AUG 18, 2000
  /*Line& operator= (Line& l)
    {
    end1=l.end1;
    end2=l.end2;
    vector=l.vector;
    length=l.length;
    return (*this);
    };*/

  };


  #endif _LINE_H
```

listline.h

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                    (contact einstein@mit.edu)                   *
// *******************************************************************

#ifndef _LISTLINE_H
#define _LISTLINE_H
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "polygon.h"
#include "line.h"
#include "stat.h"
#include "circle.h"


class Node_line
{
public:
Node_line* next_line;
Line* content;
};


class ListLines
{
 public:
Node_line* head_line;
int Nline;

ListLines ()             // default constructor
  {
  head_line=0;
  Nline = 0;
};


~ListLines () {};      //default destructor

void add_line  (Line&);
friend ostream& operator<< (ostream& , ListLines&);
void print();

//void add_listline (ListLines& );
Stat find_mean_sd_length ();
int count_traces_0(Circle& );
int count_traces_1(Circle& );
int count_traces_2(Circle& );
double find_mean_length_on_pol(Polygon& );
double find_mean_length_on_circ(Circle& );

//NEWLY ADDED FUNCTION OCT. 8, 2000
//removes line from the list
void remove_line(Line &);

};


#endif _LISTLINE_H
```

listlistpol.h

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _LISTLISTPOL_H
#define _LISTLISTPOL_H
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "polygon.h"
#include "line.h"
#include "listline.h"
#include "listpol.h"
#include "surface.h"
#include "cubic.h"
#include "stat.h"
#include "box.h"
#include "borehole.h"
#include "cell.h"


class Node_list
{
 public:

Node_list* next_list;
ListPolygons* content;
};



class ListListPol
{
 public:
Node_list* head_list;
int Nlist;
ListListPol ()   // default constructor
  {
head_list = 0;
Nlist = 0;
  };


void add_listpol(ListPolygons&);
void name_list();

~ListListPol () {};     // default destructor
};

#endif _LISTLISTPOL_H
```

listpol.h

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                    (contact einstein@mit.edu)                   *
// *******************************************************************

#ifndef _LISTPOL_H
#define _LISTPOL_H
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "polygon.h"
#include "line.h"
#include "listline.h"
#include "surface.h"
#include "cubic.h"
#include "stat.h"
#include "box.h"
#include "borehole.h"
#include "cell.h"


class ListListPol;

class Node
{
 public:

Node* next_pol;
Polygon* content;
};



class ListPolygons
{
 public:
Node* head_pol;
int Npol, name_list;
ListPolygons ()   // default constructor
  {
head_pol = 0;
Npol = 0;
name_list=0;
};


void add_polygon  (Polygon&);
void add_polygon_tail (Polygon&);
void remove_from_list (Polygon&);
friend ostream& operator<< (ostream& , ListPolygons&);
void print();
void center_print();
void intersect_by_line (Line& );
void name_pol();
ListPolygons make_copy();
void make_listpol_2d ();
void make_listpol_3d();
void find_area_radius_2d ();
void mark_good_shape (double, double);
void mark_good_shape_and_P (double, double, double );
double find_mean_area();
double find_SD_area ();
void add_listpol (ListPolygons& );
void fractal_tessellation (double, double);
void fractal_big_and_small (double, double);
void fractal_similar (double, double);
ListLines traces_on_plane (Plane& );
Stat find_mean_sd (double, double);
Stat find_mean_sd ();
Stat find_mean_sd_strike();
Stat find_mean_sd_dip();
void translate_2d (double, double);
void mark_parallel_to_strike (Surface&, double, double);
void mark_orthogonal_to_strike (Surface&, double, double);
void mark_by_strike (Surface&, double, double, double, char, double, double);
void mark_by_strike (Cubic&, double, double, double, char, double, double);
void mark_by_dip (Surface&, double, double, double);
void mark_by_dip (Cubic&, double, double,  double);
void cut_by_surface (Surface&, char);
void cut_by_plane(Plane&, char, double);
void cut_between_two_surfaces (Surface&, Surface&);
void cut_outside_two_surfaces (Surface&, Surface&);
double shortest_distance_from_polygon (Polygon& );
boolian if_zone_mark_polygon (Polygon&, int, double*, double*);
void mark_by_zones (ListPolygons& , int, double* , double*);
void mark_by_box (Box&, double*);
Borehole intersections_with_logline (Line&);
void ListPolygons :: mark_to_surface (Surface& , double , char );
void discard_high_porosity (Column*, int, double, double);
void discard_shale (Column*, int, double, double, double);
void size_distribution (int, int*, double*, double);
ListListPol split_into_networks();
double find_c8(int);

~ListPolygons () {};      // default destructor

//NEWLY ADDED FUNCTION OCT. 8, 2000
// function to create a list of intersection lines
ListLines& make_list_of_intersections();


};

#endif _LISTPOL_H
```

plane.h

```
// ********************************************************************
// *                          G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                    (contact einstein@mit.edu)                    *
// ********************************************************************

#ifndef _PLANE_H
#define _PLANE_H

#include "polar.h"
#include "cartesian.h"
#include "point.h"

#include <math.h>
#include <iostream.h>

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)

class Circle;

// Class Plane is used for the planes of tension and shear which are
// generated according to a spherical PDF.
// A plane in 3D is described by the equation Ax+By+Cz=D. This plane is
// perpendicular to vector (A, B, C) and passes through point (X,Y,Z)
// where A*X+B*Y+C*Z=D.


class Plane
{
 public:
    double A, B, C, D, strike, dip;
     Polar MeanPole;
     Polar rel_polar, abs_polar;      // Normal vector in polar coordinates
     Cartesian rel_cart, abs_cart;    // Normal vector in cartesian coord.

     Plane (void)                        // default constructor: xy plane
     {MeanPole.theta=0.; MeanPole.phi=0.;
      rel_polar.theta = 0.0; rel_polar.phi = 0.0;
      abs_polar.theta = 0.; abs_polar.phi = 0.;
      rel_cart.X = 0.0;  rel_cart.Y = 0.0;  rel_cart.Z = 1.0;
      abs_cart.X = 0.0;  abs_cart.Y = 0.0;  abs_cart.Z = 1.0;
      A=0.; B=0.; C=1.; D=0.;
      strike =0.; dip = 0.;
     };

Plane (double, double, double, double);  //constructor from A, B, C, D

Plane (Point&, Point&, Point&);  //constructor from three points P, Q, R
Plane (Cartesian& , Point&, Polar&);
                                        // abs_cart is given by vector
                                        // N (A,B,C). Constructor from
                               // N and point P (D=A*X+B*Y+C*Z)

Plane (Polar&, Polar&, Point&);    // MeanPole is given by first Polar;
                                // rel_cart is given by second Polar;
                                // Point is in absolute f.o.r.


Plane (char, Polar&, double );      //constructor to create plane according
                                   // to an orientation distribution (char)

Cartesian find_binormal();             // Calculates a vector which is
                                    // perpendicular both to the strike and
                                     // dip of the plane and points upward.

int is_point_front_behind(Point&);
Circle circle_on_horizontal();
Circle circle_on_vertical();

~Plane (void) {};                //Default destructor

friend ostream& operator<< (ostream&, Plane&);

Plane& operator= (Plane p)
  {
      A = p.A; B = p.B; C = p.C; D = p.D;
       abs_cart = p.abs_cart;  abs_polar = p.abs_polar;
       rel_cart = p.rel_cart;  rel_polar = p.rel_polar;
       MeanPole = p.MeanPole; strike = p.strike; dip = p.dip;
      return (*this);
  };

};


#endif _PLANE_H
```

point.h

```
// *******************************************************************
// *                          G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998     *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                     *
// *******************************************************************

#ifndef _POINT_H
#define _POINT_H
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include "cartesian.h"

#define TRUE 1
#define FALSE 0
#define boolian int
#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)
extern double Datum;

class Volume;


class Point: public Cartesian
{
public:
Point* next;

Point (void)                  //default constructor
  {
      X=0.0;
      Y=0.0;
      Z=0.0;
      next = 0;
   };

Point (double x, double y, double z)    //constructor for 3d point
   {
      X=x; Y=y; Z=z;
      if (X <=0.000001 && X>=-0.000001)
                X=0.;
      if (Y <=0.000001 && Y>=-0.000001)
              Y=0.;
      if (Z <=0.000001 && Z>=-0.000001)
              Z=0.;
         next = 0;
   };

~Point () {};          // default destructor
//Point (Volume& v);     // Constructor for a random point
                       // in the modeling volumevolume

Point& operator= (Point p)
  {
      X = p.X;
      Y = p.Y;
      Z = p.Z;
      return (*this);
  };

boolian operator==(Point& p)
  {
      if (X == p.X && Y == p.Y && Z == p.Z)
            return (TRUE);
            return (FALSE);
  };

void operator+ (Cartesian& c)       // Addition of a vector: translation
     {X+=c.X; Y+=c.Y; Z+=c.Z;
      return;
     };

void operator- (Cartesian& c)       // Substraction of a vector:
     {X-=c.X; Y-=c.Y; Z-=c.Z;       // back translation
      return;
     };


friend ostream& operator<< (ostream& o , Point& p)
 {
  o <<p.X<<" "<<p.Y<<" "<<p.Z+Datum<<endl;
  return o;
};

void print();
double triple_product(Point&, Point&, Point&);
double give_z();

};

#endif _POINT_H
```

polar.h

```
// *******************************************************************
// *                             G E O F R A C                       *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _POLAR_H
#define _POLAR_H

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)

#include <iostream.h>
#include <fstream.h>
#define FALSE 0
#define TRUE 1
#define boolian int

class Cartesian;


class Polar
{
  public:
     double theta, phi;
     Polar (void)                        // Default constructor
        {theta=0.0;
         phi=0.0;
        };
     Polar (double angle1, double angle2)   // Constructor from two angles
        {theta=angle1;
         phi=angle2;
        };

boolian operator== (Polar& p)
  {
   if (theta == p.theta && phi == p.phi)
       return TRUE;
       return FALSE;
  };


friend ostream& operator<< (ostream& o ,Polar& p)
  { return o << p.theta <<" "<< p.phi<<endl;};

Polar& operator= (Polar p)
        {
         theta = p.theta;
         phi = p.phi;
```

```
          return (*this);
         };
~Polar (void) {};                // Default destructor
Cartesian convert_to_cartesian ();

};


#endif _POLAR_H
```

polygon.h

```
// *******************************************************************
// *                            G E O F R A C                        *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _POLYGON_H
#define _POLYGON_H

#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "point.h"
#include "line.h"
#include "plane.h"
#include "volume.h"
#include "surface.h"
#include "cell.h"
/*#include "initial.C"*/

#define boolian int
#define TRUE 1
#define FALSE 0


class Polygon
{
  public :
      Polar setPole;               // in global f.o.r
      Polar Pole;                  // in global f.o.r.
      double strike, dip;
      Point* head;
      int noP, name, name_list;
      Point center;
      double radius, area;

      Polygon () {                              // Default constructor
       setPole.phi =0.; setPole.theta=0.;
       Pole.theta=0.; Pole.phi=0.;
       head=0; noP=0; name=0; name_list=0;
       strike =0.; dip =0.;
       center.X = 0.; center.Y = 0.; center.Z =0.;
       radius =0.; area = 0.;};

Polygon (Plane&);           // Constructor for polygon parallel to
                            // a plane

friend ostream& operator<< (ostream& o , Polygon& pol)
 {
  Point* marker = pol.head; int i;
  for (i=0; i < pol.noP; ++i)
    {o << (*marker);
    marker = marker -> next;
    };
//  o<<pol.area<<" "<<pol.radius<<" "<<pol.center;

 return o;
};


void operator+ (Cartesian &);             // Translation
void operator- (Cartesian &);             // Translation back


boolian is_it_member (Point& );
void add_point (Point&);
void find_area_radius_2d ();                          // for polygon in 2d
void area_radius_3d ();                // for 3d polygon
void find_center ();                     // for 3d polygon
double find_max_coord(int);
double find_min_coord(int);
void make_polygon_2d ();
void make_polygon_3d ();
void sort_points_2d ();                  // for polygon in 2d
void print();
void add_points_on_surface (Plane&, Surface& , int, double Zm);
boolian if_good_shape (double, double);
double minR_inscribe ();
int how_line_intersects (Line&);         // in 2d
boolian if_line_intersects(Line& );     // in 2d
Polygon divide_by_line (Line&);          // in 2d
double find_elongation ();
boolian if_intersects_plane(Plane& );
Line intersection_with_plane (Plane& );
void translate_2d (double, double);
boolian mark_parallel_to_strike (double, double, double);
boolian mark_orthogonal_to_strike (double, double, double);
boolian mark_by_dip (double, double, double);
void rotate_by_strike (double);
void rotate_by_dip (double);
int above_or_below_surface (Surface& );
void cut_by_surface (Surface&, char);
void cut_by_plane(Plane&, char);
double distance_from_point (Point& );          // in 3d
double distance_from_polygon (Polygon& );      // in 3d
boolian if_3d_line_intersects (Line& );         // in 3d
Point intersection_with_line (Line&);        // in 3d
boolian if_polygon_intersects (Polygon& );
boolian if_front_of_polygon (Polygon& );
double find_average_porosity (Column&);
double find_average_GR (Column& C);
boolian mark_by_porosity (Column*, int, double, double);
boolian mark_by_porosity_and_GR (Column*, int, double, double, double);
int find_closest_column (Column* , int);
void include_in_size_pdf (int, int*, double*, double);
void create_test_polygon();
```

```
// NEWLY ADDED FUNCTION. AUG 18, 2000
Line& line_of_intersection_with_polygon(Polygon&);

~Polygon () {};

};

#endif _POLYGON_H
```

stat.h

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein        *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _STAT_H
#define _STAT_H
#include <iostream.h>
#include <fstream.h>
#include <math.h>


class Stat {

public:
  double Mean;
  double SD;
  double total;
Stat ()
  { Mean = 0.; SD =0.; total = 0.;};

Stat (double M, double sd, double t)
  {
Mean = M;
SD = sd;
total = t;
};



friend ostream& operator<< (ostream& o , Stat s)
 {
  o <<s.Mean<<" "<<s.SD<<" "<<s.total;
  return o;
};

};


#endif _STAT_H
```

surface.h

```
// *******************************************************************
// *                           G E O F R A C                         *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#ifndef _SURFACE_H
#define _SURFACE_H
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include "polar.h"
#include "cartesian.h"
#include "point.h"
#include "line.h"

#define TRUE 1
#define FALSE 0
#define boolian int

class Line;

// Class of surface is defined by quadratic equation
// z = Ax^2 + Bxy + Cy^2 + Dx + Ey + F
// This class is used for topographic surface and for internal
// surfaces for example shale layers

class Surface
{
public:
double A, B, C, D, E, F;

Surface()                        //Default constructor
  { A=0.; B=0.; C=0.; D=0.; E=0.; F=0.;};

Surface (double a, double b, double c, double d, double e, double f)
  { A=a; B=b; C=c; D=d; E=e; F=f;};

friend ostream& operator<< (ostream& o , Surface& s)
 {
  o <<s.A<<" "<<s.B<<" "<<s.C<<" "<<s.D<<" "<<s.E<<" "<<s.F<<endl;
  return o;
};

Surface& operator= (Surface s)
 {
  A=s.A; B=s.B; C=s.C; D=s.D; E=s.E; F=s.F;
  return (*this);
};
~Surface () {};                  //Default destructor

boolian is_point_on_surface(Point&);
boolian is_line_on_surface (Line&);
Cartesian normal_at_point(Point&);
Polar polar_normal_at_point (Point&);
double find_Z_on_surface (double, double);
Polar strike_dip_at_point (Point&);
double find_enclosed_volume (double, double);
double Zmax_over_XY(double, double);
int is_point_above_below (Point&);
int how_line_intersect (Line& L);
Point intersection_by_line (Line&);

// More functions to be created : for tangent plane, etc

};


#endif _SURFACE_H
```

volume.h

```
// *******************************************************************
// *                          G E O F R A C                           *
// *   Copyright Massachusetts Institute of Technology 1995-1998      *
// *        Violeta Ivanova, Thomas Meyer, Herbert Einstein           *
// *                                                                  *
// *        Don't use or modify without written permission            *
// *                  (contact einstein@mit.edu)                      *
// *******************************************************************

#ifndef _VOLUME_H
#define _VOLUME_H
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include "polar.h"
#include "cartesian.h"
#include "point.h"
#include "surface.h"

#define TRUE 1
#define FALSE 0
#define boolian int


// Class Volume is used for calculation of the modeling volume for fracture
// sets enclosed between the topographic surface and five planes (horizontal
// and four vertical). The horizontal bottom plane  at z=0; The
// vertical planes are at x=Xm, x=-Xm, y=Ym, and y=-Ym. The origin (0,0,0)
// of the coordinate system is set to to the center of the bottom plane.
// The four points P[i] are the points in the corners of the volume on
// the top surface.

class Surface;

class Volume
{
public:
 Point P[4];
 double volume;
 Surface top;
 double Zmax;

Volume (Surface& ground);   // Constructor

~Volume () {};          //Default destructor



friend ostream& operator<< (ostream& o, Volume& v)
   {
int i;
for (i=0; i<4; ++i)
    o << v.P[i];
    o <<"volume: "<<v.volume<<" Zmax: "<<v.Zmax<<endl;
    o <<"top surface: "<<v.top<<endl;
  };

Point random_point();
boolian is_point_inside(Point&);
double corner_min_z();

};


#endif _VOLUME_H
```

C Files

borehole.C

```
// ********************************************************************
// *                          G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998     *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                     *
// ********************************************************************

#include "borehole.h"
#include "listpol.h"
#include "polygon.h"
#include "stat.h"
#include <math.h>
#include <iostream.h>

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)
extern ofstream out;

ostream& operator<< (ostream& o, Borehole& bh)
{

  o<<"Logline "<<endl<<bh.log<<"N intersections: "<<bh.Nfrac<<endl<<endl;

int i;
Node_frac* current = bh.head_frac;
o<<"Intersection with borehole "<<endl<<"Number-X-Y-Z-t-STRIKE-AZIMUTH-DIP-
AREA-RADIUS-Xc-Yc-Zc"<<endl<<endl;

for (i=0; i<bh.Nfrac; ++i)
  {o<<i+1<<" "<<*current;
 current = current->next_frac;};

o<<endl;
return o;

};


// Procedure to add the intersection of a log line with a fracture (Polygon)
// to the list of intersections in a Borehole. the function first checks
// if the line and the polygon intersect at all.


void Borehole :: if_exists_add_intersection (Polygon& pol)
{
if (pol.if_3d_line_intersects (log))
  {
Point P = pol.intersection_with_line (log);
Node_frac* new_nf = new Node_frac;
new_nf->X = P.X; new_nf->Y = P.Y; new_nf->Z = P.Z;
new_nf->fracture = &pol;


if (log.vector.Z)
    new_nf->t = (new_nf->Z - log.end1.Z) / log.vector.Z;
else if (log.vector.Y)
    new_nf->t = (new_nf->Y - log.end1.Y) / log.vector.Y;
else
    new_nf->t = (new_nf->X - log.end1.X) / log.vector.X;

if (!Nfrac)
  {
head_frac = new_nf;
new_nf ->next_frac = 0;
  }

else if (head_frac->t >= new_nf->t)
      {
       new_nf->next_frac = head_frac;
       head_frac = new_nf;}

else
     {

Node_frac* current = head_frac;
while (current->next_frac && (current->next_frac->t < new_nf->t))
     current = current->next_frac;

new_nf->next_frac = current->next_frac;
current->next_frac = new_nf;
       };

Nfrac++;
};

return;

};



// Procedure to find the intersections of a log line with a list of
// polygon-fractures. Returns an ordered log, i.e. the intersections with
// polygons form end1 to end2 of the given line.


Borehole ListPolygons :: intersections_with_logline (Line& L)
{
Node* current = head_pol;
int i = Npol;
Borehole* new_bh = new Borehole (L);

for (i=0; i<Npol; ++i)
   {new_bh -> if_exists_add_intersection (*current->content);
   current = current->next_pol;};
```

```
return *new_bh;

};


// Procedure to find the mean and standard deviation of the spacing
// between the fracture intersections in a borehole.


Stat Borehole :: find_mean_sd_spacing ()
{
Node_frac* current = head_frac;
int i;
double total = log.length;
double meanS=0., sdS=0.;
double spacing;

if (Nfrac>1)
  {

for (i=1; i<Nfrac; ++i)
  {
spacing = current->next_frac->t - current->t;
meanS += spacing;
sdS += spacing*spacing;
current = current->next_frac;
  };

meanS /= (Nfrac -1);

if (Nfrac == 2)
 sdS = 0.;
else
 sdS = sqrt ((sdS - (Nfrac-1)*meanS*meanS) / (Nfrac -2));
  };

Stat spacingMSD (meanS, sdS, total);
return spacingMSD;
};


// Procedure to print the polygons-fractures of a borehole
// for graphical output


void Borehole :: print()
{
int i;
Node_frac* current = head_frac;
if (current->fracture)
  {
for (i=0; i<Nfrac-1; ++i)
  {current->fracture->print();
   out<<",  ";
   current=current->next_frac;};
current->fracture->print();
```

```
    };
  return;
  };
```

box.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "polar.h"
#include "cartesian.h"
#include "point.h"
#include "plane.h"
#include "box.h"

// Default constructor (1x1x1 cube)

Box :: Box (void)
  {
  Corner[0].X=0.; Corner[0].Y=0.; Corner[0].Z=1.;
  Corner[1].X=1.; Corner[1].Y=0.; Corner[1].Z=1.;
  Corner[2].X=1.; Corner[2].Y=1.; Corner[2].Z=1.;
  Corner[3].X=0.; Corner[3].Y=1.; Corner[3].Z=1.;
  Corner[4].X=0.; Corner[4].Y=0.; Corner[4].Z=0.;
  Corner[5].X=1.; Corner[5].Y=0.; Corner[5].Z=0.;
  Corner[6].X=1.; Corner[6].Y=1.; Corner[6].Z=0.;
  Corner[7].X=0.; Corner[7].Y=1.; Corner[7].Z=0.;
  volume=1.;

  Pl_045=Plane(Corner[0], Corner[4], Corner[5]);
  Pl_051=Plane(Corner[0], Corner[5], Corner[1]);
  Pl_156=Plane(Corner[1], Corner[5], Corner[6]);
  Pl_162=Plane(Corner[1], Corner[6], Corner[2]);
  Pl_632=Plane(Corner[6], Corner[3], Corner[2]);
  Pl_673=Plane(Corner[6], Corner[7], Corner[3]);
  Pl_703=Plane(Corner[7], Corner[0], Corner[3]);
  Pl_740=Plane(Corner[7], Corner[4], Corner[0]);
  Pl_547=Plane(Corner[5], Corner[4], Corner[7]);
  Pl_576=Plane(Corner[5], Corner[7], Corner[6]);
  Pl_130=Plane(Corner[1], Corner[3], Corner[0]);
  Pl_123=Plane(Corner[1], Corner[2], Corner[3]);
  };

// Constructor

Box :: Box (Point& P0, Point& P1, Point& P2, Point& P3, Point& P4, Point& P5,
Point& P6, Point& P7)
  {
   Corner[0].X=P0.X; Corner[0].Y=P0.Y; Corner[0].Z=P0.Z;
   Corner[1].X=P1.X; Corner[1].Y=P1.Y; Corner[1].Z=P1.Z;
   Corner[2].X=P2.X; Corner[2].Y=P2.Y; Corner[2].Z=P2.Z;
   Corner[3].X=P3.X; Corner[3].Y=P3.Y; Corner[3].Z=P3.Z;
   Corner[4].X=P4.X; Corner[4].Y=P4.Y; Corner[4].Z=P4.Z;
```

```
    Corner[5].X=P5.X; Corner[5].Y=P5.Y; Corner[5].Z=P5.Z;
    Corner[6].X=P6.X; Corner[6].Y=P6.Y; Corner[6].Z=P6.Z;
    Corner[7].X=P7.X; Corner[7].Y=P7.Y; Corner[7].Z=P7.Z;

    volume=Corner[1].triple_product(Corner[2], Corner[6],
 Corner[3])+Corner[1].triple_product(Corner[6], Corner[7],
 Corner[3])+Corner[1].triple_product(Corner[7], Corner[0],
 Corner[3])+Corner[7].triple_product(Corner[5], Corner[6],
 Corner[1])+Corner[7].triple_product(Corner[0], Corner[5],
 Corner[1])+Corner[7].triple_product(Corner[4], Corner[5], Corner[0]);

   Pl_045=Plane(Corner[0], Corner[4], Corner[5]);
   Pl_051=Plane(Corner[0], Corner[5], Corner[1]);
   Pl_156=Plane(Corner[1], Corner[5], Corner[6]);
   Pl_162=Plane(Corner[1], Corner[6], Corner[2]);
   Pl_632=Plane(Corner[6], Corner[3], Corner[2]);
   Pl_673=Plane(Corner[6], Corner[7], Corner[3]);
   Pl_703=Plane(Corner[7], Corner[0], Corner[3]);
   Pl_740=Plane(Corner[7], Corner[4], Corner[0]);
   Pl_547=Plane(Corner[5], Corner[4], Corner[7]);
   Pl_576=Plane(Corner[5], Corner[7], Corner[6]);
   Pl_130=Plane(Corner[1], Corner[3], Corner[0]);
   Pl_123=Plane(Corner[1], Corner[2], Corner[3]);
   };


// Procedure to test whether or not a point is inside the box. Returns 1 if
// inside or on the surface, 0 if outside.

int Box::is_point_inside(Point& P)
{
int i;
int decide=1;
int test[12];
test[0]=Pl_045.is_point_front_behind(P);
test[1]=Pl_051.is_point_front_behind(P);
test[2]=Pl_156.is_point_front_behind(P);
test[3]=Pl_162.is_point_front_behind(P);
test[4]=Pl_632.is_point_front_behind(P);
test[5]=Pl_673.is_point_front_behind(P);
test[6]=Pl_703.is_point_front_behind(P);
test[7]=Pl_740.is_point_front_behind(P);
test[8]=Pl_547.is_point_front_behind(P);
test[9]=Pl_576.is_point_front_behind(P);
test[10]=Pl_130.is_point_front_behind(P);
test[11]=Pl_123.is_point_front_behind(P);

for(i=0; i<12; ++i)
  {
  if(test[i]>0)
    decide=0;
  };
return decide;
};
```

cartesian.C

```
// ***********************************************************************
// *                           G E O F R A C                             *
// *    Copyright Massachusetts Institute of Technology 1995-1998        *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein             *
// *                                                                     *
// *          Don't use or modify without written permission             *
// *                   (contact einstein@mit.edu)                        *
// ***********************************************************************

#include "cartesian.h"
#include "polar.h"
#include <math.h>


// Procedure to transform the cartesian coordinates of a direction
// into polar. Not to be used for points because polar coordinates
// are only two angles theta and phi, there is no distance.

Polar Cartesian::convert_to_polar ()
        {
double theta, phi;
double l3d = sqrt(X*X+Y*Y+Z*Z);
double l2d = sqrt(X*X+Y*Y);

if (X==0. && Y==0. && Z>=0.)
   theta = phi =0.;
else if (X==0. && Y==0. && Z<0.)
   theta = phi =PI;
else
  {
phi = acos(Z/l3d);
theta = acos(Y/l2d);
if (X<0.)
  theta = -theta;};

Polar* polar_normal = new Polar(theta, phi);
return (*polar_normal);
};


// Function to find the coordinates of a vector into the LOCAL coordinate
// system. The Polar MeanPole (theta, phi) gives the orientation of the
// mean vector in the global plane of reference. The direction of this
// vector is the axis Z of the local coordinate system.

Cartesian Cartesian::local_coordinates (Polar& MP)  {
double ct = cos(MP.theta);
double st = sin(MP.theta);
double cp = cos(MP.phi);
double sp = sin(MP.phi);

double xx = X*ct - Y*st;
double yy = X*st*cp + Y*ct*cp - Z*sp;
double zz = X*st*sp + Y*ct*sp + Z*cp;
Cartesian local (xx, yy, zz);
return local;
  };


// Function to find the coordinates of a vector into the GLOBAL coordinate
// system. The Polar MeanPole (theta, phi) gives the orientation of the
// mean vector in the global plane of reference. The direction of this
// vector is the axis Z of the local coordinate system.

Cartesian Cartesian::global_coordinates (Polar& MP) {
double ct = cos(MP.theta);
double st = sin(MP.theta);
double cp = cos(MP.phi);
double sp = sin(MP.phi);

double xx = X*ct + Y*st*cp + Z*st*sp;
double yy = -X*st + Y*ct*cp + Z*sp*ct;
double zz = -Y*sp + Z*cp;

Cartesian global (xx, yy, zz);
return global;
  };
```

cell.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "cell.h"
#include "polygon.h"
#include "listpol.h"

#include <math.h>
#include <iostream.h>




#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)
extern ofstream out;
double Random01 ();

ostream& operator<< (ostream& o, Column& c)
{
int i;
o<<" X "<<c.X<<" Y "<<c.Y<<endl;

Cell* current = c.head_cell;

for (i=0; i<c.Ncells; ++i)
  {o<<i+1<<" Z "<<current->Z<<" V "<<current->value<<endl;
  if (current->GR)
    o<<" GR "<<current->GR<<endl;
  current = current->next_cell;};
return o;

};



// Function to add a new cell (e.g. porosity at elevation Z)
// to a column of data

void Column :: add_cell (Cell c)
{
Cell* newCell = new Cell (c.Z, c.value, c.GR);
newCell->next_cell = head_cell;
head_cell = newCell;
++Ncells;
return;
```

```
 };


// Function to find the average porosity of a column where it is intersected
by a polygon.
// Approximate calculation: X and Y of the column have been checked to be the
closest to
// the center of the polygon X and Y. Then the minimum and maximum Z are
calculated as
// Zmin=Zc-Re, and Zmax=Zc+Re, where Re is the equivalent radius of the
polygon. All values
// of porosity for cells which are at elevation between Zmin and Zmax are
sumemd, and
// the sum (divided by the number of intersected cells) is returned.

double Polygon :: find_average_porosity (Column& C)
{
int N=0, i;
double Zmax = center.Z + radius + 700.;
double Zmin = center.Z - radius + 700.;
//if (Zmin < (C.head_cell->Z))
//  return -1.;
double porosity = 0.;

Cell* current = C.head_cell;
for (i=0; i<C.Ncells; ++i)
  {
if (current->Z > Zmin && current->Z < Zmax)
{porosity += current->value;
N++;};
current = current->next_cell;
  };

if (N)
 porosity /= N;

//cout<<"R Zmin Zmax "<<radius<<" "<<Zmin<<" "<<Zmax<<endl;
//cout<<"average porosity "<<porosity<<endl;

return porosity;
};



// Function to find the average GR of a column where it is intersected by a
polygon.
// Approximate calculation: X and Y of the column have been checked to be the
closest to
// the center of the polygon X and Y. Then the minimum and maximum Z are
calculated as
// Zmin=Zc-Re, and Zmax=Zc+Re, where Re is the equivalent radius of the
polygon. All values
// of porosity for cells which are at elevation between Zmin and Zmax are
sumemd, and
```

```
// the sum (divided by the number of intersected cells) is returned.

double Polygon :: find_average_GR (Column& C)
{
int N=0, i;
double Zmax = center.Z + radius + 700.;
double Zmin = center.Z - radius + 700.;
double gamaray = 0.;

Cell* current = C.head_cell;
for (i=0; i<C.Ncells; ++i)
  {
if (current->Z > Zmin && current->Z < Zmax)
{gamaray += current->GR;
N++;};
current = current->next_cell;
  };

if (N)
 gamaray /= N;

//cout<<"R Zmin Zmax "<<radius<<" "<<Zmin<<" "<<Zmax<<endl;
//cout<<"average porosity "<<porosity<<endl;

return gamaray;
};


// Function to mark a polygon according to the averaged porosity of the
surrounding rock.
// This function is specifically written to interpret the Yates field
Stratamodel porosity
// data. Above a certain elevation (Zmax) there are no porosity readings
available, but it
// it known that the rock is very porous (unconformity) therefore the polygons
are marked
// with the same probability as in the strata where teh averaged porosity is
higher than
// maxPor.


boolian Polygon :: mark_by_porosity (Column* sgm, int Ncol, double maxPor,
double ratioP)
{
int N = find_closest_column (sgm, Ncol);
//cout<<"center "<<center;
//cout<<"closest column "<<N<<" "<<sgm[N].X<<" "<<sgm[N].Y<<endl;

double ave_por = find_average_porosity (sgm[N]);

if ((ave_por>maxPor || !ave_por) && Random01() > ratioP)
  {return FALSE;}
else
  { return TRUE;};
};
```

```
// Function to mark a polygon according to the averaged porosity and GR of the
surrounding rock.
// This function is specifically written to interpret the Yates field
Stratamodel porosity & GR
// data. Above a certain elevation (Zmax) there are no porosity readings
available, but it
// it known that the rock is very porous (unconformity) therefore the polygons
are marked
// with the same probability as in the strata where the rock is shale.


boolian Polygon :: mark_by_porosity_and_GR (Column* sgm, int Ncol, double
minPor, double maxGR, double ratioP)
{
int N = find_closest_column (sgm, Ncol);
//cout<<"center "<<center;
//cout<<"closest column "<<N<<" "<<sgm[N].X<<" "<<sgm[N].Y<<endl;

double ave_por = find_average_porosity (sgm[N]);
//if (ave_por <0. )
//{if ( Random01() > ratioP)
//  return FALSE;
//else
//  return TRUE;};

double ave_GR = find_average_GR (sgm[N]);
cout<<" P GR "<<ave_por<<" "<<ave_GR;

if ((!ave_por || ave_por<minPor && ave_GR>maxGR ) && Random01() > ratioP)
  {cout<<" No "<<endl; return FALSE;}
else
  {cout<<" Yes "<<endl;  return TRUE;};
};



// Procedure to find and return the number of the column in the Stratamodel in
which a polygon
// is located. The function finds the column for which the horizontal distance
between the
// center of the polygon and the axis (center) of the column is shortest.

int Polygon :: find_closest_column (Column* cp, int Ncol)
{
int i, colN=0;
double minD, temp;

double xx = cp->X - center.X;
double yy = cp->Y - center.Y;
minD = sqrt (xx*xx+yy*yy);

for (i=1; i<Ncol; ++i)
  {
xx = (cp+i)->X - center.X;
yy = (cp+i)->Y - center.Y;
temp = sqrt (xx*xx+yy*yy);
```

```
if (temp < minD)
  {minD = temp;
  colN = i;};
  };

return colN;
};


// Function to mark polygons-fractures by porosity, discarding those which are
in very
// porous rock. This function is specific for the Yates field where highly
porous rock is
// too ductile to farture.

void ListPolygons :: discard_high_porosity (Column* sgm, int Ncol, double
maxPor, double ratioP)

{
Node* current = head_pol;
while (current->next_pol)
  {
if (!current->next_pol->content->mark_by_porosity (sgm, Ncol, maxPor, ratioP))
  {
  current->next_pol = current->next_pol->next_pol;
  --Npol;}
else
  current = current->next_pol;

  };


if (!head_pol->content->mark_by_porosity (sgm, Ncol, maxPor, ratioP))
  {
  head_pol = head_pol->next_pol;
  --Npol;};
return;
};




// Function to mark polygons-fractures by porosity and GR, discarding those
which are in
//  shale (low porosity, high GR) . This function is specific for TRACT17.


void ListPolygons :: discard_shale (Column* sgm, int Ncol, double minPor,
double maxGR, double ratioP)

{
Node* current = head_pol;
while (current->next_pol)
  {
if (!current->next_pol->content->mark_by_porosity_and_GR (sgm, Ncol, minPor,
maxGR, ratioP))
```

```
    {
    current->next_pol = current->next_pol->next_pol;
    --Npol;}
  else
    current = current->next_pol;

    };


  if (!head_pol->content->mark_by_porosity_and_GR (sgm, Ncol, minPor, maxGR,
ratioP))
    {
    head_pol = head_pol->next_pol;
    --Npol;};
  return;
  };
```

circle.C

```
// ********************************************************************
// *                          G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998     *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                     *
// ********************************************************************

#include "polar.h"
#include "cartesian.h"
#include "point.h"
#include "plane.h"
#include "circle.h"

// Default constructor (radius=1, centered on origin, horizontal)

Circle :: Circle (void)
  {
  radius=1.;
  center.X=0.; center.Y=0.; center.Z=0.;
  support.A=0.; support.B=0.; support.C=1.; support.D=0.;
  };

// Constructor

Circle :: Circle (double& R, Point& C, Plane& Pl)
  {
  radius=R;
  center.X=C.X; center.Y=C.Y; center.Z=C.Z;
  support.A=Pl.A; support.B=Pl.B; support.C=Pl.C; support.D=Pl.D;
  };


// Procedure to test whether or not a point is inside the circle. Returns 1 if
// inside or on the perimeter, 0 if outside. Make sure that both point and
// circle are on the same plane.

int Circle::is_point_inside(Point& P)
{
Cartesian vect;
double l_vect;

vect.X=P.X-center.X;
vect.Y=P.Y-center.Y;
vect.Z=P.Z-center.Z;
l_vect=sqrt(vect.X*vect.X+vect.Y*vect.Y+vect.Z*vect.Z);

if(l_vect<=radius)
  return 1;
else
  return 0;
};
```

cubic.C

```
// *****************************************************************
// *                            G E O F R A C                      *
// *    Copyright Massachusetts Institute of Technology 1995-1998   *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein        *
// *                                                               *
// *           Don't use or modify without written permission       *
// *                    (contact einstein@mit.edu)                  *
// *****************************************************************

#include "cubic.h"


// This function calculates the coordinates of the normal vector
// to a cubic surface at a point p(X,Y,Z) on the surface.
// The normal vector is defined by the first derivative of the
// surface equation f(x,y,z)=0, i.e. n=(df/dx,df/dy,df/dz).
// For cubic surface (see definiton of class Cubic)
// the normal vector is (-3AXX-2BXY-CYY-2EX-FY-H, -BXX-2CXY-3DYY-FX-2GY-I, 1)

Cartesian Cubic::normal_at_point(Point& p) {
double xx = 3*A*p.X*p.X + 2*B*p.X*p.Y + C*p.Y*p.Y + 2*E*p.X + F*p.Y + H;
double yy = B*p.X*p.X + 2*C*p.X*p.Y + 3*D*p.Y*p.Y + F*p.X + 2*G*p.Y + I;

Cartesian* vector = new Cartesian(-xx, -yy, 1.);
return (*vector);
};


// This function finds the azimuth and latitude of a cubic surface at a point.
// Latitude is calculated as the angle from north (axis X). Positive values
// are to the east, negative values are to the west. Axis Y is east.
// Azimuth is calculated as the angle between the normal vector at the point
// and the vertical axis Z. A Polar (latitude, azimuth) is returned.
// The coordinate system (X,Y,Z) is left-handed.

Polar Cubic :: polar_normal_at_point (Point& p) {
Cartesian N = normal_at_point(p);

Polar* polar_normal = new Polar;
*polar_normal = N.convert_to_polar ();

return (*polar_normal);
};


// This function returns the strike and dip of a cubic surface at a point,
// calculated as the strike and dip of a tangent plane. Dip is between zero
// and pi/2 measured from the horizontal to the slope in a vertical plane.
// Strike is between -pi to pi, negative values are west from north,
// positive values are east from north.
// The system NTB is left-handed, where N is the normal vector, T is the
// strike tangent vector, and B=NxT is binormal pointing upward along
// the dip.

Polar Cubic :: strike_dip_at_point (Point& p) {
Polar SD = polar_normal_at_point(p);
SD.theta -= HalfPI;

if (SD.theta < -PI)
    SD.theta += TwoPI;
return SD;
};
```

divide.C

```
// *******************************************************************
// *                            G E O F R A C                        *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *         Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "line.h"
#include "polygon.h"
#include "listpol.h"

#define TRUE 1
#define FALSE 0
#define boolian int
#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)

#define x1(a, b, f)  (-b+sqrt(f))/(2*a)
#define x2(a, b, f)  (-b-sqrt(f))/(2*a)

extern double MeanArea;
double Random0a (double a);
double exp_value (double lambda);
int PoissonN (double lambda, double Area);
extern double AT;

// Procedure to create a 2D line from an angle alpha, distance D and a
// circle radius R  and center given by a Point C.  The line
// equation is X*cos(alpha) + Y*sin(alpha) = D. The two end points are
// calculated to lie on the circle. Z coordinates are ZERO.

Line :: Line (double angle, double D, double R, Point& C)
{
end1.Y = C.Y + R;
end2.Y = C.Y - R;
end1.X = C.X + (D-R*sin(angle))/cos(angle);
end2.X = C.X + (D+R*sin(angle))/cos(angle);
end1.Z = end2.Z = C.Z;

length = 2*(sqrt(R*R - D*D));
vector.X = -sin(angle);
vector.Y = cos(angle);
vector.Z = 0.;
};


// Procedure to find out if a line intersects a polygon in 2d.

boolian Polygon :: if_line_intersects (Line& l)
  {
```

```
int i;
Point* current = head;
for (i=0; i<noP; ++i)
  {
Line temp (*current, *current->next);
if (temp.intersect (l))
      return TRUE;

current=current->next;
  };
return FALSE;
  };



// Procedure to find out how a line intersects a polygon. Returns 0 if they
// don't intersect, 1 if one side of the polygon is intersected by the line,
// 2 if two sides are intersected, and 3 if the line is inside the polygon.
// The line and the polygon are 2D and lie in the SAME PLANE. It is essential
// to have calculated the center of the polygon correctly.

int Polygon :: how_line_intersects (Line& l)
  {
int i=0, c1=0, c2=0, c=0;
Point* current = head;
Line line1 (center, l.end1);
Line line2 (center, l.end2);
for (i=0; i<noP; i++)
  {
Line temp (*current, *current->next);
if (temp.intersect (line1))
      c1++;
if (temp.intersect (line2))
      c2++;
if (temp.intersect (l))
      c++;
current=current->next;
  };
if (!c1 && !c2)
  return 3;            // the line lies inside the polygon

if ((!c1 && c2) || (!c2 && c1))
  return 1;        // one end of the line is inside the polygon,
                    // the other end is outside
if (c1 && c2)
  { if (c)
    return 2;          // the line intersects two sides of the polygon
    else
      return 0;};       // the line does not intersect the polygon
  };


// Procedure to divide a polygon into two polygons by a line that intersects
// it (first make sure the line intersects two sides of the polygon using
// the function above. One of the two new polygons is returned, the other new
// polygon has replaced the original polygon. POLYGON AND LINE ARE IN 2D.
```

```
 Polygon Polygon :: divide_by_line (Line& L)
   {
 Point* new_points = new (Point[4]);
 Point* current = head;
 int i, j=0;
 for (i=0; i<noP && j<2; ++i)
   {
 Line a_line (*current,*current->next);
 if (L.intersect(a_line))
   {
     new_points[j] = L.intersection_with_line (a_line);
     new_points[j].next = current->next;
     current->next = new_points+j;
     j++;
     current=(current->next)->next;
  }
 else
    current=current->next;
 };

 noP+=4;
 new_points[2] = new_points[0];
 new_points[3] = new_points[1];

 Polygon* new_pol = new Polygon ();

 new_pol->head = new_points+2;
 new_pol->head->next = new_points[0].next;
 new_points[3].next = new_points[1].next;
 new_points[1].next = new_pol->head;
 new_points[0].next = new_points+3;

 current=new_pol->head;
 new_pol->noP = 0;
 do {
 new_pol->noP++;
 current=current->next;
 }
 while (!(current==new_pol->head));

 noP -= new_pol->noP;

 new_pol->setPole = setPole;
 new_pol->Pole = Pole;
 new_pol->strike = strike;
 new_pol->dip = dip;

 return (*new_pol);
 };


 // Procedure to tessellate a 2D polygon with Poisson lines. The following
 // property of a line process is used: if the line intensity is lambda (i.e.
 // the expected number of Poisson lines is lambda*the area), then the ordered
 // distances from an arbitrary point to the lines forms a Poisson process
 // with intensity 2*lambda (the distance increments are exponential). A
```

```
// Poisson line is produced by generating an angle alpha uniformly between 0
// and TwoPI, and a distance as a sum of exponential distances with point
// intensity 2*lambda until the cumulative distance becomes larger than
// the radius of the surcumscribed circle. Thus the number of lines is
// Poisson.


ListPolygons Poisson_lines_on_polygon (Polygon& initial, double lambda) {
ListPolygons* lp = new ListPolygons;
Node* current;
int i, j, lineInt;
double R = initial.minR_inscribe();
double expD = exp_value (lambda);
double cumD = -R + expD;
double angle;
lp->add_polygon (initial);
double xx = initial.center.X;
double yy = initial.center.Y;
double zz = initial.center.Z;
Point C (xx, yy, zz);

int k =0;
do {
++k;
angle = Random0a (TwoPI);
Line L (angle, cumD, R, C);

current = lp->head_pol;
j = lp->Npol;
for (i=0; i<j; ++i)
  {
if ( current->content->if_line_intersects (L))

      {Polygon* new_pol = new Polygon;
       *new_pol = current->content->divide_by_line (L);
       new_pol->find_area_radius_2d ();
       new_pol->find_center();
       current->content->find_area_radius_2d ();
       current->content->find_center();
       lp->add_polygon (*new_pol);
       };
       current = current->next_pol;};

expD = exp_value (lambda);
cumD += expD;

} while (cumD < R);
return *lp;

};

// Function to produce a Poisson line network with intensity lambda on a
// polygon. The lines produced are stored in a list. The initial polygon is
// not tessellated.

ListLines Poisson_lines_on_pol (Polygon& initial, double lambda)
{
 ListLines* LL=new ListLines;
 int i;
 double R=initial.minR_inscribe();
 double expD=exp_value(lambda);
 double cumD=-R+expD;
 double angle;
 double xx=initial.center.X;
 double yy=initial.center.Y;
 double zz=initial.center.Z;
 Point C(xx, yy, zz);

 do{
   angle=Random0a(TwoPI);
   Line L(angle, cumD, R, C);
   LL->add_line(L);
   expD=exp_value(lambda);
   cumD+=expD;
   } while (cumD<R);
return *LL;
};


// Procedure to perform a fractal tessellation to a polygon which is divided
// into polygons already. The same line tessellation which has been done
// to the original polygon is performed into all of the polygons of the
// list until the polygons become too small to be affected by a line
// network with the given intensity.

void ListPolygons :: fractal_tessellation (double lambda, double fractal)
{
Node* current = head_pol;
int i, j;
int N = Npol; cout<<"beginning "<<N<<endl;
ListPolygons* new_list = new ListPolygons;
for (i=0; i<Npol; ++i)
  {
     ListPolygons* new_lp = new ListPolygons;
*new_lp = Poisson_lines_on_polygon (*current->content, fractal*lambda);
new_list->add_listpol (*new_lp);
  current = current->next_pol;

  };
*this = *new_list;
   int N1 = Npol; cout<<" end "<<N1<<" new "<<N1-N<<endl;
return;
};



// Procedure to perform a fractal tessellation to a polygon which is divided
// into polygons already. The same line tessellation which has been done
// to the original polygon is performed into all of the polygons of the
// list until the polygons become too small to be affected by a line
// network with the given intensity.

void ListPolygons :: fractal_big_and_small (double lambda, double fractal)
{
```

```
Node* current = head_pol;
int i, j;
int N = Npol; cout<<"beginning "<<N<<endl;
ListPolygons* new_list = new ListPolygons;
for (i=0; i<Npol; ++i)
  {

if (current->content->area < AT*MeanArea)
 {
ListPolygons* new_lp = new ListPolygons;
*new_lp = Poisson_lines_on_polygon (*current->content, fractal*lambda);
new_list->add_listpol (*new_lp);
 }
else
 {
Polygon* new_pol = current->content;
new_list->add_polygon (*new_pol);
 };
current = current->next_pol;
  };

*this = *new_list;
   int N1 = Npol; cout<<" end "<<N1<<" new "<<N1-N<<endl;
return;
};



// Procedure to perform a fractal tessellation to a polygon which is divided
// into polygons already. The same line tessellation which has been done
// to the original polygon is performed into all of the polygons of the
// list until the polygons become too small to be affected by a line
// network with the given intensity.

void ListPolygons :: fractal_similar (double lambda, double fractal)
{
Node* current = head_pol;
int i, j;
int N = Npol; cout<<"beginning "<<N<<endl;
ListPolygons* new_list = new ListPolygons;
for (i=0; i<Npol; ++i)
  {

if (current->content->area> AT*MeanArea)
 {
ListPolygons* new_lp = new ListPolygons;
*new_lp = Poisson_lines_on_polygon (*current->content, fractal*lambda);
new_list->add_listpol (*new_lp);
 }
else
 {
Polygon* new_pol = current->content;
new_list->add_polygon (*new_pol);
 };
current = current->next_pol;
  };
```

```
*this = *new_list;
   int N1 = Npol; cout<<" end "<<N1<<" new "<<N1-N<<endl;
return;
};
```

fractal.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include <math.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)
#define max(a,b) ((a) > (b) ? (a) : (b))
#define min(a,b) ((a) < (b) ? (a) : (b))
double Xm, Ym;                          // Lateral boundaries of the volume
double TRatio;
double AngleMin, elongation;

double MeanArea, MeanA;
ofstream out;
double ratioMA, CA, gama;
double Datum;
double MaxPhi;
double Fk, bFk2, bFk1;

int sizeNpdf;
double maxR;
int Type;
int FLTiterations;
double FLT;
double P;
char mark;
double AT;

#define x1(a, b, f)  (-b+sqrt(f))/(2*a)
#define x2(a, b, f)  (-b-sqrt(f))/(2*a)


#include "polar.h"
#include "cartesian.h"
#include "point.h"
#include "line.h"
#include "surface.h"
```

```
#include "volume.h"
#include "plane.h"
#include "polygon.h"
#include "listpol.h"
#include "listline.h"
#include "borehole.h"
#include "stat.h"

time_t time(time_t *tloc);
double Random01();
double Random0a(double);
double  RandomBC(double , double);
double exp_value (double);
int PoissonN (double, double);
Polar ran_uniform_orientation();
Polar uniform_max_phi_orient(double);
Polar constant_orientation();
Polar Fisher_orientation (double);
Polygon make_initial (Plane&, Volume&);
ListPolygons Poisson_lines_on_polygon (Polygon& , double);


main ()
{
srand(time(0) * getpid());
int i;

/*******************************************************************
*             INPUT OF PARAMETERS FROM FILE "FractalInput.dat"
*******************************************************************/
ifstream in ("FractalInput.dat");
if (!in)
 cout<<"cannot open file"<<endl;

Datum = 0.;
in>>Xm>>Ym;                  // Extent of rectangular area (tract)


// INPUT OF TYPE OF MODEL MARKING
in>>mark;
in>>ratioMA>>maxR;      // Coefficients of the model plane, line,
                             // and marking processes
in>>CA;
in>>Type;                // Type of fractal line tessellation

in>>FLTiterations>>FLT>>P;         // parameters of Fractal Line Tessellation

in>>AngleMin;                // Minimum allowed angle for polygon shapes
in>>elongation;              // Maximum allowed elongation of polygons

in>>sizeNpdf;
int Nsize_int [sizeNpdf+1];
double sizeMax [sizeNpdf];

for (i=0; i<sizeNpdf; ++i)
  {Nsize_int[i] = 0;
  in>>sizeMax[i];};
```

```
Nsize_int[sizeNpdf] = 0;


double FracInt;
double MeanR;
double TRatio;


// INPUT OF PARAMETERS FOR INDIVIDUAL FRACTURE SETS

in>>TRatio;                    // Translation (non-coplanarity)
in>>MeanR;


in.close();



/**************************************************************************
*                       GENERATION OF FRACTURE PLANE
*************************************************************************/


Stat meanSD;
out.open ("FractalFrac.txt");
out<<endl<<"mark > "<<ratioMA<<" and < "<<maxR<<endl;
out<<"ratioMA CA "<<ratioMA<<" "<<CA<<endl;

MeanA = PI*MeanR*MeanR;
cout<<MeanR<<" "<<MeanA<<endl;
MeanArea = MeanA/CA;
double intensity = sqrt(PI/(MeanArea));

out<<"EXPECTED RADIUS AREA "<<MeanR<<" "<<MeanA<<endl;
out <<"LINE INTENSITY "<<intensity<<endl;

// GENERATION OF THE FRACTURE PLANE

ListPolygons* FracPlane = new ListPolygons;
Polygon initial;
Point p1 (Xm, Ym, 0.);
initial.add_point (p1);
Point p2 (-Xm, Ym, 0.);
initial.add_point (p2);
Point p3 (-Xm, -Ym, 0.);
initial.add_point (p3);
Point p4 (Xm, -Ym, 0.);
initial.add_point (p4);
initial.find_area_radius_2d();
Line l1 (p1, p2);
Line l2 (p2, p3);
Line l3 (p3, p4);
Line l4 (p4, p1);
ListLines LL;
LL.add_line(l1);
LL.add_line(l2);
LL.add_line(l3);
LL.add_line(14);


double InitialA = initial.area;

// TESSELLATION OF A FRACTURE PLANE INTO POLYGONS

*FracPlane = Poisson_lines_on_polygon (initial, intensity);
cout<<"PLT N polygons "<<FracPlane->Npol<<endl;


//ALL FRACTURES in a file "Fractal.m"
out.open ("Fractal.m");
out<<"Show [ ";
LL.print();
out<<" ,";
FracPlane->print();
out<<"]  "<<endl<<endl;
out.close();


meanSD = FracPlane->find_mean_sd();

out.open ("FractalFrac.txt", ios::app);

out<<"All polygons by Poisson LT "<<endl;
out<<"Mean SD Total N "<<meanSD<<" "<<FracPlane->Npol<<" "<<endl;
out<<"Mean/expected mean SD/Mean "<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean<<endl;
out.close ();

//SIZE OF ALL POLYGONS IN FILE "FractalSize.txt"
out.open ("FractalSize.txt");
out<<"ALL POLYGONS by Poisson line tessellation "<<endl;
out<<"Mean SD Total N "<<meanSD<<" "<<FracPlane->Npol<<" "<<endl;
out<<"Mean/expected mean SD/Mean "<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean<<endl<<endl;
int j;
FracPlane->size_distribution (sizeNpdf, Nsize_int, sizeMax, MeanA);
for (j=0; j<sizeNpdf; ++j)
  out<<sizeMax[j]<<"  "<<Nsize_int[j]<<endl;
out<<Nsize_int[sizeNpdf]<<endl<<endl;
out.close();


// MARKING OF POLYGONS WITH GOOD SHAPE:

if (mark == 'y' || mark == 'Y')
  {
FracPlane->mark_good_shape_and_P(AngleMin, elongation, P);
cout<<"after mark "<<FracPlane->Npol<<endl;

//ALL FRACTURES in a file "Fractal.m"
out.open ("Fractal.m", ios::app);
out<<"Show [ ";
LL.print();
out<<" ,";
```

```
FracPlane->print();
out<<"]  "<<endl<<endl;
out.close();

meanSD = FracPlane->find_mean_sd(MeanR, maxR);

out.open ("FractalFrac.txt", ios::app);
out<<"Marked polygons by Poisson LT "<<endl;
out<<"Mean SD Total N "<<meanSD<<" "<<FracPlane->Npol<<" "<<endl;
out<<"Mean/expected mean SD/Mean "<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean<<endl;
out.close ();


//SIZE OF MARKED POLYGONS IN FILE "FractalSize.txt"
out.open ("FractalSize.txt", ios::app);
out<<"Marked polygons by Poisson LT "<<endl;
out<<"Mean SD Total N "<<meanSD<<" "<<FracPlane->Npol<<" "<<endl;
out<<"Mean/expected mean SD/Mean "<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean<<endl<<endl;
int j;
FracPlane->size_distribution (sizeNpdf, Nsize_int, sizeMax, MeanA);
for (j=0; j<sizeNpdf; ++j)
  out<<sizeMax[j]<<"   "<<Nsize_int[j]<<endl;
out<<Nsize_int[sizeNpdf]<<endl<<endl;;
out.close();
  };

// FRACTAL LINE TESSELLATION if specified in INPUT

if (FLTiterations)
  {
int m;
for (m=0; m<FLTiterations; ++m)
  {
if (Type == 2)
FracPlane->fractal_big_and_small (intensity, FLT, AT);
else if (Type == 3)
FracPlane->fractal_similar(intensity, FLT, AT);
else
FracPlane->fractal_tessellation (intensity, FLT);

cout<<"before mark "<<FracPlane->Npol<<endl;

// MARK FRACTAL POLYGONS
if ( (mark == 'y' || mark == 'Y') && m == FLTiterations-1)
  {
FracPlane->mark_good_shape_and_P(AngleMin, elongation, P);
cout<<"after mark "<<FracPlane->Npol<<endl;
  };


//ALL FRACTURES in a file "Fractal.m"
if (m >= FLTiterations-2)
  {
out.open ("Fractal.m", ios::app);
out<<"Show [ ";
LL.print();
out<<" ,";
FracPlane->print();
out<<"]  "<<endl<<endl;
out.close();
  };

meanSD = FracPlane->find_mean_sd(MeanR, maxR);

out.open ("FractalFrac.txt", ios::app);
out<<"All polygons by Fractal LT iteration "<<m<<endl;
out<<"Mean SD Total N "<<meanSD<<" "<<FracPlane->Npol<<" "<<endl;
out<<"Mean/expected mean SD/Mean gama "<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean<<" "<<meanSD.total/InitialA<<endl;
out.close ();

//SIZE OF POLYGONS IN FILE "FractalSize.txt"
out.open ("FractalSize.txt", ios::app);
out<<"FLT POLYGONS after marking iteration "<<m<<endl;
out<<"Mean SD Total N "<<meanSD<<" "<<FracPlane->Npol<<" "<<endl;
out<<"Mean/expected mean SD/Mean gama"<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean<<" "<<meanSD.total/InitialA<<endl<<endl;
int j;
FracPlane->size_distribution (sizeNpdf, Nsize_int, sizeMax, MeanA);
for (j=0; j<sizeNpdf; ++j)
  out<<sizeMax[j]<<"  "<<Nsize_int[j]<<endl;
out<<Nsize_int[sizeNpdf]<<endl<<endl;;
out.close();

  };

  };


};
```

initial.C

```
// ********************************************************************
// *                          G E O F R A C                            *
// *    Copyright Massachusetts Institute of Technology 1995-1998     *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                     *
// ********************************************************************

#include "point.h"
#include "polygon.h"
#include "plane.h"
#include "volume.h"

#include <iostream.h>
#include <fstream.h>
extern ofstream out;
extern double Xm, Ym;
extern int Np;

#define x1(a, b, f)  (-b+sqrt(f))/(2*a)
#define x2(a, b, f)  (-b-sqrt(f))/(2*a)


// A function to create the polygon which a plane P cuts from the modeling
// volume V. If p intersects the top quadratic surface of V, then the
// curve of intersection is approximated by N straight line segments between
// points N+1 on the surface. Xm and Ym (also -Xm and -Ym) are the lateral
// boundaries of the volume.


Polygon make_initial (Plane& p, Volume& v) {

Polygon initial (p);
double A = p.A, B=p.B, C=p.C, D=p.D; int i;
double xx, yy, zz, a, b, c, d, e, f, aa, bb, BB, CC, ff;
a=v.top.A; b=v.top.B; c=v.top.C; d=v.top.D; e=v.top.E; f=v.top.F;

                            // First calculate points of intersections,
                           // if any, with the lateral planes at elevation 0.

if (B)                                  // Possible points of intersection
  { for (i=0; i<2; ++i)                 // with x=Xm and x=-Xm at z=0.
    { if (!i)
         xx=Xm;
      else
         xx=-Xm;
yy=(D-A*xx)/B;
    if (yy>=-Ym && yy<=Ym)
      {Point* P = new Point (xx, yy, 0.);
         initial.add_point (*P);
       if (!p.C)                          // if the plane is vertical
                  {double zz1=v.top.find_Z_on_surface (xx, yy);
```

```
                   Point* P1=new Point (xx, yy, zz1);
                   initial.add_point (*P1);};     }; };};

if (A)                                 // Possible points of intersection
  { for (i=0; i<2; ++i)                // with y=Ym and y=-Ym
      {if (!i)
         yy=Ym;
      else
         yy=-Ym;
xx=(D-B*yy)/A;
    if (xx>=-Xm && xx<=Xm)
      {Point* P=new Point (xx, yy, 0.);
         initial.add_point (*P);
       if (!p.C)                        // if the plane is vertical
                  {double zz1=v.top.find_Z_on_surface (xx, yy);
                   Point* P1=new Point (xx, yy, zz1);
                   initial.add_point (*P1);};     }; };};


if (C)                     // if the plane is not vertical
  { for (i=0; i<4; ++i)    // checking if it intersects the vertical edges
    {xx=v.P[i].X; yy=v.P[i].Y;
     zz=(D-A*xx-B*yy)/C;
     if (zz<=v.P[i].Z && zz>=-0.)
       {Point* P=new Point (xx, yy, zz);
        initial.add_point (*P);}; }; };

if (C&&A || C&&B)
  {
    { for (i=0; i<2; ++i)                   // Possible ntersections with
      {if (!i)                               // the top surface of the volume
      xx=Xm;                               // at x=Xm and x=-Xm
      else
         xx=-Xm;
aa=(D-A*xx)/C; bb=B/C;
BB=(bb+b*xx+e); CC=(a*xx*xx+d*xx-aa+f);
if (c)
  {
    ff=BB*BB-4*CC*c;
    if (ff>=0.)
      {yy=x1(c, BB, ff);  zz=aa-bb*yy;
       if (yy>=-Ym && yy<=Ym)
       {Point* P = new Point (xx, yy, zz);
          initial.add_point (*P);}; };
    if (ff>0.)
      {yy=x2(c, BB, ff);  zz=aa-bb*yy;
       if (yy>=-Ym && yy<=Ym)
       {Point* P = new Point (xx, yy, zz);
          initial.add_point (*P);}; }; }
else if (BB)
  {yy=-CC/BB;  zz=aa-bb*yy;
   if (yy>=-Ym && yy<=Ym)
       {Point* P = new Point (xx, yy, zz);
       initial.add_point (*P);}; }; };};

    { for (i=0; i<2; ++i)                     // Possible intersections with
      {if (!i)                                 // the top surface of the volume
      yy=Ym;                                 // at y=Ym and y=-Ym
    else
         yy=-Ym;
aa=(D-B*yy)/C; bb=A/C;
BB=(bb+b*yy+d); CC=(c*yy*yy+e*yy-aa+f);

if (a)
  {
    ff=BB*BB-4*CC*a;
    if (ff>=0.)
      {xx=x1(a, BB, ff);  zz=aa-bb*xx;
       if (xx>=-Xm && xx<=Xm)
       {Point* P = new Point (xx, yy, zz);
          initial.add_point (*P);}; };
    if (ff>0.)
      {xx=x2(a, BB, ff);  zz=aa-bb*xx;
       if (xx>=-Xm && xx<=Xm)
       {Point* P = new Point (xx, yy, zz);
       initial.add_point (*P);}; }; }
else if (BB)
  {xx=-CC/BB;  zz=aa-bb*xx;
   if (xx>=-Xm && xx<=Xm)
       {Point* P = new Point (xx, yy, zz);
       initial.add_point (*P);}; }; };};
  };

initial.find_center();
return initial;
}


// Function to add N points to a side of a polygon which is not linear
// but lies on a surface. The curved side is approximated by N+1 linear
// segments.

void Polygon::add_points_on_surface (Plane& pl, Surface& s, int N, double Zm)
{
double a=s.A, b=s.B, c=s.C, d=s.D, e=s.E, f=s.E;
if (a || b || c)
  {
Point* current = head;
int i, j;
double xx, yy, zz, x, y, z, ff;
double A=pl.A, B=pl.B, C=pl.C, D=pl.D;

Cartesian CB = pl.find_binormal();

for (i=0; i<noP; i++)
  {
if (s.is_point_on_surface(*current) && s.is_point_on_surface(*current->next))
{ xx =(current->next->X - current->X) / ((double)N+1.);
 yy = (current->next->Y - current->Y)  / ((double)N+1.);
 zz = (current->next->Z - current->Z)  / ((double)N+1.);
x = current->X; y=current->Y; z=current->Z;
for (j=0; j<N; ++j)
   {x+=xx; y+=yy; z+=zz;
    Point p (x, y, z);
    Line l (p, CB, Zm);
    Point *P = new Point;
    *P = s.intersection_by_line (l);
    P->next = current->next;
    current->next = P;
    noP ++;
    current=current->next;
   };
  return;
}
else
    current=current->next;
  };
  };
return; };
```

intersections.C

```
// *********************************************************************
// *                           G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998      *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein           *
// *                                                                   *
// *          Don't use or modify without written permission           *
// *                    (contact einstein@mit.edu)                     *
// *********************************************************************

#include "point.h"
#include "line.h"
#include "surface.h"
#include "plane.h"
#include "polygon.h"
#include "listpol.h"
#include "listline.h"
#include "circle.h"

#define x1(a, b, f)  (-b+sqrt(f))/(2*a)
#define x2(a, b, f)  (-b-sqrt(f))/(2*a)


// This procedure calculates the coordinates of the point in which a line
// intersects a surface. First use the procedures in "surface.C" to find out
// if the line intersects the surface at all. All calculations are in 3D.


Point Surface :: intersection_by_line (Line& L) {
Point* P = new Point;
double t, aa, bb, cc, ff;
double a = L.vector.X, b = L.vector.Y, c = L.vector.Z;
double X1 = L.end1.X, Y1 = L.end1.Y, Z1=L.end1.Z;

aa = A*a*a + B*a*b + C*b*b;
bb = 2*A*a*X1 + B*a*Y1 +B*b*X1 + 2*C*b*Y1 + D*a + E*b - c;
cc = A*X1*X1 + B*X1*Y1 + C*Y1*Y1 + D*X1 + E*Y1 + F - Z1;

if (!aa)
  {t = -cc/bb;
   P->X = a*t+X1; P->Y = b*t+Y1; P->Z = c*t+Z1;
   return *P; }
else
  {
ff = bb*bb - 4*aa*cc;
if (ff >= 0.)
  {
 t = x1(aa, bb, ff);
 if(t>=0. && t<=L.length)
   {P->X = a*t+X1; P->Y = b*t+Y1; P->Z = c*t+Z1;
   return *P; }
 else
   {
  t = x2(aa, bb, ff);
  P->X = a*t+X1; P->Y = b*t+Y1; P->Z = c*t+Z1;
  return *P; }; };
};
};
```

```
// Function to calculate if a plane intersects a line segment. The parametric
// equation of a line is used: x = X + at, y = Y + bt, z = Z + ct, where
// P(X, Y, Z) is a point on the line, (a, b, c) is the vector parallel
// to the line. If P is the first end of the segment, then the line intersects
// the plane if t is between 0 and the length.

boolian Line :: if_intersects_plane (Plane& p)
{

double a=vector.X, b=vector.Y, c=vector.Z;
double abc = p.A*a + p.B*b + p.C*c;
if (abc>= -0.00001 && abc<=0.00001)
  return FALSE;                        // the line is parallel to the plane
else
   {
double t = (p.D - p.A*end1.X - p.B*end1.Y - p.C*end1.Z)/abc;
if (t>=0. && t<=length)
    return TRUE;
else
    return FALSE;
   };
};
```

```
// Procedure to find the point of intersection of a line segment and a plane.
// First use the procedure to find out if they intersect at all.

Point Line :: intersection_with_plane (Plane& p)
{
double t = (p.D-p.A*end1.X-p.B*end1.Y-
p.C*end1.Z)/(p.A*vector.X+p.B*vector.Y+p.C*vector.Z);
double xx = end1.X + vector.X*t;
double yy = end1.Y + vector.Y*t;
double zz = end1.Z + vector.Z*t;

Point* intersection = new Point(xx, yy, zz);
return *intersection;
};
```

```
// Procedure to find out if a line intersects a polygon. Temporarily creates
// a plane through the polygon.
// If the line intersects the plane, the point of intersection is calculated.
// If the point is inside the polygon, the function returns TRUE.

boolian Polygon :: if_3d_line_intersects (Line& L)
{
Cartesian N (Pole);
Plane p (N, *head, setPole);
if (!(L.if_intersects_plane (p)))
  return FALSE;
else
  {
int how = TRUE;
Point P = L.intersection_with_plane (p);
Cartesian temp (P.X, P.Y, P.Z);
temp = temp.local_coordinates (Pole);
P.X = temp.X; P.Y=temp.Y; P.Z = temp.Z;
make_polygon_2d();

Line a_line (P, center);
if (if_line_intersects(a_line))
     how = FALSE;

make_polygon_3d();
return how;
  };
};
```

```
// Procedure to find the point of intersection of a line with a polygon.
// First use the procedure above to find out if the line intersects the
polygon
// at all. If the line intersects the polygon, then there is only one
// point where they intersect, and it is the point where the line segment
// intersects the plane of the polygon.

Point Polygon :: intersection_with_line (Line& L)
{
Cartesian N (Pole);
Plane p (N, *head, setPole);
Point* new_point = new Point;
*new_point = L.intersection_with_plane (p);
return *new_point;
};
```

```
// Procedure to find out if a polygon intersects another polygon in 3D.

boolian Polygon :: if_polygon_intersects (Polygon& pol)
{
Cartesian N (Pole);
Plane p (N, *head, setPole);
if (pol.if_intersects_plane(p))
  {
Line L = pol.intersection_with_plane (p);
make_polygon_2d();
L.local_coordinates(Pole);
if (if_line_intersects (L))
  { make_polygon_3d();
  return TRUE; }
else
  make_polygon_3d();};
return FALSE;
};
```

```
// Procedure to find out if a plane intersects a polygon. Returns TRUE if
// any of the sides of the polygon intersects the plane.

boolian Polygon :: if_intersects_plane(Plane& p)
{
Point* current = head;
int i;
for (i=0; i<noP; ++i)
  {
Line temp (*current, *current->next);
if (temp.if_intersects_plane (p))
  return TRUE;
else
  current = current->next;
  };
return FALSE;
};
```

```
// Procedure to calculate the line of intersection of a polygon with a plane.
// First use the procedure above to find out if they intersect at all.

Line Polygon :: intersection_with_plane (Plane& p)
{
Point* current = head;
int i, j=0;
Point first, second;

for (i=0; i<noP && j<2; ++i)
  {
Line l(*current, *current->next);
if (l.if_intersects_plane (p))
{ if(!j)
   first = l.intersection_with_plane (p);
  else
   second = l.intersection_with_plane (p);
++j; };
current = current->next;
  };
Line* trace = new Line(first, second);
return *trace;

};
```

```
// Procedure to calculate the list of lines which are the intersections of
// the list of polygons with a plane.


ListLines ListPolygons :: traces_on_plane (Plane& p)
{
ListLines* traces = new ListLines;
int i;
Node* current = head_pol;
for (i=0; i<Npol; ++i)
  {
if (current->content->if_intersects_plane (p))
    { Line* new_line = new Line;
      *new_line = current->content->intersection_with_plane(p);
      traces->add_line (*new_line);};
current = current->next_pol;
  };

return *traces;
};
```

```
// Procedure to find where a polygon is located relative to a surface.
// The function returns 1 if the polygon (all of its vertices) is entirely
// above the surface, -1 if it is entirely below the surface, or 0 if
// it intersects the surface.

int Polygon :: above_or_below_surface (Surface& s)
{
Point* current = head;
int i, counter =0;
int how;

for (i=0; i<noP; ++i)
  {
how = s.is_point_above_below (*current);
counter += how;
current = current->next;
  };


if (counter == noP )
  return 1;                        // All vertices are above the surface

if (counter == -noP)
  return -1;                     // All vertices are below the surface

else
  return 0;                        // The surface intersects the polygon
};
```

```
// Procedure to cut a polygon by a surface. The new polygon is the portion
// above the surface if the character argument is 'A' or 'a', or the portion
// below the surface if the character constant is 'B' or 'b'
// Use only after the function above has returned 0 for the polygon, i.e.
// if the polygon is intersected for sure by the surface.
// After returning from the function make sure to use the function for
// calculating area, center, and radius of the new polygon.

void Polygon :: cut_by_surface (Surface& s, char how)

{
int how_head = s.is_point_above_below (*head);
Point* current = head;
int i, j=0;
Point* new_points = new (Point[2]);

for (i=0; i<noP && j<2; ++i)
  {
Line a_line (*current,*current->next);
if (!s.how_line_intersect (a_line))
  {
    new_points[j] = s.intersection_by_line (a_line);

    new_points[j].next = current->next;
    current->next = new_points+j;
    j++;
    current=current->next->next;
 }
else
   current=current->next;
};


if ((how_head <=0 && (how == 'B' || how=='b')) || (how_head>=0&&(how == 'A' ||
how=='a')))
  { head = &new_points[0];
    head->next = &new_points[1];
  }
else
  {
   head = &new_points[1];
   head->next = &new_points[0];
  };

current = head;
noP = 0;
do {
noP++;
current=current->next;
}
while (!(current==head));

area_radius_3d();
return;
```

```
};

// Procedure to cut a polygon by a plane. The new polygon is the portion
// in front of the plane if the character argument is 'F' or 'f', or the
portion
// behind the plane if the character constant is 'B' or 'b'
// Use only after the function if_intersects_plane has returned TRUE for
// the polygon, i.e. if the polygon is intersected for sure by the plane.
// After returning from the function make sure to use the function for
// calculating area, center, and radius of the new polygon.

void Polygon::cut_by_plane(Plane &P, char how)

{
int how_head = P.is_point_front_behind(*head);
Point* current = head;
int i, j=0;
Point* new_points = new (Point[2]);

for (i=0; i<noP && j<2; ++i)
  {
Line a_line (*current,*current->next);
if (a_line.if_intersects_plane(P))
  {
    new_points[j] = a_line.intersection_with_plane(P);

    new_points[j].next = current->next;
    current->next = new_points+j;
    j++;
    current=current->next->next;
 }
else
   current=current->next;
};


if ((how_head <=0 && (how == 'B' || how=='b')) || (how_head>=0&&(how == 'F' ||
how=='f')))
  { head = &new_points[0];
    head->next = &new_points[1];
  }
else
  {
   head = &new_points[1];
   head->next = &new_points[0];
  };

current=head;
noP=0;
do
  {
  noP++;
  current=current->next;
  }
while(!(current==head));
area_radius_3d();
find_center();
```

```
return;
};
```

line.C

```
// *******************************************************************
// *                           G E O F R A C                         *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                    (contact einstein@mit.edu)                   *
// *******************************************************************

#include "point.h"
#include "line.h"
extern ofstream out;

// Function to print a line for graphics output (code in mathematica)

void Line::print(){
out<<"Graphics3D [Line [{";
 end1.print();
 out<<", ";
 end2.print();
 out << "}], PlotRange->All, Axes->True, Boxed->True]";
     return;
};


// Function to find if a Point P(X,Y,Z) is on a line segment L1L2.
// The function uses the parametric equation for a line (see "line.h")

boolian Line ::  is_point_on_line (Point& P)
{
double xx=0., yy=0., zz=0., t1, t2;
if (vector.X<-0.000001 || vector.X > 0.000001)
     xx = (P.X - end1.X) / vector.X;
if (vector.Y<-0.000001 || vector.Y>0.000001)
     yy = (P.Y - end1.Y) / vector.Y;
if (vector.Z<-0.000001 || vector.Z>0.000001)
     zz = (P.Z - end1.Z) / vector.Z;

if (xx && yy && zz)
  {
t1 = xx-yy, t2 = xx-zz;
if ((t1<0.00001) && (t1>-0.00001) && (t2<0.00001) && (t2>-0.00001) &&
(xx<=length) && (xx>0.))
  return TRUE;
  return FALSE;}
else if (xx && yy)
  {t1 = xx-yy;
if (t1<0.00001 && t1>-0.00001 && xx <= length && xx >= 0.)
  return TRUE;
  return FALSE;}
else if (xx && zz)
  {t1 = xx-zz;
if (t1<0.00001 && t1>-0.00001 && xx <= length && xx >= 0.)
```

```
   return TRUE;
   return FALSE;}
 else if (yy && zz)
   {t1 = yy-zz;
 if (t1<0.00001 && t1>-0.00001 && yy <= length && yy >= 0.)
   return TRUE;
   return FALSE;}
 else if (xx)
   {if (xx <= length && xx >= 0.)
   return TRUE;}
 else if (yy)
   {if (yy <= length && yy >= 0.)
   return TRUE;
   return FALSE;}
 else if (zz)
   {if (zz <= length && zz >= 0.)
     return TRUE;
     return FALSE;}
 else
      return FALSE;
 };


// Two functions necessary to determine if two lines in (X,Y) plane
// intersect or cross
// From "Algorithms in C++" by Sedgwick, Chapter 24

int ccw (Point& p0, Point& p1, Point& p2)
{
double dx1, dx2, dy1, dy2;
dx1= p1.X - p0.X; dy1 = p1.Y - p0.Y;
dx2= p2.X - p0.X; dy2 = p2.Y - p0.Y;
if (dx1*dy2 > dy1*dx2) return 1;
if (dx1*dy2 < dy1*dx2) return -1;
if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) return -1;
if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2)) return 1;
return 0;
};


boolian Line::intersect (Line& l)
{
if (((ccw(end1, end2, l.end1) * ccw(end1, end2, l.end2)) <=0 ) &&
((ccw(l.end1, l.end2, end1) * ccw(l.end1, l.end2, end2)) <=0))
return TRUE;
return FALSE;
};


// Function to find the point of intersection of a 2D line with
// another 2D line  in a plane
// FIRST USE THE TWO FUNCTIONS ABOVE TO FIND OUT IF THE TWO LINES
// INTERSECT AT ALL

Point Line::intersection_with_line (Line& l)
{
double xx, yy, slope1, slope2;
```

```
if (end1.X==end2.X)
   {
    xx = end1.X;
    slope2 = (l.end2.Y-l.end1.Y)/(l.end2.X-l.end1.X);
    yy = l.end1.Y + slope2*(xx-l.end1.X);}
else if (l.end1.X==l.end2.X)
    {
     xx = l.end1.X;
     slope1 = (end2.Y-end1.Y)/(end2.X-end1.X);
     yy = end1.Y + slope1*(xx-end1.X);}
else {
slope1 = (end2.Y-end1.Y)/(end2.X-end1.X);
slope2 = (l.end2.Y-l.end1.Y)/(l.end2.X-l.end1.X);

xx = (end1.Y-slope1*end1.X-l.end1.Y+slope2*l.end1.X)/(slope2-slope1);
yy = (slope2*end1.Y-slope2*slope1*end1.X-
slope1*l.end1.Y+slope1*slope2*l.end1.X)/(slope2-slope1);
};
Point* intersection = new Point;
*intersection = Point (xx, yy, end1.Z);
return *intersection;
};


// Procedure to find the local coordinates of a line in a frame of reference
// with Z axis given by Pole in the global f.o.r.

void Line :: local_coordinates (Polar& p)
{
Cartesian temp1, temp2;
temp1.X = end1.X; temp1.Y = end1.Y; temp1.Z = end1.Z;
temp1 = temp1.local_coordinates (p);
end1.X = temp1.X; end1.Y = temp1.Y; end1.Z = temp1.Z;

temp2.X = end2.X; temp2.Y = end2.Y; temp2.Z = end2.Z;
temp2 = temp2.local_coordinates (p);
end2.X = temp2.X; end2.Y = temp2.Y; end2.Z = temp2.Z;

vector.X = end2.X - end1.X;
vector.Y = end2.Y - end1.Y;
vector.Z = end2.Z - end1.Z;

return;
};


// Procedure to find the global coordinates of a line.

void Line :: global_coordinates (Polar& p)
{
Cartesian temp1, temp2;
temp1.X = end1.X; temp1.Y = end1.Y; temp1.Z = end1.Z;
temp1 = temp1.global_coordinates (p);
end1.X = temp1.X; end1.Y = temp1.Y; end1.Z = temp1.Z;
```

```
temp2.X = end2.X; temp2.Y = end2.Y; temp2.Z = end2.Z;
temp2 = temp2.global_coordinates (p);
end2.X = temp2.X; end2.Y = temp2.Y; end2.Z = temp2.Z;

vector.X = end2.X - end1.X;
vector.Y = end2.Y - end1.Y;
vector.Z = end2.Z - end1.Z;

return;
};

// Function to find the shortest distance between Point P and the
// line in 3D space.

double Line::shortest_dist_to_point(Point& P)
{
Cartesian unit;
unit.X=(end2.X-end1.X)/length;
unit.Y=(end2.Y-end1.Y)/length;
unit.Z=(end2.Z-end1.Z)/length;

Cartesian vect;
double l_vect;
vect.X=P.X-end1.X;
vect.Y=P.Y-end1.Y;
vect.Z=P.Z-end1.Z;
l_vect=sqrt(vect.X*vect.X+vect.Y*vect.Y+vect.Z*vect.Z);

double l_inter;
double dist;
l_inter=vect.X*unit.X+vect.Y*unit.Y+vect.Z*unit.Z;
dist=sqrt(l_vect*l_vect-l_inter*l_inter);

return dist;
};


// Function to find the intersection between the line and a perpendicular
// passing through point P. The line is considered as infinite.

Point Line::intersection_from_point(Point& P)
{
Cartesian unit;
unit.X=(end2.X-end1.X)/length;
unit.Y=(end2.Y-end1.Y)/length;
unit.Z=(end2.Z-end1.Z)/length;

Cartesian vect;
double l_vect;
vect.X=P.X-end1.X;
vect.Y=P.Y-end1.Y;
vect.Z=P.Z-end1.Z;
l_vect=sqrt(vect.X*vect.X+vect.Y*vect.Y+vect.Z*vect.Z);

double l_inter;
l_inter=vect.X*unit.X+vect.Y*unit.Y+vect.Z*unit.Z;
```

```
Point intersect;
intersect.X=end1.X+l_inter*unit.X;
intersect.Y=end1.Y+l_inter*unit.Y;
intersect.Z=end1.Z+l_inter*unit.Z;

return intersect;
};

// NEWLY ADDED FUNCTION. AUG 18, 2000
// determines if a point is an endpoint of the line.  this is used in
// the intersection calculation algorithm.
/*bool Line::is_point_an_endpoint_of_line (Point& p)
{
if (end1==p || end2==p)
  return true;
return false;
};*/

// NEWLY ADDED FUNCTION. AUG 18, 2000
// calculates the midpoint of the line segment
Point& Line::midpt_of_the_line()
{
Point *p=new Point;
p->X=(end1.X+end2.X)/2;
p->Y=(end1.Y+end2.Y)/2;
p->Z=(end1.Z+end2.Z)/2;
return (*p);
};

//NEWLY ADDED MEMBER JUN 13, 2001
// calculates the plunge of the line in degrees
// angle with horizontal
 double Line::compute_plunge()
{
  Point p0;
  Point p1(vector.X, vector.Y, 0);
  Line l1(p0, p1);
  double plunge=angle_with_line(l1);
  return plunge*180/PI;
}

//NEWLY ADDED MEMBER JUN 13, 2001
// calculates the trend of the line in degrees
// angle with north, Y-axis, clockwise
 double Line::compute_trend()
{
  Point p0;
  Point p1(vector.X, vector.Y, 0);
  Point p2(0, 1, 0);
  Line l1(p0, p1);
  Line l2(p0, p2);
  double trend=l1.angle_with_line(l2);

if(vector.X>=0 && vector.Z>0) return trend*180/PI+180;

if(vector.X<0 && vector.Z<0) return 360-trend*180/PI;
if(vector.X<0 && vector.Z>0) return 180-trend*180/PI;

else return trend*180/PI;
}
```

listline.C

```
// *****************************************************************
// *                          G E O F R A C                        *
// *    Copyright Massachusetts Institute of Technology 1995-1998   *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein        *
// *                                                               *
// *          Don't use or modify without written permission        *
// *                   (contact einstein@mit.edu)                   *
// *****************************************************************

#include "listline.h"
#include "line.h"
#include "circle.h"
#include "polygon.h"
#include <math.h>
#include <iostream.h>
extern ofstream out;

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)



// Function to add a new line at the head of a list of lines.

void ListLines :: add_line (Line& l)
{
Node_line* new_head = new Node_line;
new_head->content = &l;
new_head->next_line = head_line;
head_line = new_head;
++Nline;
//cout<<"in add_line()\n";
return;
};



// Function to print a list of lines for non graphical output.

ostream& operator<< (ostream& o, ListLines& ll)
 {

o<<"N LINE "<<ll.Nline<<endl;
Node_line* current = ll.head_line;
int i;

for (i=0; i<ll.Nline; ++i)
{
  o<<(*current->content) <<endl;
current=current->next_line;
 };
 return  o;
  };


 // Function to print a list of lines for graphics output as mathematica
 // code

 void ListLines :: print()
 {
 int i;
 int j=0;
 Node_line* current = head_line;
 if (current->content)
   {
 for (i=0; i<Nline-1; ++i)
   {if (!j)
     {
    current->content->print();
    out<<",  ";};
    j++;
    if (j==1)
        j=0;
    current=current->next_line;};
current->content->print();
   };
return;
};




// Function to find mean and standard deviation of the lengths of a list
// of lines

Stat ListLines :: find_mean_sd_length ()
{

int i ;
double total=0., sd=0., mean =0.;
Node_line* current = head_line;
double L;

if (Nline)
  {
for (i=0; i<Nline; ++i)
  {
    L = current->content->length;
    total += L;
    sd += L*L;
 current=current->next_line;
};

mean = total / Nline;
if (Nline > 1)
 sd = sqrt ((sd - Nline*mean*mean) / (Nline -1));
```

```
  };

Stat meanSD (mean, sd, total);
return meanSD;
};



// Function to count the fracture traces located within the window defined by
// circle circ. Only the traces with both ends censored are considered.

int ListLines::count_traces_0(Circle& circ)
{
int n=0;
int i;
double min_dist;
double dist_inter;
Point inter;

Node_line* current=head_line;

if(Nline)
  {
  for(i=0; i<Nline; ++i)
    {
    if((!(circ.is_point_inside(current->content->end1))) &&
(!(circ.is_point_inside(current->content->end1))))
      {
      inter=current->content->intersection_from_point(circ.center);
      min_dist=sqrt((inter.X-circ.center.X)*(inter.X-circ.center.X) +(inter.Y-
circ.center.Y)*(inter.Y-circ.center.Y) +(inter.Z-circ.center.Z)*(inter.Z-
circ.center.Z));
      dist_inter=sqrt((inter.X-current->content->end1.X)*(inter.X-current-
>content->end1.X) +(inter.Y-current->content->end1.Y)*(inter.Y-current-
>content->end1.Y) +(inter.Z-current->content->end1.Z)*(inter.Z-current-
>content->end1.Z));
      if(min_dist<circ.radius && dist_inter>0. && dist_inter<current->content-
>length)
        n++;
      };
    current=current->next_line;
  };
  };
return n;
};


// Function to count the fracture traces located within the window defined by
// circle circ. Only the traces with one end censored are considered.

int ListLines::count_traces_1(Circle& circ)
{
int n=0;
int i;

Node_line* current=head_line;
```

```
if(Nline)
  {
  for(i=0; i<Nline; ++i)
    {
    int res1=circ.is_point_inside(current->content->end1);
    int res2=circ.is_point_inside(current->content->end2);
    if((res1 && (!res2)) || ((!res1) && res2))
      n++;
    current=current->next_line;
    };
  };
return n;
};


// Function to count the fracture traces located within the window defined by
// circle circ. Only the traces with no end censored are considered (both ends
// within the circle).

int ListLines::count_traces_2(Circle& circ)
{
int n=0;
int i;

Node_line* current=head_line;

if(Nline)
  {
  for(i=0; i<Nline; ++i)
    {
    int res1=circ.is_point_inside(current->content->end1);
    int res2=circ.is_point_inside(current->content->end2);
    if(res1 && res2)
      n++;
    current=current->next_line;
    };
  };
return n;
};

// Function to find the mean length of the lines crossing the polygon. All the
// lines either completely cross the polygon (two intersections) or don't
touch
// it, since they are Poisson lines. This procedure can't be used with the
// traces on an outcrop.

double ListLines::find_mean_length_on_pol(Polygon& pol)
{
Node_line* current=head_line;
double M=0.;
Point Int[2];
double tot_length=0.;
int N=0;
int i, j, k;

for(i=0; i<Nline; ++i)
  {
```

```
  if(pol.if_line_intersects(*current->content))
    {
    Point*  cur_vert=pol.head;
    k=0;
    for(j=0; j<pol.noP; ++j)
      {
      Line temp(*cur_vert, *cur_vert->next);
      if(temp.intersect(*current->content) && k<2)
        {
        Int[k]=temp.intersection_with_line(*current->content);
        k++;
        };
      cur_vert=cur_vert->next;
      };
    N++;
    Line temp_int(Int[0], Int[1]);
    tot_length=+ temp_int.length;
    };
  current=current->next_line;
  };

M=tot_length/N;
return M;
};

// Function to find the mean length of the lines crossing the circle. All the
// lines either completely cross the circle (two intersections) or don't touch
// it, since they are Poisson lines. This procedure can't be used with the
// traces on an outcrop.

double ListLines::find_mean_length_on_circ(Circle& circ)
{
Node_line* current=head_line;
double M=0.;
Point inter;
double tot_length=0., temp_length, min_dist;
int N=0;
int i;

for(i=0; i<Nline; ++i)
  {
  inter=current->content->intersection_from_point(circ.center);
  min_dist=sqrt((inter.X-circ.center.X)*(inter.X-circ.center.X) +(inter.Y-
circ.center.Y)*(inter.Y-circ.center.Y) +(inter.Z-circ.center.Z)*(inter.Z-
circ.center.Z));
  if(min_dist<circ.radius)
    {
    temp_length=2*sqrt(circ.radius*circ.radius - min_dist*min_dist);
    tot_length=+temp_length;
    N++;
    };
  current=current->next_line;
  };

M=tot_length/N;
return M;
};
```

```
//NEWLY ADDED FUNCTION OCT. 8, 2000
// removes a line from the list
// used to edit the ListLines generated when finding the intersection lines
// between polygons
void ListLines::remove_line(Line &l)
{
Node_line *p;
for (p=head_line;p->next_line->content!=&l; p=p->next_line)
  {/*
     this does nothing!! we just need to get to the Node_line just before
     the one that holds l as its contents
   */}

//p is now the Node_line just before the one that holds l as its contents

 Node_line *tempor=p->next_line;// tempor points to Node_line holding l

 p->next_line=p->next_line->next_line;// skips the Node_line holding l

 delete tempor;//deletes the removed line!  check this deletion!!

Nline--;
};
```

listlistpol.C

```
// *******************************************************************
// *                           G E O F R A C                         *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                  (contact einstein@mit.edu)                     *
// *******************************************************************

#include "listlistpol.h"
#include "listpol.h"
#include "polygon.h"
#include "surface.h"
#include "box.h"
#include "stat.h"
#include <math.h>
#include <iostream.h>

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)
extern ofstream out;

double Random01 ();

// Function to add a new list of polygon to a list of lists of polygons
// as the new head list of the list of lists.

void ListListPol :: add_listpol (ListPolygons& LP)
{
Node_list* new_head = new Node_list;
new_head->content = &LP;
new_head->next_list = head_list;
head_list = new_head;
++Nlist;
return;
};


// Procedure to name all the objects of the list (member int name_list),
// as well as all polygons of each list (member int name_list). The name
// is i 1<= i <= Nlist.

void ListListPol :: name_list ()
{
Node_list* cur_list=head_list;
Node* current;
int i, j;

for(i=0; i<Nlist; ++i)
  {
  cur_list->content->name_list=i+1;
  current=cur_list->content->head_pol;
  for(j=0; j<cur_list->content->Npol; ++j)
    {
    current->content->name_list=i+1;
    current=current->next_pol;
    };
  cur_list=cur_list->next_list;
  };

return;
};
```

listpol.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission        *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "listpol.h"
#include "polygon.h"
#include "surface.h"
#include "box.h"
#include "stat.h"
#include "listlistpol.h"
#include <math.h>
#include <iostream.h>

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)
extern ofstream out;

double Random01 ();

// Function to add a new polygon to a list of polygons as the
// new head polygon of the list.

void ListPolygons :: add_polygon (Polygon& pol)
{
Node* new_head = new Node;
new_head->content = &pol;
new_head->next_pol = head_pol;
head_pol = new_head;
++Npol;
return;
};

// Function to add a new polygon to a list of polygons as the
// tail polygon of the list.

void ListPolygons :: add_polygon_tail (Polygon& pol)
{
Node* current=head_pol;
Node* new_tail=new Node;
new_tail->content=&pol;
new_tail->next_pol=0;
int i;

for(i=0; i<Npol; ++i)
  {
  if(current->next_pol)
    current=current->next_pol;
```

```
   };

 current->next_pol=new_tail;
 ++Npol;
 return;
 };


 // Procedure to remove a polygon from a list of polygons.

 void ListPolygons :: remove_from_list (Polygon& pol)
 {
 Node* current=head_pol;

 if(pol.name == head_pol->content->name)
   {
   head_pol=head_pol->next_pol;
   --Npol;
   }
 else
   {
   do
     {
     if((current->next_pol) && (pol.name == current->next_pol->content->name))
       {
       current->next_pol=current->next_pol->next_pol;
       --Npol;
       }
     else
       current=current->next_pol;
     }
   while(current);
   };

return;
};


// Function to print a list of polygons for non graphical output.

ostream& operator<< (ostream& o, ListPolygons& lp)
 {

o<<"N POl: "<<lp.Npol<<endl;
Node* current = lp.head_pol;
int i;

for (i=0; i<lp.Npol; ++i)
{
  o<<i+1<<endl;
  o<<(*current->content) <<endl;
current=current->next_pol;
 };

return  o;
 };
```

```
// Function to print a list of polygons for graphics output as mathematica
// code

void ListPolygons :: print()
{
int i;
int j=0;
Node* current = head_pol;
if (current->content)
  {
for (i=0; i<Npol-1; ++i)
  {
    if (!j)
    {
   current->content->print();
   out<<",  ";};
   j++;
   if (j==1)
       j=0;
   current=current->next_pol;};
current->content->print();
  };
return;
};

// Function to print the centers of the polygons belonging to a list

void ListPolygons :: center_print()
{
int i;
Node* current = head_pol;

  for (i=0; i<Npol; ++i)
  {
    out<<current->content->name<<" "<<current->content->center.X<<"
"<<current->content->center.Y<<" "<<current->content->center.Z<<" "<<current-
>content->area<<" "<<current->content->strike<<" "<<current->content-
>dip<<endl;
    current=current->next_pol;
  };
return;
  };

// Procedure to assign a name (member int name) to each one of the polygons
// of the list. The name is equal to the rank of the polygon in the list.

void ListPolygons :: name_pol ()
{
int i;
Node* current = head_pol;

for (i=0; i<Npol; ++i)
  {
  current->content->name=i+1;
  current=current->next_pol;
  };
return;
};

// Procedure to create a copy of a list of polygons

ListPolygons ListPolygons :: make_copy()
{
ListPolygons* copy = new ListPolygons;
Node* current=head_pol;
copy->add_polygon(*current->content);

int i;
for(i=0; i<Npol-1; ++i)
  {
  current=current->next_pol;
  copy->add_polygon_tail(*current->content);
  };

return *copy;
};


// Function to make all polygons in a list of polygons 2d

void ListPolygons :: make_listpol_2d ()
{
Node* current = head_pol;
int i;

for (i=0; i<Npol; i++)
  {current->content->make_polygon_2d();
   current->content->find_center();
   current=current->next_pol; };

return;
};


// Function to make all polygons in a 2d list of polygons 3d

void ListPolygons :: make_listpol_3d ()
{

Node* current = head_pol;
int i;

for (i=0; i<Npol; i++)
  {current->content->make_polygon_3d();
 current->content->find_center();
 current=current->next_pol; };
return;
};
```

```
// Function to find the areas and radii of a list of 2d polygons

void ListPolygons :: find_area_radius_2d () {

Node* current = head_pol;
int i;

for (i=0; i<Npol; ++i)
  {current->content->find_area_radius_2d();
  current=current->next_pol; };
return;
};


// Function to discard the polygons with bad shape, i.e. only polygons with
// angles larger than MinAngle and elongation less than MaxE are retained
// in the list

void ListPolygons :: mark_good_shape (double MinAngle, double MaxE)
{

Node* current = head_pol;

do
  {
if ((current->next_pol) && !(current->next_pol->content->if_good_shape
(MinAngle, MaxE)))
     { if (current->next_pol->next_pol)
                  delete current->next_pol->content;
       current->next_pol = current->next_pol->next_pol;
       --Npol; }
else
   current=current->next_pol;
  }
while (current);

if(!(head_pol->content->if_good_shape (MinAngle, MaxE)))
{
head_pol = head_pol->next_pol;
  --Npol;
};

return;
};


// Function to discard all polygons with bad shape, i.e. only polygons with
// angles larger than MinAngle and elongation less than MaxE are retained
// in the list. The function also marks the good polygons with probability P,
// only P% of the good polygons are kept

void ListPolygons :: mark_good_shape_and_P (double MinAngle, double MaxE,
double P)
{
```

```
Node* current = head_pol;

do
  {
if ((current->next_pol) && (!(current->next_pol->content->if_good_shape
(MinAngle, MaxE)) || Random01 () > P))
  { if (current->next_pol->next_pol)
                  delete current->next_pol->content;
       current->next_pol = current->next_pol->next_pol;
       --Npol; }
else
   current=current->next_pol;
  }
while (current);

if(!(head_pol->content->if_good_shape (MinAngle, MaxE)))
{
head_pol = head_pol->next_pol;
  --Npol;
};

return;
};


// Function to mark the polygons below (B) or above (A) a surface with
probability P.
// Such marking decreases P32 below or above the surface to P of P32 initial.
0<P<1.

void ListPolygons :: mark_to_surface (Surface& S, double P, char how)
{

int HOW;

Node* current = head_pol;
do
  {
if (current->next_pol)
  {
HOW =  S.is_point_above_below (current->next_pol->content->center);
if ((how == 'A' && HOW > 0 || how == 'B' && HOW < 0) && Random01 () > P)
     { current->next_pol = current->next_pol->next_pol;
     --Npol; }
else
  current=current->next_pol;
}

else
 current=current->next_pol;
  }
while (current);


HOW =  S.is_point_above_below (head_pol->content->center);
if ((how == 'A' && HOW > 0 || how == 'B' && HOW < 0) && Random01 () > P)
{head_pol = head_pol->next_pol;
```

```
  --Npol;
};

return;
};



// Function to add a list of polygons to a  list of polygons. The main list
// (which calls the function) now contains the two lists. The argument list is
// added at the head of the main list.

void ListPolygons :: add_listpol (ListPolygons& lp1)
{

if (lp1.Npol)
  {
if (!Npol)
  {head_pol = lp1.head_pol;}
else
  {
int i;
Node* current = lp1.head_pol;

for (i=0; i<lp1.Npol-1; ++i)
    current=current->next_pol;
current->next_pol = head_pol;
head_pol = lp1.head_pol;
  };
Npol += lp1.Npol;
  };
return;
};



// Function to find mean and standard deviation of the areas of a list
// of polygons.

Stat ListPolygons :: find_mean_sd ()
{

int i ;
double total=0., sd=0., mean =0.;
Node* current = head_pol;
double area;

if (Npol)
  {
for (i=0; i<Npol; ++i)
  {
    area = current->content->area;
    total += area;
    sd += area*area;
 current=current->next_pol;
};
```

```
 mean = total / Npol;
 if (Npol > 1)
  sd = sqrt ((sd - Npol*mean*mean) / (Npol -1));

   };

 Stat meanSD (mean, sd, total);
 return meanSD;
 };


 // Function to find the WEIGHTED  average and standard deviation of the
 // strikes of a list of polygons (weight = area).

 Stat ListPolygons :: find_mean_sd_strike ()
 {

 int i ;
 double total=0., totals=0., sd=0., mean =0.;
 Node* current = head_pol;
 Stat stat_area=find_mean_sd();
 double str, ar;

 if (Npol)
   {
 for (i=0; i<Npol; ++i)
   {
     str = current->content->strike;
     ar = current->content->area;
     total += ar*str;
     totals += ar*str*ar*str;
  current=current->next_pol;
 };

 if (Npol > 1)
  totals = sqrt ((totals - total*total/Npol) / (Npol -1));

 mean = total / (Npol*stat_area.Mean);
 sd = sqrt (totals*totals - stat_area.SD*stat_area.SD*mean*mean) /
 stat_area.Mean;
   };

 Stat meanSD (mean, sd, total);
 return meanSD;
 };


 // Function to find the WEIGHTED  average and standard deviation of the
 // dips of a list of polygons (weight = area).

 Stat ListPolygons :: find_mean_sd_dip ()
 {

 int i ;
 double total=0., totald=0., sd=0., mean =0.;
 Node* current = head_pol;
```

```
Stat stat_area=find_mean_sd();
double dip, ar;

if (Npol)
  {
for (i=0; i<Npol; ++i)
  {
    dip = current->content->dip;
    ar = current->content->area;
    total += ar*dip;
    totald += ar*dip*ar*dip;
 current=current->next_pol;
};

if (Npol > 1)
 totald = sqrt ((totald - total*total/Npol) / (Npol -1));

mean = total / (Npol*stat_area.Mean);
sd = sqrt (totald*totald - stat_area.SD*stat_area.SD*mean*mean) /
stat_area.Mean;
  };

Stat meanSD (mean, sd, total);
return meanSD;
};



// Function to find mean and standard deviation of the areas of a list
// of polygons. At this point polygons that are too large are discarded from
// the list o ffractures.

Stat ListPolygons :: find_mean_sd (double meanR, double maxRatio)
{

int i=0, j=0 ;
double total=0., sd=0., mean =0.;
Node* current = head_pol;
double area;
cout<<" calculate statistics of "<<Npol<<" polygons "<<endl;

while (current->next_pol)
 {

if (current->next_pol && (current->next_pol->content->radius <
meanR*maxRatio))
  {
    area = current->next_pol->content->area;
    total += area;
    sd += area*area;
 current=current->next_pol;
  }
else
  {
current->next_pol = current->next_pol->next_pol;
   -- Npol;
  };
 };


 if (head_pol->content->radius < meanR*maxRatio)
   {
     area = head_pol->content->area;
     total += area;
     sd += area*area;
   }
 else
   {
   head_pol = head_pol->next_pol;
    -- Npol;
   };


mean = total / Npol;
if (Npol > 1)
 sd = sqrt ((sd - Npol*mean*mean) / (Npol -1));


Stat meanSD (mean, sd, total);
return meanSD;
};



// Procedure to translate the polygons in a list of polygons in the vicinity
// of their original position. Bigger polygons are translated closer to the
// the original plane than smaller polygons.

void ListPolygons :: translate_2d (double meanR, double ratio)
{
if(ratio)
  {
Node* current = head_pol;
int i;

for (i=0; i<Npol; ++i)
  {
current->content->translate_2d (meanR, ratio);
current = current->next_pol;
  };
  };
return;

};



// Procedure to cut a list of polygons above of below a surface. Cuts
// the polygons only if they are intersected by the surface at all.
// Retains the portion above the surface if the character argument is 'A'
// or 'a', or the portion below the surface if the character is 'B' or 'b'

void ListPolygons :: cut_by_surface (Surface& s, char how)
```

```
{
Node* current = head_pol;
int i;

for (i=0; i<Npol; ++i)
  {
if (!current->content->above_or_below_surface (s))
  current->content->cut_by_surface (s, how);

current = current->next_pol;

  };
return;
};


// Procedure to cut a list of polygons "in front" or "behind" a plane. Cuts
// the polygons only if they are intersected by the plane at all.
// Retains the portion in front of the plane if the character argument is 'F'
// or 'f', or the portion behind the plane if the character is 'B' or 'b'
// Cutting occurs only if a randomly generated number is smaller than a
// specified threshold.

void ListPolygons::cut_by_plane(Plane &P, char how, double CutPr)

{
Node* current=head_pol;
int k;
double test;
for(k=0; k<Npol; ++k)
  {
  test=Random01();
  if((current->content->if_intersects_plane(P)) && (test<CutPr))
    current->content->cut_by_plane(P, how);
  current=current->next_pol;
  };
return;
};


// Two procedure to cut polygons by two surfaces. The first Surface sA has to
// be above the second Surface sB thorughout the extent of the modeled region
// i.e. for -Xm<x<Xm and -Ym<y<Ym. The location of the center of every polygon
// is calculated to find out if it is below or above the surfaces. The first
// function retains only the portion of polygons which is between the two
// surfaces, if the center is located between the surfaces. Otherwise the
// polygons are discarded.


void ListPolygons :: cut_between_two_surfaces  (Surface& sA, Surface& sB)
{
Node* current=head_pol;
int i, cA, cB;

while (current->next_pol)
  {
cA = sA.is_point_above_below (current->next_pol->content->center);
```

```
cB = sB.is_point_above_below (current->next_pol->content->center);

if (cA<0 && cB>0)                  // the center is between the surfaces
  {
                                   // retain only portion  below sA
if (!current->next_pol->content->above_or_below_surface (sA))
  current->next_pol->content->cut_by_surface (sA, 'B');
                                   // retain only portion above sB
if (!current->next_pol->content->above_or_below_surface (sB))
  current->next_pol->content->cut_by_surface (sB, 'A');

current = current->next_pol;
  }

else
{ current ->next_pol = current->next_pol->next_pol;
--Npol; }; };



cA = sA.is_point_above_below (head_pol->content->center);
cB = sB.is_point_above_below (head_pol->content->center);

if (cA<0 && cB>0)                  // the center is between the surfaces
  {
                                   // retain only portion  below sA
if (!head_pol->content->above_or_below_surface (sA))
  head_pol->content->cut_by_surface (sA, 'B');
                                   // retain only portion above sB
if (!head_pol->content->above_or_below_surface (sB))
  head_pol->content->cut_by_surface (sB, 'A');
  }

else

  {
head_pol = head_pol->next_pol;
--Npol;};


return;
};


// The second procedure discards polygons with centers between the two
// surfaces. For all others, only the portions below sB or above sA are
// retained. sA has to be above sB throughout the modeling volume.


void ListPolygons :: cut_outside_two_surfaces  (Surface& sA, Surface& sB)
{
Node* current=head_pol;
int i, cA, cB;

while (current->next_pol)
  {
cA = sA.is_point_above_below (current->next_pol->content->center);
```

```
cB = sB.is_point_above_below (current->next_pol->content->center);

if (cA<0 && cB>0)                   // the center is between the surfaces

   {current->next_pol = current->next_pol->next_pol;
   -- Npol; }
else
  {
                                    // retain only portion  above sA
if (cA>0 && !current->next_pol->content->above_or_below_surface (sA))
  current->next_pol->content->cut_by_surface (sA, 'A');

                                    // OR retain only portion below sB
if (cB<0 && !current->next_pol->content->above_or_below_surface (sB))
  current->next_pol->content->cut_by_surface (sB, 'B');

current = current->next_pol;
  };
  };


cA = sA.is_point_above_below (head_pol->content->center);
cB = sB.is_point_above_below (head_pol->content->center);

if (cA<0 && cB>0)                   // the center is between the surfaces

   {head_pol = head_pol->next_pol;
   -- Npol; }
else
  {
                                    // retain only portion  above sA
if (cA>0 && !head_pol->content->above_or_below_surface (sA))
  head_pol->content->cut_by_surface (sA, 'A');

                                    // OR retain only portion below sB
if (cB<0 && !head_pol->content->above_or_below_surface (sB))
  head_pol->content->cut_by_surface (sB, 'B');
  };


return;

};


// Procedure to find the shortest distance from a polygon pol in 3d to a
member
// of a list of polygons.

double ListPolygons :: shortest_distance_from_polygon (Polygon& pol)
{
double distance = head_pol->content->distance_from_polygon (pol);
Node* current = head_pol->next_pol;
int i;
double temp;

for (i=1; i<Npol; ++i)
```

```
  {
temp = current->content->distance_from_polygon (pol);
if (temp < distance)
      distance = temp;
current = current->next_pol;
};

return distance;
};


// Procedure to split the main list of polygons into sub-networks. Each
// sub-network is stored in a list of polygons that is then added to the
// main list of lists of polygons. This latter is returned.

ListListPol ListPolygons :: split_into_networks()
{
ListListPol* Net=new ListListPol;
ListPolygons* Isolated_pol=new ListPolygons;
Node* current=head_pol;
Node* temp;
int i=0;
cout<<"Fracture considered: "<<endl;

do
  {
    cout<<i+1<<"/"<<Npol<<endl;
    ListPolygons* Sub_net=new ListPolygons;
    Sub_net->add_polygon(*current->content);
    temp=current->next_pol;
    remove_from_list(*current->content);
    current=temp;
    Node* current_sub;
    current_sub=Sub_net->head_pol;

    if(Npol)
    {
    do{
      Node* current_int;
      current_int=head_pol;
      do
        {
        if(current_int->content->if_polygon_intersects(*current_sub->content)
|| current_sub->content->if_polygon_intersects(*current_int->content))
          {
          Sub_net->add_polygon_tail(*current_int->content);
          temp=current_int->next_pol;
          if(current->content->name == current_int->content->name)
            current=temp;
          remove_from_list(*current_int->content);
          current_int=temp;
          }
        else
          current_int=current_int->next_pol;
        }
      while(current_int);
      current_sub=current_sub->next_pol;
    }
    while (current_sub);
    };
    if(Sub_net->Npol==1)
     Isolated_pol->add_polygon(*Sub_net->head_pol->content);
    else
      Net->add_listpol(*Sub_net);
  ++i;
  }
  while(current);

Net->add_listpol(*Isolated_pol);
return *Net;
};


// Procedure to compute the maximum extent of a sub-network in direction x
// (j=1), y (j=2) or z (j=3). It corresponds to the measures C8x developed
// by Dershowitz (1986). The maximum distance between fracture centers is
// considered.

double ListPolygons :: find_c8(int j)
{
Node* current;
int i;
double cmin, cmax, cdist=0.;

cmax=head_pol->content->find_max_coord(j);
cmin=cmax;

current=head_pol->next_pol;
for (i=0; i<Npol-1; ++i)
  {
  if(cmax < current->content->find_max_coord(j))
    cmax=current->content->find_max_coord(j);
  if(cmin > current->content->find_min_coord(j))
    cmin=current->content->find_min_coord(j);

  current=current->next_pol;
  };

cdist=cmax-cmin;

return cdist;
};

//NEWLY ADDED FUNCTION OCT. 8, 2000
//creates a list of lines of intersections for each network
// we need to create a list of intersection lines in the order that
// they are connected.
// MAKE SURE THE LINES ARE IN 3-D!!!!

ListLines& ListPolygons::make_list_of_intersections()
{
ListLines *intersection_list=new ListLines;
Node *p=head_pol;
Node *pr;
```

```
for(int i=0; i<Npol-1; i++)
      {

      pr=p->next_pol;
      for(int j=0; j<Npol-i-1; j++)
        {
          if (p->content->if_polygon_intersects(*(pr->content))==TRUE || pr-
>content->if_polygon_intersects(*(p->content))==TRUE)
            {
            cout<<"\n\npolygon 1:\n";
            cout<<*(p->content);
            cout<<"polygon 2:\n";
            cout<<*(pr->content);
            intersection_list->add_line((p->content-
>line_of_intersection_with_polygon(*(pr->content))));

            }

          pr=pr->next_pol;

        }

      p=p->next_pol;

      }
 cout<<"returning *intersection_list\n";
return *intersection_list;
};
```

main.C

```
// ********************************************************************
// *                          G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998     *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                     *
// ********************************************************************

#include <math.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)
#define max(a,b) ((a) > (b) ? (a) : (b))
#define min(a,b) ((a) < (b) ? (a) : (b))
double Xm, Ym;                       // Lateral boundaries of the volume
int Nsets;                           // No. of fracture sets
double A, B, C, D, E, F;            // Top surface of modeling volume
double mA, mB, mC, mD, mE, mF, mG, mH, mI, mJ;   // Cubic surface of the fold
double oAv, oBv, oCv, oDv;
double oAh, oBh, oCh, oDh;
double bX1, bX2, bY1, bY2, bZ1, bZ2;
double bFk1, bFk2, Fk, MaxPhi;
double TRatio;
double AngleMin, elongation;

double angleR, angleC, angleNew;
double ratioR, ratioC;

double MeanArea, MeanA;
double AT;
ofstream out;
char strike, dip;
double angleD, angleDip, ratioD;
double ratioMA, CA, gama;
double Zmin;

#define x1(a, b, f)  (-b+sqrt(f))/(2*a)
#define x2(a, b, f)  (-b-sqrt(f))/(2*a)


#include "polar.h"
#include "cartesian.h"
#include "point.h"
```

```
#include "line.h"
#include "surface.h"
#include "box.h"
#include "circle.h"
#include "volume.h"
#include "plane.h"
#include "polygon.h"
#include "listpol.h"
#include "listlistpol.h"
#include "listline.h"
#include "borehole.h"
#include "stat.h"

time_t time(time_t *tloc);
double Random01();
double Random0a(double);
double  RandomBC(double , double);
double exp_value (double);
int PoissonN (double, double);
Polar ran_uniform_orientation();
Polar uniform_max_phi_orient(double);
Polar constant_orientation();
Polar Fisher_orientation (double);
Polygon make_initial (Plane&, Volume&);
ListPolygons Poisson_lines_on_polygon (Polygon& , double);
ListLines Poisson_lines_on_pol (Polygon& , double);
int Np;
int sizeNpdf;
double maxR;
double Datum;
char folding;
int Nboreholes;
char is_box;
char sn;
char gF, gH, gV, cF, cH, cV;

main ()
{
srand(time(0) * getpid());
int i;

/*********************************************************************
*              INPUT OF MODELING VOLUME FROM FILE "INPUT"            *
*********************************************************************/
ifstream in ("INPUT");
if (!in)
 cout<<"cannot open file"<<endl;

in>>Xm>>Ym;                  // Extent of rectangular area
in>>Datum;                   // Datum (Z=0) is bottom surface of modeling volume
in>>A>>B>>C>>D>>E>>F;        // Top surface


// INPUT OF TYPE OF MODEL MARKING
in>>ratioMA>>maxR;            // Coefficients of the model plane, line,
                              // and marking processes
in>>CA>>gama;
in>>Nsets;                              // Number of fracture sets

in>>folding;

in>>gF>>gH>>gV;          // Graphical output: gF for 3D system; gH for
horizontal
                         // outcrop plane; gV for vertical outcrop plane
in>>cF>>cH>>cV;          // Coordinate output: cF for 3D system; cH for
horizontal
                         // traces; gV for vertical traces

in>>oAv>>oBv>>oCv>>oDv;                 // profile outcrop plane
in>>oAh>>oBh>>oCh>>oDh;                 // plan view outcrop plane

in>>Nboreholes;                     // Number of boreholes desired

in>>is_box;                //Has to be y or Y if a computing box is desired

in>>sn;                    // Has to be y or Y if the subdivision into
                           // sub-networks is desired

in>>Np;                    // Np is the number of points on the top surface
in>>MaxPhi;                // Maximum azimuth, if uniform orientation
in>>Fk;                    // Fisher constant, if Fisher orientation
in>>AngleMin;              // Minimum allowed angle for polygon shapes
in>>elongation;            // Maximum allowed elongation of polygons
in>>strike;                // Relationship to strike: C(oncentric), R(adial)
in>>angleR>>ratioR;        // maximum angle and ratio between strike of
                           // polygon and slope of surface
in>>angleC>>ratioC;        // maximum angle and ratio between strike of
                           // polygon and strike of surface
in>>angleNew;              // new maximum angle between strike of polygon
                           // and strike (or slope) of surface
in>>dip;
in>>angleD>>ratioD;
in>>angleDip;

in>>sizeNpdf;
int Nsize_int [sizeNpdf+1];
double sizeMax [sizeNpdf];

for (i=0; i<sizeNpdf; ++i)
  {Nsize_int[i] = 0;
  in>>sizeMax[i];};
Nsize_int[sizeNpdf] = 0;


Polar MeanP[Nsets];
double FracInt[Nsets];
char pdf[Nsets];
double MeanR[Nsets];
double Ca[Nsets];
double TRatio[Nsets];
int Nzones[Nsets];
int Nfaults[Nsets];
```

```
char ZoneType[Nsets];
int totnz;
int maxnz;


// INPUT OF PARAMETERS FOR INDIVIDUAL FRACTURE SETS

for (i=0; i<Nsets; ++i)
  {
in>>FracInt[i];                 // Fracture intensity: cum frac area / volume
in>>MeanP[i].theta>>MeanP[i].phi;     // Mean oorientation of the set
in>>pdf[i];                     // Spherical orientation PDF
in>>TRatio[i];                  // Translation (non-coplanarity)
in>>MeanR[i];                   // Mean radius
in>>ZoneType[i];                // Type of zone markiing (Fault or Box)
in>>Nzones[i];                  // Number of probability marking zones

  };
in.close();


/***************************************************************************
*                       CREATING THE MODELING VOLUME                       *
***************************************************************************/

Surface ground (A, B, C, D, E, F);          // Topographic surface
Volume rock (ground);                       // Modeling volume over area Xm, Ym

double VOLUME = ground.find_enclosed_volume (Xm, Ym);

Zmin=rock.corner_min_z();
double Rmax = sqrt(Xm*Xm + Ym*Ym + rock.Zmax*rock.Zmax);
double Zm;
cout<<"Number of fracture sets "<<Nsets<<endl;


// CREATING FOLD IF ANY

Cubic fold;
if (folding == 'y' || folding == 'Y')
  {
in.open ("FOLD");
in>>fold.A>>fold.B>>fold.C>>fold.D>>fold.E>>fold.F>>fold.G>>fold.H>>fold.I>>fo
ld.J;
                            // Input of the cubic surface of a fold
in.close();
  };


/**************************************************************************
*            INPUT OF FRACTURE ZONES (if any)  FROM FILE "ZONES"          *
*   Zones are defined according to either faults and distance             *
*                                 or to boxes                             *
**************************************************************************/

int n;
totnz=0;
```

```
maxnz=0;

for(n=0; n<Nsets; ++n)
  {
  totnz+=Nzones[n];

  if(maxnz && (Nzones[n]>maxnz))
    maxnz=Nzones[n];
  if(!maxnz)
    maxnz=Nzones[n];
  };
if(maxnz==0)
  maxnz=1;

ListPolygons faults[Nsets];
double Distances[Nsets][maxnz-1];
double Marks[Nsets][maxnz];
double FaultCut[Nsets];
Box Zone_Box[Nsets];
Point BoxCorner[8];
Box Comp_Box;
double xx, yy, zz;

int j, k;

if(totnz>0)
  {
  in.open("ZONES");
  if(!in)
    cout<<"cannot open file"<<endl;

  for(i=0; i<Nsets; ++i)
    {
    if(ZoneType[i]=='B' || ZoneType[i]=='b')
      {
      Point TmpCorn[8];
      for(j=0; j<Nzones[i]; ++j)
        in>>Marks[i][j];          // Mark probability: ratio of intensity
                                  // in the zone to the greatest
                                  // intensity of the fracture sets
      for(j=0; j<8; ++j)
        in>>TmpCorn[j].X>>TmpCorn[j].Y>>TmpCorn[j].Z;   // Corners of the box

Zone_Box[i]=Box(TmpCorn[0],TmpCorn[1],TmpCorn[2],TmpCorn[3],TmpCorn[4],TmpCorn
[5],TmpCorn[6],TmpCorn[7]);
      in>>FaultCut[i];            // Cutting probability
      in>>Nfaults[i];             // Number of faults
      if(Nfaults[i]>0)
        {
        for(j=0; j<Nfaults[i]; ++j)
          {
          Polygon pol;
          int N;
          in>>N;                        // Number of vertices defining a fault
          for(k=0; k<N; ++k)
            {
            in>>xx>>yy>>zz;             // Coordinates of fault vertices
```

```
             Point* P=new Point(xx,yy,zz);
             pol.add_point(*P);
             };
           Polygon* new_pol=new Polygon;
           *new_pol=pol;
           faults[i].add_polygon(*new_pol);  // Adding the fault to the list
           };
         };
       };
     if((ZoneType[i]=='F' || ZoneType[i]=='f') && Nzones[i]>0)
       {
       for(j=0; j<Nzones[i]-1; ++j)
         in>>Distances[i][j];          // Distances that define the zones
       for(j=0; j<Nzones[i]; ++j)
         in>>Marks[i][j];              // Mark probability: ratio of intensity
                                       // in the zone to the greatest
                                       // intensity of the fracture sets

       in>>FaultCut[i];                // Cutting probability
       in>>Nfaults[i];                 // Number of faults
       for(j=0; j<Nfaults[i]; ++j)
         {
         Polygon pol;
         int N;
         in>>N;                        // Number of vertices defining a fault
         for(k=0; k<N; ++k)
           {
           in>>xx>>yy>>zz;             // Coordinates of fault vertices
           Point* P=new Point(xx,yy,zz);
           pol.add_point(*P);
           };
         Polygon* new_pol=new Polygon;
         *new_pol=pol;
         faults[i].add_polygon(*new_pol);  // Adding the fault to the list
         };
       };
     };

//Creating the computing box, if any

if(is_box=='y' || is_box=='Y')
  {
  for(j=0; j<8; ++j)
    in>>BoxCorner[j].X>>BoxCorner[j].Y>>BoxCorner[j].Z;

Comp_Box=Box(BoxCorner[0],BoxCorner[1],BoxCorner[2],BoxCorner[3],BoxCorner[4],
BoxCorner[5],BoxCorner[6],BoxCorner[7]);
  };

  in.close();
  };

/*********************************************************************
*                     CREATING OUTCROP PLANES                        *
*********************************************************************/


// VERTICAL OUTCROP PLANE
```

```
Plane profile (oAv, oBv, oCv, oDv);
Polygon outPolV = make_initial(profile, rock);
outPolV.make_polygon_2d();
outPolV.sort_points_2d();
outPolV.make_polygon_3d();
Zm =  2*(rock.Zmax / sin(profile.dip));
outPolV.add_points_on_surface (profile, rock.top, Np, Zm);
Circle cirV=profile.circle_on_vertical();


// HORIZONTAL OUTCROP PLANE

Plane plan (oAh, oBh, oCh, oDh);
Polygon outPolH = make_initial(plan, rock);
outPolH.make_polygon_2d();
outPolH.sort_points_2d();
outPolH.make_polygon_3d();
Circle cirH=plan.circle_on_horizontal();


// 3D VIEW OF THE OUTCROP PLANES

out.open("OUTCROPS.m");
out<<"Show [ ";
outPolV.print();
out<<", ";
outPolH.print();
out<<"]  ";
out.close();



/**********************************************************************
*                        GENERATION OF FRACTURE SETS                  *
**********************************************************************/


ListPolygons FracSet[i];
ListPolygons FracSys;
Stat meanSD;
double P32;
out.open ("FRAC.txt");
out<<"MARKING RULE"<<endl;
out<<"Mark: > "<<ratioMA<<" and < "<<maxR<<"  Gama: "<<gama<<"  CA:
"<<CA<<endl;
out<<"angleR--ratioR--angleC--ratioC--strike--angleNew :"<<endl;
out<<angleR<<" "<<ratioR<<" "<<angleC<<" "<<ratioC<<" "<<strike<<"
"<<angleNew<<endl;
out.close();

for (i=0; i<Nsets; ++i)
  {
int counter = 1;
double Mu = FracInt[i]/gama;           // Intensity of Poisson plane network

double expD = exp_value(Mu);
double planeD = -Rmax+expD;
MeanA = PI*MeanR[i]*MeanR[i];
cout<<MeanR[i]<<" "<<MeanA<<endl;
MeanArea = MeanA/CA;
double intensity = sqrt(PI/(MeanArea));

cout<<"Fracture set "<<i+1<<endl;

out.open ("FRAC.txt", ios::app);
out<<endl<<"FRACTURE SET "<<i+1<<endl;
out<<"Expected fracture intensity : "<<FracInt[i]<<endl;
out<<"Plane intensity : "<<Mu<<endl;
out<<"Exp. radius-Exp. area : "<<MeanR[i]<<" "<<MeanA<<endl;
out<<"Line intensity : "<<intensity<<endl;
out.close();


// GENERATION OF THE FRACTURE PLANES OF A FRACTURE SET

while (planeD < Rmax)
 {
ListPolygons* FracPlane = new ListPolygons;
cout<<" set  "<<i+1<<" plane "<<counter<<endl;
Plane p (pdf[i], MeanP[i], planeD);
Polygon temp =  make_initial(p, rock);

if (temp.noP >2)
  {
Polygon* pol = new Polygon;
*pol = temp;

pol->make_polygon_2d();
pol->sort_points_2d();
pol->make_polygon_3d();

Zm =  2*(rock.Zmax / sin(p.dip));
pol->add_points_on_surface (p, rock.top, Np, Zm);
pol->area_radius_3d ();


// PLOT PRIMARY PLANE PROCESS
//out.open ("PRIMARY.m", ios::app);
//out<<"Show [ ";
//pol->print();
//out<<"]  "<<endl<<endl;;
//out.close();


pol->make_polygon_2d();
double InitialA = pol->area;


// TESSELLATION OF A FRACTURE PLANE INTO POLYGONS

*FracPlane = Poisson_lines_on_polygon (*pol, intensity);
cout<<"before mark "<<FracPlane->Npol<<endl;
```

```
// PLOT SECONDARY PROCESS: TESSELLATION
//FracPlane->make_listpol_3d();
//out.open ("TESSELLATION.m", ios::app);
//out<<"Show [ ";
//FracPlane->print();
//out<<"]  "<<endl;
//out.close();
//FracPlane->make_listpol_2d();


// MARKING OF POLYGONS WITH GOOD SHAPE: ~40% of the TOTAL AREA

FracPlane->mark_good_shape(AngleMin, elongation);
cout<<"after mark "<<FracPlane->Npol<<endl;

//FracPlane->make_listpol_3d();
//out.open ("MARKING.m", ios::app);
//out<<"Show [ ";
//FracPlane->print();
//out<<"]  "<<endl;
//out.close();
//FracPlane->make_listpol_2d();


// TRANSLATION

FracPlane->translate_2d (MeanR[i], TRatio[i]);


// ZONE MARKING OF POLYGONS FOR DIFFERENT ZONE FRACTURE INTENSITY

FracPlane->make_listpol_3d();

double Dist[maxnz-1];
double Mar[maxnz];

if ((ZoneType[i]=='B' || ZoneType[i]=='b') && FracPlane->Npol)
  {
  for(j=0; j<Nzones[i]; ++j)
    Mar[j]=Marks[i][j];
  FracPlane->mark_by_box(Zone_Box[i], Mar);
  cout<<"After marking by zones "<<FracPlane->Npol<<" fractures"<<endl;
  };

if ((ZoneType[i]=='F' || ZoneType[i]=='f') && FracPlane->Npol && Nzones[i] &&
Nfaults[i])
  {
  for(j=0; j<Nzones[i]-1; ++j)
    Dist[j]=Distances[i][j];
  for(j=0; j<Nzones[i]; ++j)
    Mar[j]=Marks[i][j];
  FracPlane->mark_by_zones(faults[i], Nzones[i], Dist, Mar);
  cout<<"After marking by zones "<<FracPlane->Npol<<" fractures"<<endl;
  };
```

```
 if (FracPlane->Npol)
    FracSet[i].add_listpol(*FracPlane);
 else
    delete FracPlane;
 cout<<"total "<<(FracSet[i].Npol+FracSys.Npol)<<endl;
  };

 expD = exp_value (Mu);
 planeD += expD;
 ++ counter;

   };                   // End of line tessellation for one plane

 // CUTTING OF FRACTURES BY FAULTS LIMITING THE SET'S ZONE

 if(totnz>0)
   {
   Node* cur_pol=faults[i].head_pol;
   cout<<"Cutting Set "<<i+1<<endl;
   for(j=0; j<Nfaults[i]; ++j)
     {
     Polar Ppole= cur_pol->content->Pole;
     Point Ppoint= *(cur_pol->content->head);
     Polar Pmean_pole(0., 0.);
     Plane Pl(Ppole, Pmean_pole, Ppoint);

     FracSet[i].cut_by_plane(Pl, 'F', FaultCut[i]);    //Keeps everything
that's in FRONT

     cur_pol= cur_pol->next_pol;
     };
   };

FracSys.add_listpol (FracSet[i]);

};                            // End of generation of one fracture set

// NAMING THE POLYGONS
// (name is their rank in the list)

FracSys.name_pol();

//DTERMINING THE PROPERTIES OF THE FRACTURE SYSTEM

double Radius = sqrt (MeanA/PI);
meanSD = FracSys.find_mean_sd(Radius, maxR);
P32 = meanSD.total / VOLUME;

// DETERMINING THE PROPERTIES OF THE FRACTURE SYSTEM INSIDE THE
// COMPUTING BOX (if any)

ListPolygons FracSys_Box;
double P32_Box=0.;
Stat meanSD_Box;

if(is_box=='y' || is_box=='Y')
  {
```

```
  Node* cur_pol=FracSys.head_pol;
  for(j=0; j<FracSys.Npol; ++j)
    {
    if(Comp_Box.is_point_inside(cur_pol->content->center))
      FracSys_Box.add_polygon(*(cur_pol->content));
    cur_pol=cur_pol->next_pol;
    };
  meanSD_Box=FracSys_Box.find_mean_sd();
  P32_Box=meanSD_Box.total/Comp_Box.volume;
  };

// OUTPUT OF PROPERTIES (File FRAC.txt)

out.open ("FRAC.txt", ios::app);
out<<endl<<"MODELLING VOLUME"<<endl;
out<<"Mean--SD--Total--N--P32 : "<<meanSD<<" "<<FracSys.Npol<<" "<<P32<<endl;
out<<"Mean/expected--SD/Mean : "<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean
<<endl;
out<<endl<<"COMPUTING BOX"<<endl;
out<<"Volume--Mean--SD--Total--N--P32 : "<<Comp_Box.volume<<" "<<meanSD_Box<<"
"<<FracSys_Box.Npol<<" "<<P32_Box<<endl<<endl;
cout<<"Finished calculating statistics "<<endl;
out.close ();


ListLines tracesH;
ListLines tracesV;
ListLines faultsH[Nsets];
ListLines faultsV[Nsets];

// GRAPHICAL OUTPUT BEFORE ROTATION

//out.open ("FRACTURES.m", ios::app);
//out<<"Show [ ";
//FracSys.print();
//out<<"]  "<<endl;
//out.close();

// VERTICAL OUTCROP in a file "PROFILE.m"
//tracesV = FracSys.traces_on_plane(profile);

//out.open ("PROFILE.m");
//out<<"Show [ ";
//outPolV.print();
//out<<",  ";
//tracesV.print();
//out<<"]  "<<endl<<endl;;
//out.close();

//HORIZONTAL OUTCROP in a file "PLAN.m"
//tracesH = FracSys.traces_on_plane(plan);

//out.open ("PLAN.m");
//out<<"Show [ ";
//outPolH.print();
//out<<",  ";
```

```
//tracesH.print();
//out<<"]  "<<endl<<endl;;
//out.close();


/********************************************************************
*     ROTATION OF FRACTURES TO BETTER FIT SURFACE STRIKE AND DIP    *
*******************************************************************/

// RADIAL AND CONCENTRIC FRACTURES WITH ONE OF THE ORIENTATIONS PREFERRED

if (folding == 'y' || folding == 'Y')
  {
if ( strike == 'r' || strike == 'c')
FracSys.mark_by_strike (fold, angleR, angleC, angleNew, strike, ratioR,
ratioC);

if (dip == 'd')
FracSys.mark_by_dip (fold, angleD, angleDip, ratioD);
  };

/********************************************************************
*                          SIZE  DISTRIBUTION
*******************************************************************/

out.open ("SIZE.txt");
out<<"Mean--SD--Total--N--P32 "<<meanSD<<" "<<FracSys.Npol<<" "<<P32<<endl;
out<<"Mean/expected mean--SD/Mean "<<meanSD.Mean/MeanA<<" "<<meanSD.SD /
meanSD.Mean<<endl<<endl;
FracSys.size_distribution (sizeNpdf, Nsize_int, sizeMax, meanSD.Mean);
for (j=0; j<sizeNpdf; ++j)
  out<<sizeMax[j]<<"  "<<Nsize_int[j]<<endl;
out<<Nsize_int[sizeNpdf]<<endl<<endl;;
out.close();



/********************************************************************
 *        INTERSECTION WITH BOREHOLE(S) IF DESIRED
*******************************************************************/

if (Nboreholes)
  {
in.open ("BOREHOLE");

for (i=0; i<Nboreholes; ++i)
  {
in>>bX1>>bY1>>bZ1>>bX2>>bY2>>bZ2;      // borehole

// INTERSECTIONS WITH A BOREHOLE
Point P1 (bX1, bY1, bZ1);
Point P2 (bX2, bY2, bZ2);

Line boreL (P1, P2);
Borehole BoreLine = FracSys.intersections_with_logline (boreL);
Stat boreMSD = BoreLine.find_mean_sd_spacing();
if (BoreLine.Nfrac)
  {
if (!i)
  {out.open ("BORE.txt");}
else
   out.open ("BORE.txt", ios::app);
out<<endl<<endl<<"Next borehole "<<endl;
out<<BoreLine;
out<<"SPACING mean--sd--boreline length "<<boreMSD<<endl;
out.close();
cout<<"Borehole calculated "<<endl;

out.open ("FRAC.txt", ios::app);
out<<"SPACING along borehole "<<i+1<<endl;
out<<"mean--sd--boreline length "<<boreMSD<<endl;
out.close();

if (!i)
  {out.open ("BORE.m");}
else
   out.open ("BORE.m", ios::app);
out<<"Show [ ";
BoreLine.print();
out<<"]  ";
out.close();

  };

  };
in.close();
  };

/******************************************************************
 *      SUBDIVISION OF FRACTURE SYSTEM INTO SUB-NETWORKS          *
 ******************************************************************/

if (sn == 'Y' || sn == 'y')
  {
ListPolygons FracSys_copy=FracSys.make_copy();

cout<<"Subdividing into networks..."<<endl;

ListListPol Networks;
Networks=FracSys_copy.split_into_networks();

Networks.name_list();

Node_list* cur_list=Networks.head_list;

cout<<"There are: "<<Networks.Nlist<<" sub-networks"<<endl;

// NEWLY ADDED SECTION OCT. 25, 2000
// CREATION OF THE LIST OF INTERSECTION LINES FOR EACH NETWORK
/*******************************************************/
```

```
 ListLines *intersection_lines[Networks.Nlist-1];// does not include isolated
fractures

Node_list *p_list=Networks.head_list->next_list;

for (int k=0; k<Networks.Nlist-1; k++)
  {
    intersection_lines[k]=&(p_list->content->make_list_of_intersections());
    p_list=p_list->next_list;
  }

/*************************************************/

// NEWLY ADDED SECTION OCT. 27, 2000
//OUTPUTS ALL THE LOI'S TO FILE IN MATHEMATICA FORMAT

out.open("LOI1.m");
 out<<"Show [ ";
for (int k=0; k<Networks.Nlist-1; k++)
  {
intersection_lines[k]->print();
 if (k<Networks.Nlist-2) out<<", ";// separator
  }
 out<<"]  ";
 out.close();

/**************************************************/
 // NEWLY ADDED SECTION JUNE 5, 2001
 // OUTPUTS ALL THE LOI LENGTHS INTO A FILE

 int line_count=0;
 Node_line *p_line=0;
 out.open("ALLINTLENGTHS.txt");
 out<<"LENGTH\t  Trend\t Plunge\t Vector.X\tVector.Y\tVector.Z"<<endl;
 for(int k=0; k<Networks.Nlist-1; k++)
   {
     line_count+=intersection_lines[k]->Nline;
     p_line=intersection_lines[k]->head_line;
     for(int kj=0; kj<intersection_lines[k]->Nline; kj++)
       {
       out<<p_line->content->length<<"  "<<p_line->content->compute_trend()<<"
"<<p_line->content->compute_plunge()<<"  "<<p_line->content->vector;
       p_line=p_line->next_line;
       }
   }
 out<<"there are "<<line_count<<" intersections"<<endl;
 out.close();
 cout<<"there are "<<line_count<<" intersections"<<endl<<endl;

/*******************************************************/

// COMPUTATION OF THE WEIGHTED AVERAGE ORIENTATION OF THE SUB-NETWORKS

Stat subnet_strike[Networks.Nlist];
Stat subnet_dip[Networks.Nlist];

cur_list=Networks.head_list;
```

```
for(i=0; i<Networks.Nlist; ++i)
  {
  subnet_strike[i]=cur_list->content->find_mean_sd_strike();
  subnet_dip[i]=cur_list->content->find_mean_sd_dip();

  cur_list=cur_list->next_list;
  };

// COMPUTATION OF C8x, C8y, C8z

cur_list=Networks.head_list;
double c8_x[Networks.Nlist];
double c8_y[Networks.Nlist];
double c8_z[Networks.Nlist];

for(i=0; i<Networks.Nlist; ++i)
  {
  c8_x[i]=cur_list->content->find_c8(1);
  c8_y[i]=cur_list->content->find_c8(2);
  c8_z[i]=cur_list->content->find_c8(3);

  cur_list=cur_list->next_list;
  };

// GRAPHICAL OUTPUT FILES FOR THE SUB-NETWORKS

cur_list=Networks.head_list->next_list;
int N_print=3;

int Loc_print[N_print];
int Size_print[N_print];

for(j=0; j<N_print; ++j)
  {
  Loc_print[j]=0;
  Size_print[j]=0;
  };

for(j=0; j<N_print; ++j)
  {
  cur_list=Networks.head_list->next_list;
  if(!j)
   {
    for(i=1; i<Networks.Nlist; ++i)
    {
    if(cur_list->content->Npol > Size_print[j])
      {
      Size_print[j]=cur_list->content->Npol;
      Loc_print[j]=i;
      };
    cur_list=cur_list->next_list;
    };
   }
  else
   {
    for(i=1; i<Networks.Nlist; ++i)
    {
     if((cur_list->content->Npol > Size_print[j]) && (cur_list->content->Npol <
 Size_print[j-1]))
       {
       Size_print[j]=cur_list->content->Npol;
       Loc_print[j]=i;
       };
     cur_list=cur_list->next_list;
     };
    };
   };

out.open("SUB_NET1.m");
cur_list=Networks.head_list->next_list;
for(i=1; i<Networks.Nlist; ++i)
  {
  if(Loc_print[0]==i)
    {
    out<<"Show [ ";
    cur_list->content->print();
    out<<"]  ";
    };
  cur_list=cur_list->next_list;
  };
out.close();

out.open("SUB_NET2.m");
cur_list=Networks.head_list->next_list;
for(i=1; i<Networks.Nlist; ++i)
  {
  if(Loc_print[1]==i)
    {
    out<<"Show [ ";
    cur_list->content->print();
    out<<"]  ";
    };
  cur_list=cur_list->next_list;
  };
out.close();

out.open("SUB_NET3.m");
cur_list=Networks.head_list->next_list;
for(i=1; i<Networks.Nlist; ++i)
  {
  if(Loc_print[2]==i)
    {
    out<<"Show [ ";
    cur_list->content->print();
    out<<"]  ";
    };
  cur_list=cur_list->next_list;
  };
out.close();

cur_list=Networks.head_list;
out.open ("ISOLATED.m");
out<<"Show [ ";
cur_list->content->print();
```

```
out<<"]  ";
out.close();

/*****************************************************************************
*/
 // NEWLY ADDED SECTION JUNE 5, 2001
 // OUTPUTS ALL THE LOI LENGTHS FOR ALL THE SUB-NETWORKS INTO 3 FILES

 if(Loc_print[0]>0 && Loc_print[0]<Networks.Nlist)
   {
 p_line=0;// re-initialize
out.open("SUB1_LOI.txt");
out<<"LENGTH\t Trend\t Plunge\t Vector.X\tVector.Y\tVector.Z"<<endl;
 p_line=intersection_lines[Loc_print[0]-1]->head_line;
for(int kj=0; kj<intersection_lines[Loc_print[0]-1]->Nline; kj++)
       {
       out<<p_line->content->length<<"  "<<p_line->content->compute_trend()<<"
"<<p_line->content->compute_plunge()<<"  "<<p_line->content->vector;
       p_line=p_line->next_line;
       }
 out<<"there are "<<intersection_lines[Loc_print[0]-1]->Nline<<" intersections
in sub-network 1"<<endl;
 out.close();
 cout<<"sub-network 1 is the number "<<Loc_print[0]+1<<" network"<<endl;
 cout<<"the corresponding intersection list is
intersection_lines["<<Loc_print[0]-1<<"]"<<endl;
 cout<<"there are "<<intersection_lines[Loc_print[0]-1]->Nline<<"
intersections in sub-network 1"<<endl<<endl;
   }

/*****/

 if(Loc_print[1]>0 && Loc_print[1]<Networks.Nlist)
   {
 p_line=0;// re-initialize
out.open("SUB2_LOI.txt");
out<<"LENGTH\t Trend\t Plunge\t Vector.X\tVector.Y\tVector.Z"<<endl;
 p_line=intersection_lines[Loc_print[1]-1]->head_line;
for(int kj=0; kj<intersection_lines[Loc_print[1]-1]->Nline; kj++)
       {
       out<<p_line->content->length<<"  "<<p_line->content->compute_trend()<<"
"<<p_line->content->compute_plunge()<<"  "<<p_line->content->vector;
       p_line=p_line->next_line;
       }
 out<<"there are "<<intersection_lines[Loc_print[1]-1]->Nline<<" intersections
in sub-network 2"<<endl;
 out.close();
 cout<<"sub-network 2 is the number "<<Loc_print[1]+1<<" network"<<endl;
 cout<<"the corresponding intersection list is
intersection_lines["<<Loc_print[1]-1<<"]"<<endl;
 cout<<"there are "<<intersection_lines[Loc_print[1]-1]->Nline<<"
intersections in sub-network 2"<<endl<<endl;
   }

/*****/

 if(Loc_print[2]>0 && Loc_print[2]<Networks.Nlist)
```

```
    {
  p_line=0;// re-initialize
 out.open("SUB3_LOI.txt");
 out<<"LENGTH\t Trend\t Plunge\t Vector.X\tVector.Y\tVector.Z"<<endl;
  p_line=intersection_lines[Loc_print[2]-1]->head_line;
 for(int kj=0; kj<intersection_lines[Loc_print[2]-1]->Nline; kj++)
        {
        out<<p_line->content->length<<"  "<<p_line->content->compute_trend()<<"
 "<<p_line->content->compute_plunge()<<"  "<<p_line->content->vector;
        p_line=p_line->next_line;
        }
  out<<"there are "<<intersection_lines[Loc_print[2]-1]->Nline<<" intersections
 in sub-network 3"<<endl;
  out.close();
  cout<<"sub-network 3 is the number "<<Loc_print[2]+1<<" network"<<endl;
  cout<<"the corresponding intersection list is
 intersection_lines["<<Loc_print[2]-1<<"]"<<endl;
  cout<<"there are "<<intersection_lines[Loc_print[2]-1]->Nline<<"
 intersections in sub-network 3"<<endl<<endl;
    }


/*******************************************************************************
*/
// NEWLY ADDED SECTION JUNE 5, 2001
// OUTPUTS ALL THE LOI'S FOR EACH SUB_NETWORK FOR MATHEMATICA

 if(Loc_print[0]>0 && Loc_print[0]<Networks.Nlist)
   {
     out.open("SUB1_LOI.m");
       out<<"Show[ ";
     intersection_lines[Loc_print[0]-1]->print();
     out<<"] ";
     out.close();
   }

if(Loc_print[1]>0 && Loc_print[1]<Networks.Nlist)
   {
     out.open("SUB2_LOI.m");
       out<<"Show[ ";
     intersection_lines[Loc_print[1]-1]->print();
     out<<"] ";
     out.close();
   }

if(Loc_print[2]>0 && Loc_print[2]<Networks.Nlist)
   {
     out.open("SUB3_LOI.m");
       out<<"Show[ ";
     intersection_lines[Loc_print[2]-1]->print();
     out<<"] ";
     out.close();
   }

/*******************************************************************************
*/
```

```
// PRINTING THE CHARACTERISTICS OF THE SUB-NETWORKS IN FRAC.txt
int Ntot=0;
out.open ("FRAC.txt", ios::app);
out<<endl<<"SUB-NETWORKS"<<endl;
out<<"Number of isolated fractures: "<<Networks.head_list->content-
>Npol<<endl;
out<<"For the "<<Networks.Nlist<<" sub-networks (N-strike-dip-c8x-c8y-c8z):
"<<endl;
for(i=0; i<Networks.Nlist; ++i)
  {
  out<<cur_list->content->Npol<<"  "<<subnet_strike[i].Mean*180/PI<<"
"<<subnet_dip[i].Mean*180/PI<<" "<<c8_x[i]<<" "<<c8_y[i]<<" "<<c8_z[i]<<endl;
  Ntot+=cur_list->content->Npol;
  cur_list=cur_list->next_list;
  };
out<<"Total fractures involved: "<<Ntot<<endl<<endl;
out.close();

  };

/*******************************************************************
*       OUTPUT FOR FRACTURE SYSTEM AND TRACES ON OUTCROP PLANES
*******************************************************************/

//ALL FRACTURES in a file "FRACTURES.m"
if (gF == 'Y' || gF == 'y' || cF == 'y' || cF == 'Y')
  {
if (gF == 'Y' || gF == 'y')
  {
out.open ("FRACTURES.m");
out<<"Show [ ";
for(i=0; i<Nsets; ++i)
  {
  if(faults[i].Npol)
    {
    faults[i].print();
    out<<", ";
    };
  };
FracSys.print();
out<<"]  ";
out.close();
  };
if (cF == 'Y' || cF == 'y')
  {
out.open ("FRACTURES.txt");
        out<<FracSys<<endl;
out.close();
  };
  };

// CENTER OUTPUT

out.open ("CENTER.txt");
out<<"X--Y--Z--area--strike--dip"<<endl;
FracSys.center_print();
out<<endl<<endl;
```

```
FracSys_Box.center_print();
out.close();

// VERTICAL OUTCROP in a file "PROFILE.m"

if (gV == 'Y' || gV == 'y' || cV == 'y' || cV == 'Y')
  {
tracesV = FracSys.traces_on_plane(profile);
cout<<"After changes there are "<<tracesV.Nline<<" profile traces "<<endl;

double n0V, n1V, n2V;
n0V=tracesV.count_traces_0(cirV);
n1V=tracesV.count_traces_1(cirV);
n2V=tracesV.count_traces_2(cirV);

for(i=0; i<Nsets; ++i)
   faultsV[i]=faults[i].traces_on_plane(profile);

Stat outVmsd = tracesV.find_mean_sd_length();
out.open ("FRAC.txt", ios::app);
out<<endl<<"OUTCROPS (traces lengths)"<<endl;
out<<"VERTICAL: Mean--SD--Total--P21 : "<<outVmsd<<"
"<<outVmsd.total/outPolV.area<<endl;
out<<"Window: R-N0--N1--N2 : "<<cirV.radius<<" "<<n0V<<" "<<n1V<<"
"<<n2V<<endl;
out.close();

if (gV == 'Y' || gV == 'y')
  {
out.open ("PROFILE.m");
out<<"Show [ ";
for(i=0; i<Nsets; ++i)
  {
  if(faultsV[i].Nline)
    {
    faultsV[i].print();
    out<<", ";
    };
  };
// outPolV.print();
// out<<",  ";
tracesV.print();
out<<"]  ";
out.close();
  };

if (cV == 'Y' || cV == 'y')
  {
out.open ("PROFILE.txt");
        out<<tracesV<<endl;
out.close();
  };
  };

//HORIZONTAL OUTCROP in a file "PLAN.m"

if (gH == 'Y' || gH == 'y' || cH == 'y' || cH == 'Y')
```

```
  {
tracesH = FracSys.traces_on_plane(plan);
cout<<"After changes there are "<<tracesH.Nline<<" plan traces"<<endl;

double n0H, n1H, n2H;
n0H=tracesH.count_traces_0(cirH);
n1H=tracesH.count_traces_1(cirH);
n2H=tracesH.count_traces_2(cirH);

for(i=0; i<Nsets; ++i)
   faultsH[i]=faults[i].traces_on_plane(plan);

Stat outHmsd = tracesH.find_mean_sd_length();
out.open ("FRAC.txt", ios::app);
out<<"HORIZONTAL: Mean--SD--Total--P21 : "<<outHmsd<<"
"<<outHmsd.total/outPolH.area<<endl;
out<<"Window: R-N0--N1--N2 : "<<cirH.radius<<" "<<n0H<<" "<<n1H<<"
"<<n2H<<endl;
out.close();

if (gH == 'Y' || gH == 'y')
  {
out.open ("PLAN.m");
out<<"Show [ ";
for(i=0; i<Nsets; ++i)
  {
  if(faultsH[i].Nline)
    {
    faultsH[i].print();
    out<<", ";
    };
  };
// outPolH.print();
// out<<",  ";
tracesH.print();
out<<"]  ";
out.close();
  };

if (cH == 'Y' || cH == 'y')
  {
out.open ("PLAN.txt");
        out<<tracesH<<endl;
out.close();
  };
  };

};
```

plane.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "polar.h"
#include "plane.h"
#include "cartesian.h"
#include "point.h"
#include "circle.h"
#include "volume.h"
extern double bFk1, bFk2, Fk, MaxPhi, Xm, Ym, Zmin;

Polar ran_uniform_orientation(void);
Polar uniform_max_phi_orient(double);
Polar constant_orientation(void);
Polar Fisher_orientation (double);
Polar bivariate_Fisher_orientation (double, double);

#define PI M_PI
#define HalfPI (M_PI/2)
#define TwoPI (M_PI*2)


ostream& operator<< (ostream& o, Plane& p)
 {
     o <<p.A<<" "<<p.B<<" "<<p.C<<" "<<p.D<<endl;
     o <<"rel polar "<<p.rel_polar<<" abs polar "<<p.abs_polar<<endl;
     o << "strike "<<p.strike<<" dip "<<p.dip<<endl;
     return o;
};


// Constructor for the plane when the A, B, C, and D in the equation
// Ax+By+Cz=D are given

Plane :: Plane (double a, double b, double c, double d) {

A=a; B=b; C=c; D=d;
MeanPole.theta=0.; MeanPole.phi=0.;
rel_cart.X=A; rel_cart.Y=B; rel_cart.Z=C;
abs_cart.X=A; abs_cart.Y=B; abs_cart.Z=C;
rel_polar = rel_cart.convert_to_polar();
abs_polar.theta = rel_polar.theta;
abs_polar.phi = rel_polar.phi;

strike = abs_polar.theta - HalfPI;
if (strike < -PI)
```

```
  strike += TwoPI;

dip = abs_polar.phi;
if (dip > HalfPI)
     {dip = PI-dip;
 if (strike > 0.)
    strike-=PI;
 else
    strike +=PI;
     };
};


// Constructor for the plane defined by three points p, q, r. The normal
// vector N is given by the cross product N=pqxpr.

Plane :: Plane (Point& p, Point& q, Point& r) {

A=(q.Y-p.Y)*(r.Z-p.Z)-(q.Z-p.Z)*(r.Y-p.Y);
B=(q.Z-p.Z)*(r.X-p.X)-(q.X-p.X)*(r.Z-p.Z);
C=(q.X-p.X)*(r.Y-p.Y)-(q.Y-p.Y)*(r.X-p.X);
D=A*p.X+B*p.Y+C*p.Z;

MeanPole.theta=0.; MeanPole.phi=0.;
rel_cart.X=A; rel_cart.Y=B; rel_cart.Z=C;
abs_cart.X=A; abs_cart.Y=B; abs_cart.Z=C;
rel_polar = rel_cart.convert_to_polar();
abs_polar.theta = rel_polar.theta;
abs_polar.phi = rel_polar.phi;

strike = abs_polar.theta - HalfPI;
if (strike < -PI)
  strike += TwoPI;

dip = abs_polar.phi;
if (dip > HalfPI)
     {dip = PI-dip;
 if (strike > 0.)
    strike-=PI;
 else
    strike +=PI;
     };
};

// Constructor for a plane from a normal vector v(A, B, C) and a point
// p(X,Y,Z) in the global f.o.r. The plane is defined by equation Ax+By+Cz=D
// where D=AX+BY+CZ. The plane belongs to a set with mean pole orientation
// given by the class Polar MeanP. The coordinates of the pole in the local
// frame of reference (Z=MeanPole) are calculated correspondingly.


Plane::Plane (Cartesian& v, Point& p, Polar& MeanP) {
D=v.X*p.X+v.Y*p.Y+v.Z*p.Z;
A=v.X; B=v.Y; C=v.Z;
MeanPole = MeanP;
abs_cart.X=A; abs_cart.Y=B; abs_cart.Z=C;
abs_polar = abs_cart.convert_to_polar();
```

```
rel_cart = abs_cart.local_coordinates(MeanPole);
rel_polar = rel_cart.convert_to_polar();

strike = abs_polar.theta - HalfPI;
if (strike < -PI)
  strike += TwoPI;

dip = abs_polar.phi;
if (dip > HalfPI)
     {dip = PI-dip;
 if (strike > 0.)
    strike-=PI;
 else
    strike +=PI;
     };
};


// Constructor to create a plane through a given point P(X,Y,Z) in space.
// The MeanPole (Z in relative f.o.r.) is given by first class Polar.
// The orientation of the plane in the RELATIVE f.o.r. is given by the
// second class Polar. The coordinates of the point are in ABSOLUTE f.o.r.

Plane::Plane (Polar& MeanP, Polar& planePole, Point& p3d)
{
MeanPole = MeanP;
rel_polar = planePole;
rel_cart =  rel_polar.convert_to_cartesian();
abs_cart = rel_cart.global_coordinates (MeanPole);
abs_polar = abs_cart.convert_to_polar();
A = abs_cart.X; B = abs_cart.Y; C = abs_cart.Z;
D=A*p3d.X+B*p3d.Y+C*p3d.Z;

strike = abs_polar.theta - HalfPI;
if (strike < -PI)
  strike += TwoPI;

dip = abs_polar.phi;
if (dip > HalfPI)
     {dip = PI-dip;
 if (strike > 0.)
    strike-=PI;
 else
    strike +=PI;
     };
};


// Procedure to create a plane with a given orientation through a given
// point in 3D. The relative polar orientation is generated in the LOCAL
// f.o.r. stochastically according to a PDF specified by the character option.
// The relative cartesian, and the absolute pole orientations are calculated
// next. The equation of the palne (A, B, C, D) is calculated so that the
// plane passes through the point p3d (X,Y,Z) given in the GLOBAL f.o.r.
// The Polar Mean gives the orientation of the local f.o.r. whihc is the mean
// pole orientation for the set of planes to which the new plane belongs.
```

```
Plane:: Plane (char option, Polar& MeanP, double Distance) {
switch (option)
  {
  case 'u':                          // uniform distribution on the hemisphere
rel_polar = ran_uniform_orientation();
  break;
  case 'p':                          // partial uniform  on the hemisphere
rel_polar =  uniform_max_phi_orient(MaxPhi);
  break;
  case 'c':                            // constant orientation = Mean
rel_polar =  constant_orientation();
  break;
  case 'f':                              // Fisher
rel_polar = Fisher_orientation (Fk);
  break;
  case 'b':                                // Bivariate Fisher
rel_polar = bivariate_Fisher_orientation (bFk1, bFk2);
  break;
     default:
     cout<<"Unknown distribution."<<endl;
     break;
  };

MeanPole = MeanP;
rel_cart =  rel_polar.convert_to_cartesian();
abs_cart = rel_cart.global_coordinates (MeanPole);
if (option == 'c')
    abs_polar = MeanPole;
else
    abs_polar = abs_cart.convert_to_polar();

if (abs_polar.phi > HalfPI)            // make pole always point upward
      {abs_polar.phi = PI - abs_polar.phi;
       abs_polar.theta -= PI;
       if (abs_polar.theta < -PI)
        abs_polar.theta += TwoPI; };



A = abs_cart.X; B = abs_cart.Y; C = abs_cart.Z;
D = Distance;

strike = abs_polar.theta - HalfPI;
if (strike < -PI)
  strike += TwoPI;

dip = abs_polar.phi;
if (dip > HalfPI)
     {dip = PI-dip;
 if (strike > 0.)
    strike-=PI;
 else
    strike +=PI;
     };
```

```
};


// Calculates a vector of unit length which is perpendicular to the pole of
the
// plane, lies in the plane and points upward along the dip of the plane.

Cartesian Plane :: find_binormal ()
{
Cartesian* bc = new Cartesian;

if (C>0.)
  {bc->X = cos(dip) * sin(abs_polar.theta+PI);
   bc->Y = cos(dip) * cos(abs_polar.theta+PI);
   bc->Z = sin(dip);};
if (C<0.)
  {bc->X = cos(dip)*sin(abs_polar.theta);
   bc->Y = cos(dip)*cos(abs_polar.theta);
   bc->Z = sin(dip);};
if (C==0.)
  {bc->Y=0.; bc->Y=0.; bc->Z=1.;};

return (*bc);
};


//Procedure to check if a point is "in front" or "behind" a plane. The point
//is located "in front" if it is on the side of the plane pointed at by the
//pole of this latter.

int Plane::is_point_front_behind(Point &P)

{
double check=A*P.X+B*P.Y+C*P.Z-D;

if(check>0.000001)             //Front
  return 1;
if(check<-0.000001)             //Behind
  return -1;
if(check>-0.000001 && check<0.000001)       //On the plane
  return 0;
};


// Procedure to create a circle on a horizontal outcrop.

Circle Plane::circle_on_horizontal()

{
Circle circ;
double size=0.9;
circ.center.X=0.;
circ.center.Y=0.;
circ.center.Z=D/C;

double r1=Xm*sqrt(1+A*A/(C*C));
double r2=Ym*sqrt(1+B*B/(C*C));
```

```
if(r1<r2)
  circ.radius=size*r1;
else
  circ.radius=size*r2;

circ.support.A=A;
circ.support.B=B;
circ.support.C=C;
circ.support.D=D;

return circ;
};


// Procedure to create a circle on a vertical outcrop.

Circle Plane::circle_on_vertical()

{
Circle circ;
double size=0.9;
double XV1, YV1, XV2, YV2;
Point P1, P2;

if(A<0.0001 && A>-0.0001)
  {
  P1.X=-Xm; P1.Y=D/B; P1.Z=0.;
  P2.X=Xm; P2.Y=D/B; P2.Z=0.;
  };

if(B<0.0001 && B>-0.0001)
  {
  P1.X=D/A; P1.Y=-Ym; P1.Z=0.;
  P2.X=D/A; P2.Y=Ym; P2.Z=0.;
  };

if((A!=0.) && (B!=0.))
  {
  XV1=Xm; YV1=(D-A*Xm)/B;
  if(YV1>Ym)
    {
    YV1=Ym; XV1=(D-B*Ym)/A;
    };
  if(YV1<-Ym)
    {
    YV1=-Ym; XV1=(D+B*Ym)/A;
    };
  P1.X=XV1; P1.Y=YV1; P1.Z=0.;

  XV2=-Xm; YV2=(D+A*Xm)/B;
  if(YV2>Ym)
    {
    YV2=Ym; XV2=(D-B*Ym)/A;
    };
  if(YV2<-Ym)
    {
    YV2=-Ym; XV2=(D+B*Ym)/A;
```

```
      };
    P2.X=XV2; P2.Y=YV2; P2.Z=0.;
    };

  circ.center.X=(P1.X+P2.X)/2;
  circ.center.Y=(P1.Y+P2.Y)/2;
  circ.center.Z=Zmin/2;

  double l_car;
  l_car=sqrt((P1.X-P2.X)*(P1.X-P2.X)+(P1.Y-P2.Y)*(P1.Y-P2.Y));
  circ.radius=size*l_car/2;
  if(Zmin<l_car)
    circ.radius=size*Zmin/2;

  circ.support.A=A;
  circ.support.B=B;
  circ.support.C=C;
  circ.support.D=D;

  return circ;
  };
```

point.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *           Don't use or modify without written permission        *
// *                     (contact einstein@mit.edu)                  *
// *******************************************************************

#include "point.h"

extern ofstream out;
extern double Datum;


void Point::print() {
  if (X >=-0.0001 && X<=0.0001)
   X=0.;
  if (Y >=-0.0001 && Y<=0.0001)
  Y=0.;
  if (Z >=-0.0001 && Z<=0.0001)
  Z=0.;

out<<"{"<<X<<","<<Y<<","<<Z+Datum<<"}";
};


// Function to compute the volume of the paralellipiped defined by four
// points. PT is the top of the pyramid, and P1 to P3 are located in order
// to satisfy the left-handed system. The computation performed is also
// called the triple-product. Volume is 1/6th of the triple-product.

double Point::triple_product(Point& P1, Point& P2, Point& P3)
{
double TP;

TP=((P1.X-X)*(P2.Y-Y)*(P3.Z-Z)+(P2.X-X)*(P3.Y-Y)*(P1.Z-Z)+(P3.X-X)*(P1.Y-
Y)*(P2.Z-Z)-(P3.X-X)*(P2.Y-Y)*(P1.Z-Z)-(P1.X-X)*(P3.Y-Y)*(P2.Z-Z)-(P2.X-
X)*(P1.Y-Y)*(P3.Z-Z))/6;

return TP;
};

// Function to return the coordinate Z of the object Point.

double Point::give_z()
{
return Z;
};
```

polar.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "polar.h"
#include "cartesian.h"


Cartesian Polar::convert_to_cartesian()
    {  Cartesian* c = new Cartesian ();
       c->X=sin(phi)*sin(theta);
       c->Y=sin(phi)*cos(theta);
       c->Z=cos(phi);
       return (*c);
};
```

polygon.C

```
// *******************************************************************
// *                            G E O F R A C                        *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "cartesian.h"
#include "point.h"
#include "polygon.h"
extern ofstream out;
extern double Xm, Ym;
extern double MeanArea;
extern double ratioMA;
extern Volume rock;
double  RandomBC (double, double);
Polygon make_initial(Plane&, Volume&);


// Procedure to print a polygon for graphics output. The function creates
// a string which can be plotted directly by running Mathematica.


void Polygon :: print()
{
Point* marker = head;
int i;
out<<"Graphics3D [Polygon [{";
for (i=0; i< (noP-1); i++)
  {
    marker->print(); out<<",";
    marker = marker->next;
   };
marker->print();
out << "}], PlotRange->All, Axes->True, Boxed->True]";
return;
};



// A constructor to create a "polygon" with one vertex coinciding with
// the point P given in the global f.o.r., and parallel to the plane p.
// The plane does not have to pass through the point.


Polygon :: Polygon (Plane& p) {
setPole = p.MeanPole;
Pole = p.abs_polar;
strike = p.strike;
dip = p.dip;
```

```
head = 0;
noP =0; name=0; name_list=0;
center.X = 0.;  center.Y = 0.;  center.Z = 0.;
area = 0.; radius = 0.;
};


// Procedure to find the coordinates of the center of a polygon.

void Polygon :: find_center ()

{
Point* current = head;
double xx=0.; double yy=0.; double zz=0.;
int i;

for (i=0; i<noP; ++i)
  {
xx+=current->X; yy+=current->Y; zz+=current->Z;
current = current->next;
};
xx /= noP; yy /= noP; zz /= noP;
center.X = xx; center.Y = yy; center.Z = zz;
return;
};


// Procedure to calculate the area of a 2d polygon (all Z coordinates of
// vertices must be the same!! The formula used is
// A = 1/2(XiYi+1 + ... - XiYi-1 - ...) for i = 0 ... noP-1

void Polygon::find_area_radius_2d() {
area =0.; radius = 0.;
Point* current = head;
int i;
double xx, yy;

for (i=0; i<noP; ++i)
  {
xx = current->X; yy = current->next->Y;
area += (xx*yy);
current = current->next;
};
for (i=0; i<noP; ++i)
  {
yy = current->Y; xx = current->next->X;
area -= (xx*yy);
current = current->next;
};
area /= 2;
if (area<0.)
       area*=(-1.);

radius = sqrt (area/PI);
return;
};
```

```
// Procedure to find the maximum coordinate of any of the vertices of
// the polygon in directions x (j=1), y (j=2) or z (j=3).

double Polygon :: find_max_coord (int j)
{
Point* current;
int i;
double cmax;

if(j==1)
  cmax=head->X;
if(j==2)
  cmax=head->Y;
if(j==3)
  cmax=head->Z;
if(j<1 || j>3)
  cmax=0.;
current=head->next;

for(i=0; i<noP-1; ++i)
  {
  if(j==1 && cmax < current->X)
    cmax=current->X;
  if(j==2 && cmax < current->Y)
    cmax=current->Y;
  if(j==3 && cmax < current->Z)
    cmax=current->Z;
  current = current->next;
  };

return cmax;
};

// Procedure to find the minimum coordinate of any of the vertices of
// the polygon in directions x (j=1), y (j=2) or z (j=3).

double Polygon :: find_min_coord (int j)
{
Point* current;
int i;
double cmin;

if(j==1)
  cmin=head->X;
if(j==2)
  cmin=head->Y;
if(j==3)
  cmin=head->Z;
if(j<1 || j>3)
  cmin=0.;
current=head->next;

for(i=0; i<noP-1; ++i)
  {
  if(j==1 && cmin > current->X)
    cmin=current->X;
```

```
 if(j==2 && cmin > current->Y)
   cmin=current->Y;
 if(j==3 && cmin > current->Z)
   cmin=current->Z;
 current = current->next;
 };

return cmin;
};


// Procedure to calculate the coordinates of a polygon in its own plane,
// i.e. the polygon becomes essentially two-D, all Z coordinates are the
// same.

void Polygon :: make_polygon_2d ()
{
Point* current = head; int i;
Cartesian temp;
for (i=0; i<noP; ++i)
{ temp.X = current->X; temp.Y = current->Y; temp.Z = current->Z;
temp = temp.local_coordinates (Pole);
Point p(temp.X, temp.Y, temp.Z);
current->X = p.X; current->Y = p.Y; current->Z = p.Z;
current = current->next; };
find_center ();
if (!area && !radius && noP>2)
 {
sort_points_2d();
find_area_radius_2d();
 };
return;
};



// Procedure to sort the vertices in a 2D polygon in order: according to
// the angle, defined by a line through a vertex and the center, and the
// local axis X. All Z coordinates are the same, no need to calculate them.

void Polygon :: sort_points_2d ()
{
Point* current = head; Point* nextP;
int i, j; double xxI, yyI, xxJ, yyJ, xx, yy, zz, angleI, angleJ;
for (i=1; i < noP; ++i)
 {   nextP = current->next;
     for (j=i; j < noP; ++j)
      {
xxI = current->X - center.X; yyI = current->Y -  center.Y;
angleI = acos (xxI / sqrt(xxI*xxI + yyI*yyI));
          if (yyI<0.)
             angleI = TwoPI - angleI;
xxJ = nextP->X - center.X; yyJ = nextP->Y -  center.Y;
angleJ = acos (xxJ / sqrt(xxJ*xxJ + yyJ*yyJ));
          if (yyJ<0.)
             angleJ = TwoPI - angleJ;
      if (angleI > angleJ)
            {
xx = current->X; yy = current->Y; zz = current->Z;
current->X = nextP->X; current->Y = nextP->Y; current->Z = nextP->Z;
nextP->X = xx; nextP->Y = yy; nextP->Z = zz;
          };
       nextP = nextP->next;
      };
      current = current->next;
};
return;
};



// Procedure to calculate the coordinates of a 2d polygon from its local
// f.o.r. back into 3d in the global f.o.r.

void Polygon :: make_polygon_3d () {
Point* current = head;
int i;
Cartesian temp;
for (i=0; i < noP; ++i)
{ temp.X= current->X; temp.Y= current->Y; temp.Z= current->Z;
temp = temp.global_coordinates (Pole);
Point p(temp.X, temp.Y, temp.Z);
current->X = p.X; current->Y = p.Y; current->Z = p.Z;
current = current->next; };

find_center ();
return;
};

// Procedure to find the area and the equivalent radius of a 3d polygon.
// The coordinates of the vertices are transformed into 2d f.o.r. The
// vertices are sorted in order. Then the area of the 2d polygon is
// calculated and the radius is found as the radius of a circle with the
// same area. The coordinates of the vertices are then back calculated in 3d.

void Polygon :: area_radius_3d () {
make_polygon_2d();
sort_points_2d ();
find_area_radius_2d ();
//cout<<" 2D POLYGON "<<*this<<endl;
make_polygon_3d();
//cout<<"POLYGON "<<*this<<endl;

return;
};


// Function to find out if a point belongs to a polygon. Returns TRUE if the
// point has the same coordinates as one of the vertices of the polygon.

boolian Polygon :: is_it_member (Point& P)
{
Point* marker = head;
```

```
int i;
for (i=0; i<noP; i++)
  {
if (marker->X == P.X && marker->Y == P.Y && marker->Z == P.Z)
       return TRUE;
   marker = marker->next;
};
      return FALSE;
};


// Function to add a point to a polygon. The new point is appended
// at the head  of the list of vertices (Points) which consitute the
// polygon. The last point points to the new point which is the new
// head point.

void Polygon :: add_point (Point& P)
{
Point* temp = new Point;
++noP;
temp->X = P.X;
temp->Y = P.Y;
temp->Z = P.Z;

temp->next = head;
head = temp;
Point* current = head;
int i;
for (i=1; i < noP; ++i)
       current = current->next;
current->next = head;


if (noP==3 && !Pole.theta && !Pole.phi)
  {
double a1 = head->next->X - head->X;
double b1 = head->next->Y - head->Y;
double c1 = head->next->Z - head->Z;
double a2 = head->next->next->X - head->X;
double b2 = head->next->next->Y - head->Y;
double c2 = head->next->next->Z - head->Z;

Cartesian N ((b1*c2-b2*c1), (a2*c1-c2*a1), (a1*b2-b1*a2));
Pole = N.convert_to_polar();
  };
return;
};


// Procedure to find out if the shape of the polygon is good.
// For every vertex,  two values are calculated: 1) the angle between the
// two adjacent sides is calculated and compared to a specified minimum
// allowed angle MinAngle; and 2) the ratio of the distance to the center
// versus the equivalent radius of the polygon is compared to a specified
// maximum allowed elongation.
```

```
boolian Polygon :: if_good_shape (double MinAngle, double MaxE) {
if (noP<4)
  return FALSE;

if (ratioMA)
  {
if (area < ratioMA*MeanArea)
return FALSE;
  };

double el = find_elongation ();
 if (el/(2*radius) > MaxE)
  return FALSE;
int i;
double ANGLE;
Point* current = head;
Point* nextP = current->next;
Point* thirdP = current->next->next;
for (i=0; i<noP; ++i)
  {
Line l1 (*nextP, *current);
Line l2 (*nextP, *thirdP);
ANGLE = l1.angle_with_line(l2);

if (ANGLE < MinAngle)
    return FALSE;
current = current->next;
nextP = nextP->next;
thirdP = thirdP->next;
  };
return TRUE;
};
```

```
// Procedure to find the elongation of a polygon, defined as the maximum
// distance between two vertices in the polygon.


double Polygon :: find_elongation () {

Point* current=head;
Point* nextP;
int i=0, j=2;
double elongation = 0.;
for (i=0; i<noP-2; i++)
  {
nextP = current->next->next;
for (j=i+2; j<noP; j++)
  {
Line l(*current, *nextP);
if (l.length > elongation)
 elongation = l.length;
nextP = nextP->next;
  };
current=current->next;
```

```
  };
return elongation;
};



// Procedure to find the radius R of the smallest sphere in which a polygon
// can be inscribed. This is assumed to be the largest of the distances
// between the center and one of the vertices.

double Polygon :: minR_inscribe () {
Point* current = head;
double R=0., temp, dx, dy, dz;
int i;

for (i=0; i<noP; ++i)
  {

dx=current->X - center.X;
dy=current->Y - center.Y;
dz=current->Z - center.Z;

temp = sqrt(dx*dx + dy*dy + dz*dz);
if (temp > R)
  R=temp;

current=current->next;
  };
return R;
};



// Two functions for translation of polygons in 3D

void Polygon :: operator+ (Cartesian& vector)
{
Point* current = head;
int i;

for (i=0; i<noP; i++)
  { *current + vector;
  current = current->next; };
center+vector;
return;
};


void Polygon :: operator- (Cartesian& vector)
{
Point* current = head;
int i;

for (i=0; i<noP; i++)
  { *current - vector;
  current = current->next; };
center-vector;
```

```
return;
};


// Function to translate a polygon perpendicular to its plane.
// It is written in such a way that bigger polygons ar eless likely
// to be translated far form their original plane. The polygon has to be 2D!!
// Translation is accomplished by changing the Z coordinate.

void Polygon :: translate_2d (double ratio, double MeanR)
{
if (ratio)
  {
double maxDZ = ratio*MeanR*MeanR/radius;

double translation = RandomBC(-maxDZ, maxDZ);
Point* current= head;
int i;
for (i=0; i<noP; ++i)
  {
(current->Z) += translation;
current=current->next;
};

center.Z += translation;
  };
return;

};


// Procedure to find the distance from a point P to a polygon. The
calculations
// are in the FRAME OF REFERENCE of the polygon. The orthogonal progection
// P1 of the point is found and if it is within the polygon, the distance PP1
// is the shortest distance. If not, the point of interseciton P2 of the
// line between P1 and the center of the polygon is found.PP2 is then the
// shortest distance which is returned by the function. The 3d polygon and
// point MUST FIRST BE TRANSFORMED INTO LOCAL COORDINATES OF THE POLYGON.


double Polygon :: distance_from_point (Point& P)
{
Point P1 (P.X, P.Y, center.Z);
double distance;
Point* current = head;
int i;

Line l1 (center, P1);
for (i=0; i<noP; ++i)
  {
Line l2(*current, *current->next);
if (l1.intersect (l2))
  {Point P2 = l1.intersection_with_line (l2);
distance = sqrt((P.X-P2.X)*(P.X-P2.X)+(P.Y-P2.Y)*(P.Y-P2.Y)+(P.Z-P2.Z)*(P.Z-
P2.Z));
return distance;}
else
  current = current->next;
  };

distance = P.Z - center.Z;
if (distance<0.)
     distance *= -1.;
return distance;

};


// Procedure to find the shortest distance from a polygon pol (usually a new
// fracture) to another polygon (usually a major feature such as fault).
// The shortest distance from a verex of pol to the major polygon is
// calculated using the function above. The initial and final coordinates
// are in teh global f.o.r.


double Polygon :: distance_from_polygon (Polygon& pol)
{
int LINE = 0;
Line L;
Cartesian N (Pole);
Plane p (N, *head, setPole);

if (pol.if_intersects_plane (p))
{ L = pol.intersection_with_plane (p);
  LINE = 1;}

Polar POLE = pol.Pole;
pol.Pole = Pole;
make_polygon_2d ();
pol.make_polygon_2d();

double distance = distance_from_point(*pol.head);

if (LINE)
  {L.local_coordinates (Pole);
  int HOW = how_line_intersects (L);
  if (HOW)
     distance = 0.;
  else {
     double distance1 = distance_from_point(L.end1);
     double distance2 = distance_from_point(L.end2);
     if (distance1 < distance)
            distance = distance1;
     if (distance2 < distance)
            distance = distance2;
  }; };


if (distance)
  {
```

```
int i;
double  temp;
Point* current = pol.head->next;

for (i=1; i<pol.noP; ++i)
  {
temp = distance_from_point (*current);
if (temp < distance);
      distance = temp;
current = current->next;
  }; };


make_polygon_3d();
pol.make_polygon_3d();
pol.Pole = POLE;


return distance;

};


//Procedure to find if the argument polygon is in front of or behind the
//current object (the fault). It is considered "in front of" if the
//dot product between: 1) the vector connecting the center of the fault to
//center of the polygon, and 2) the pole of the fault, is positive.

boolian Polygon::if_front_of_polygon (Polygon& p)
{
Cartesian faultpole(Pole);
Point vector(p.center);

vector-center;

double prod=faultpole*vector;

if(prod > 0.)
  return TRUE;
else
  return FALSE;
};

// Function to create a test polygon in the plane Z=0. Four points are
randomly
// selected, one in each quadrant.

void Polygon::create_test_polygon()
{
Point P1, P2, P3, P4;
double d1, d2, d3, d4, a1, a2, a3, a4;
double d_l, d_h, a_l, a_h;

d_l=0.7*Xm;
if(Ym<Xm)
  d_l=0.7*Ym;
d_h=1.4*d_l;
```

```
a_l=0.5236;
a_h=1.0472;

d1=RandomBC(d_l, d_h);
d2=RandomBC(d_l, d_h);
d3=RandomBC(d_l, d_h);
d4=RandomBC(d_l, d_h);

a1=RandomBC(a_l, a_h);
a2=RandomBC(a_l, a_h);
a3=RandomBC(a_l, a_h);
a4=RandomBC(a_l, a_h);

P1.X=d1*cos(a1); P1.Y=d1*sin(a1); P1.Z=0.;
P2.X=d2*cos(a2); P2.Y=-d2*sin(a2); P2.Z=0.;
P3.X=-d3*cos(a3); P3.Y=-d3*sin(a3); P3.Z=0.;
P4.X=-d4*cos(a4); P4.Y=d4*sin(a4); P4.Z=0.;

add_point(P1); add_point(P2); add_point(P3); add_point(P4);

find_center();
find_area_radius_2d();

Point* current=head;
int i;
for(i=0; i<noP; ++i)
  {
  current->X = current->X - center.X;
  current->Y = current->Y - center.Y;
  current = current->next;
  };

find_center();
return;
};

// NEWLY ADDED FUNCTION. AUG 18, 2000
// returns the line of intersection between two polygons.  use only after
determining that
// the two polygons intersect.  note that the case for non-intersecting
polygons is not
// handled in this function.

// function that finds the length of intersection with another
// polygon.  use if_polygon_intersects(Polygon&) function first
// to determine if the polygons intersect.
// the polygons are in 2D!!!

// VERY IMPORTANT REMINDER!!!
// MAKE SURE THAT THE OBJECTS ARE IN THE SAME FRAME OF REFERENCE
// BEFORE CALCULATING INTERSECTION LENGTHS!!!
Line& Polygon::line_of_intersection_with_polygon(Polygon& pol)
{
  //make_polygon_2d();
  //sort_points_2d();
  //make_polygon_3d();
  //pol.make_polygon_2d();
  //pol.sort_points_2d();
  //pol.make_polygon_3d();

// this case is for two line segment intersections that have no
// common points

  cout<<"in line_of_intersection_with_polygon()\n";

Plane Plane1(*head,*(head->next),*(head->next->next));// 3D Plane object
Plane Plane2(*(pol.head),*(pol.head->next),*(pol.head->next->next));// 3D
Plane

//Polygon poly1, poly2;
//poly1=make_initial(Plane1, rock);// "rock" is the name of the modeling
//poly2=make_initial(Plane2, rock);// volume.  both in 3D

Line line1, line2;
//line1=poly1.intersection_with_plane(Plane2);
//line2=poly2.intersection_with_plane(Plane1);

line1=intersection_with_plane(Plane2);
line2=pol.intersection_with_plane(Plane1);

// sorting of the points according to position with respect to
// a chosen reference point.
Cartesian dir1, factor1, factor2;
// the Point end1 of line1 is chosen as the reference point.

// line1: vector pointing from end1 to end2
dir1.X=line1.end2.X-line1.end1.X;
dir1.Y=line1.end2.Y-line1.end1.Y;
dir1.Z=line1.end2.Z-line1.end1.Z;

// vector pointing from line1.end1 to line2.end1
factor1.X=line2.end1.X-line1.end1.X;
factor1.Y=line2.end1.Y-line1.end1.Y;
factor1.Z=line2.end1.Z-line1.end1.Z;

// vector pointing from line1.end1 to line2.end2
factor2.X=line2.end2.X-line1.end1.X;
factor2.Y=line2.end2.Y-line1.end1.Y;
factor2.Z=line2.end2.Z-line1.end1.Z;

// now use the overloaded operator / to sort the points according
// to their position with respect to the reference point.

factor1/=dir1;// est. position of line2.end1
factor2/=dir1;// est. position of line2.end2

Line *loi=new Line;// to be returned

//double scalar1, scalar2;
//scalar1=(sqrt((factor1.X)^2+(factor1.Y)^2+(factor1.Z)^2))/3;
//scalar2=(sqrt((factor2.X)^2+(factor2.Y)^2+(factor2.Z)^2))/3;

// if all the components of a resulting Cartesian are +, then
// it points the same direction as dir1.

// make a switch for the four possible cases
// case 1: factor1 and factor2 are both +
// case 2: factor1 and factor2 are both - cannot happen
// because only intersecting fractures are processed here
// case 3: factor1 is + and factor2 is -
// case 4: factor1 is - and factor2 is +

if (factor1.X>=0 && factor1.Y>=0 && factor1.Z>=0 && factor2.X>=0 &&
    factor2.Y>=0 && factor2.Z>=0)
{
Point Side_One[2];
Side_One[0]=line2.end1;
Side_One[1]=line2.end2;
Point farthest_point=line1.end2;
double max_distance_to_point=line1.length, distance=0;
int i=0;
for (i=0;i<2;i++)
    {
    distance=sqrt((Side_One[i].X-line1.end1.X)*(Side_One[i].X-
line1.end1.X)+(Side_One[i].Y-
    line1.end1.Y)*(Side_One[i].Y-
    line1.end1.Y)+(Side_One[i].Z-line1.end1.Z)*(Side_One[i].Z-line1.end1.Z));

    if (distance>max_distance_to_point)
        {
        max_distance_to_point=distance;
        farthest_point=Side_One[i];
        }
    }
if (farthest_point==line1.end2)
    {
    *loi=line2;
    cout<<line2;
    return (*loi);
    }
if (farthest_point==Side_One[0])
    {
    Line line3(line1.end2,line2.end2);
    *loi=line3;
    cout<<line3;
    return (*loi);
    }
if (farthest_point==Side_One[1])
  {
    Line line4(line1.end2,line2.end1);
    *loi=line4;
    cout<<line4;
    return (*loi);
  }
}

if (factor1.X>0 && factor1.Y>0 && factor1.Z>0 && factor2.X<0 &&
    factor2.Y<0 && factor2.Z<0)
{
double distance1=0;
distance1=sqrt((line2.end1.X-line1.end1.X)*(line2.end1.X-
line1.end1.X)+(line2.end1.Y-
    line1.end1.Y)*(line2.end1.Y-
    line1.end1.Y)+(line2.end1.Z-line1.end1.Z)*(line2.end1.Z-line1.end1.Z));
if (distance1>=line1.length)
    {
    *loi=line1;
    cout<<line1;
    return (*loi);
    }
if (distance1<line1.length)
  {
    Line line5(line1.end1, line2.end1);
    *loi=line5;
    cout<<line5;
    return (*loi);
  }
}

// actually this case needs no 'if' statement!!
if (factor1.X<0 && factor1.Y<0 && factor1.Z<0 && factor2.X>0 &&
    factor2.Y>0 && factor2.Z>0)
{
double distance2=0;
distance2=sqrt((line2.end2.X-line1.end1.X)*(line2.end2.X-
line1.end1.X)+(line2.end2.Y-
    line1.end1.Y)*(line2.end2.Y-
    line1.end1.Y)+(line2.end2.Z-line1.end1.Z)*(line2.end2.Z-line1.end1.Z));
if (distance2>=line1.length)
    {
    *loi=line1;
    cout<<line1;
    return (*loi);
    }
if (distance2<line1.length)
  {
    Line line6(line1.end1, line2.end2);
    *loi=line6;
    cout<<line6;
    return (*loi);
  }
}
};
```

random.C

```
// *******************************************************************
// *                             G E O F R A C                         *
// *    Copyright Massachusetts Institute of Technology 1995-1998     *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein          *
// *                                                                  *
// *          Don't use or modify without written permission          *
// *                   (contact einstein@mit.edu)                     *
// *******************************************************************

#include <math.h>
#include <iostream.h>
#include <stdlib.h>

#include "polar.h"
class Polar;

int rand(void);

// Procedure to generate a random number between 0 and 1

double Random01 ()
     {
     double  random_number =  (float) rand() / (float) 32767;
     return  random_number;
 };


// Procedure to generate a random number between 0 and a specified value

double Random0a (double a)
     {
       double random_number = a * (float) rand() / (float) 32767;
        return  random_number;
 };


// Procedure to generate a random number between two numbers b and c

double  RandomBC (double  b, double c)
      {
        if (c<b)
          {float temp=c;
           c=b;
           b=temp;}
      double  random_number = b+(c-b) * (float) rand() / (float) 32767;
     return random_number;
 };


// Procedure to generate an orientation in spherical coordinates (phi,theta)
// according to a uniform distribution on a unit hemisphere

Polar ran_uniform_orientation(void)
```

```
    {
    double fromX = RandomBC (-PI,PI);
    double fromZ = Random0a (HalfPI);
    Polar orientation (fromX, fromZ);
return orientation;
   };


// Procedure to generate an orientation in spherical coordinates (phi,theta)
// according to a uniform distribution on a unit hemisphere where theta
// varies between -PI and PI, and phi between 0 and PhiMax < HalfPI

Polar uniform_max_phi_orient(double PhiM)
   {
    double fromX = RandomBC (-PI, PI);
    double fromZ = Random0a (PhiM);
    Polar orientation (fromX, fromZ);
    return orientation;
   };

// Procedure to generate constant orientation equal to the mean orientation
// in spherical coordinates (MeanTheta, MeanPhi)

Polar constant_orientation(void)
    {
Polar orientation;
     return orientation;
    };


// Procedure to generate an orientation according to a univariate Fisher
// distribution on a unit hemisphere

Polar Fisher_orientation (double k)
    {double fromX = RandomBC (-PI, PI);
     double fromZ = acos(1/k*log(exp(k)-(Random01())*(exp(k)-1)));
     Polar orientation (fromX, fromZ);
     return orientation;
    };


// Procedure to generate an orientation according to a bivariate Fisher
// distribution on a unit hemisphere

Polar bivariate_Fisher_orientation (double k1, double k2)
     {double fromX = RandomBC (-PI, PI);
      double K = k1*sin(fromX)*sin(fromX) + k2*cos(fromX)*cos(fromX);
      double fromZ = acos(1/K*log(exp(K)-(Random01())*(exp(K)-1)));
      Polar orientation (fromX, fromZ);
  return orientation;
     };


// Procedure to generate a value (distance, area or other continuous
// variable) according to an exponential distribution with density lambda.
// A  random number is generated uniformly between 0 and 1, and then
// the cumulative exponential distribution is used to back-calculate
// an exponentially distributed value.
// exp. pdf(rv)=lambda*exp(-lambda*rv); cum. PDF(rv)=1-exp(-lambda*rv);

double exp_value (double lambda)
{
double y = Random01 ();
double RV = -(log(1-y))/lambda;
return RV;
};


// Procedure to calculate a random Poisson number with expected value
// N=lambda*A, where lambda is the density of the process (mean occurrence
// per unit area (or per unit distance, etc.), and A is the total area (or
// distance, etc.). This procedure is used mainly to calculate the random
// number of lines which will produce a line tessellation of intensity
// lambda over a polygon with area A.

int PoissonN (double lambda, double Area)
{
int PN = 0;
double sumA = 0.;
do {
PN++;
sumA += exp_value (lambda);
}
while (sumA < Area);
return (PN-1);
};
```

rotation.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                  (contact einstein@mit.edu)                     *
// *******************************************************************

#include "cartesian.h"
#include "point.h"
#include "polygon.h"
#include "listpol.h"
extern ofstream out;
extern double Xm, Ym;
extern double MeanArea;
double Random01 ();


double  RandomBC (double, double);


// Procedure to mark the polygon so that it is subparallel to
// the strike of a surface at its center. The strike of a surface
// is calculated with function of class Surface. A ration Ps of polygons
// is marked as fractures if the angle between the strike of the polygon
// and the strike of the surface does not exceed the specified angle. Angle
// between the strike is calculated using the dot product between two
// unit vectors: cos angle = a1*a2 + b1*b2 + c1*c2.

boolian Polygon :: mark_parallel_to_strike (double Sstrike, double angle,
double Ps)

{

double cosD = cos(Sstrike)*cos(strike)+sin(Sstrike)*sin(strike);
if ((cosD >= cos(angle) && cosD <=1 || cosD<=-cos(angle) && cosD>=-1.) &&
Random01 () < Ps)
  return TRUE;
else
 return FALSE;
};


// Procedure to mark the polygon so that it is sub orthogonal
// to the strike of a surface at its center. The strike of a surface
// is calculated with function of class Surface. A ration Po of polygons
// is marked as fractures if the angle between the strike of the polygon and
//  the strike of the surface deviates from PI/2 not more that a specified
// angle. Angle between the strike is calculated using the dot product
// between two unit vectors: cos angle = a1*a2 + b1*b2 + c1*c2.
```

```
boolian Polygon :: mark_orthogonal_to_strike (double Sstrike, double angle,
double Po)

{
double cosD = cos(Sstrike)*cos(strike)+sin(Sstrike)*sin(strike);
if ( cosD >= -sin(angle) && cosD <= sin(angle) && Random01() < Po)
  return TRUE;
else
 return FALSE;
};


// Procedure to mark the polygon accoding to its dip compared to the dip of
// a surface. Both dips are between zero and PI/2. Return True if the dip
// difference is smaller than teh specified angle.

boolian Polygon :: mark_by_dip (double Sdip, double angle, double P)

{
double cosD = cos(Sdip)*cos(dip)+sin(Sdip)*sin(dip);
if ( cosD >= -sin(angle) && cosD <= sin(angle) && Random01() < P)
  return TRUE;
else
 return FALSE;

};



// Procedure to rotate a polygon. The arguments are the new latitude and
// azimuth (theta and phi) of the polygon pole in the absolute frame of
// reference.

void Polygon :: rotate_by_strike (double NewStrike)
{
Cartesian V(center.X, center.Y, center.Z);
(*this)-V;
make_polygon_2d();

if (RandomBC(-1., 1.)>= 0.)
    {
Pole.theta = NewStrike + HalfPI;
if (Pole.theta > PI)
      Pole.theta -= TwoPI;
    }
else
{
Pole.theta = NewStrike - HalfPI;
if (Pole.theta < PI)
      Pole.theta += TwoPI;
};
make_polygon_3d();

strike = NewStrike;

(*this)+V;
return;
};


// Procedure to rotate a polygon. The arguments are the new latitude and
// azimuth (theta and phi) of the polygon pole in the absolute frame of
// reference.

void Polygon :: rotate_by_dip (double NewDip)
{
Cartesian V(center.X, center.Y, center.Z);
(*this)-V;
make_polygon_2d();


Pole.phi = NewDip;

make_polygon_3d();


(*this)+V;

return;
};



// Procedure to mark the polygons in a list according to their strike
// compared to the strike of a surface at their centers. Polygons are
// retained in the list if the difference in strike does not exceed some
// specified angle.


void ListPolygons :: mark_parallel_to_strike (Surface& S, double angle, double
Pc)
{
Node* current = head_pol;
int i=0;
Polar SD;

for (i=0; i<Npol; ++i)
{
SD = S.strike_dip_at_point (current->content->center);
if (!(current->content->mark_parallel_to_strike (SD.theta, angle, Pc)))
   {

double angle1 = RandomBC(-angle, angle) + SD.theta;
current->content->rotate_by_strike (angle1);
  };
 current = current->next_pol; };

return;
};



// Procedure to mark the polygons in a list according to their strike
// compared to the strike of a surface at their centers. Polygons are
// retained in the list if the difference in strike does not exceed some
// specified abgle.


void ListPolygons :: mark_orthogonal_to_strike (Surface& S, double angle,
double Pr)
{
Node* current = head_pol;
int i=0;
Polar SD;

for (i=0; i<Npol; ++i)
{
SD = S.strike_dip_at_point (current->content->center);
if (!(current->content->mark_orthogonal_to_strike (SD.theta, angle, Pr)))
   {

double angle1 = RandomBC(-angle, angle) + SD.theta + HalfPI;
current->content->rotate_by_strike (angle1);
  };
 current = current->next_pol; };

return;
};




// Procedure to mark the polygons according to the strike of a surface.
// If the polygon is either sub parallel or sub orhtogonal to the strike
// of the surface, its orientation is kept. Otherwise, the polygon is
// rotated to be sub parallel if the character option is C, or sub orthogonal
// if the character option is R

void ListPolygons :: mark_by_strike (Surface& S, double angleR, double angleC,
double angle2, char option, double Pr, double Pc)
{
Node* current = head_pol;
int i=0;
Polar SD;

for (i=0; i<Npol; ++i)
{
SD = S.strike_dip_at_point (current->content->center);
if (!(current->content->mark_orthogonal_to_strike (SD.theta, angleR, Pr)) &&
!(current->content->mark_parallel_to_strike (SD.theta, angleC, Pc)))
   {
double new_strike;
if (option == 'c')
new_strike = RandomBC(-angle2, angle2) + SD.theta;
if (option == 'r')
new_strike = RandomBC(-angle2, angle2) + SD.theta + HalfPI;
if (new_strike > PI)
   new_strike -= TwoPI;
if (new_strike < -PI)
```

```
   new_strike += TwoPI;

current->content->rotate_by_strike (new_strike);
  };
 current = current->next_pol; };

return;
};


// Procedures to mark the polygons in alist according to their dips. If the
dips
// are close to that of the surface, they are kept. Otherwise the polygons
// are rotated

void ListPolygons :: mark_by_dip (Surface& S, double angle1, double angle2,
double P)
{
Node* current = head_pol;
int i=0;
Polar SD;

for (i=0; i<Npol; ++i)
{
SD = S.strike_dip_at_point (current->content->center);
if (!(current->content->mark_by_dip (SD.phi, angle1, P) ))
  {
    double new_dip = RandomBC(-angle2, angle2) + PI/2 - SD.phi;

current->content->rotate_by_dip (new_dip);
  };

 current = current->next_pol; };

return;
};


void ListPolygons :: mark_by_dip (Cubic& C, double angle1, double angle2,
double P)
{
Node* current = head_pol;
int i=0;
Polar SD;

for (i=0; i<Npol; ++i)
{
SD = C.strike_dip_at_point (current->content->center);
double Ctheta = SD.theta + HalfPI;
   if (Ctheta > PI)
         Ctheta -= TwoPI;


if (!(current->content->mark_by_dip (SD.phi, angle1, P) ))
  {
    double new_dip = RandomBC(-angle2, angle2) - SD.phi + HalfPI;
 if ( (current->content->Pole.theta)*Ctheta > 0. )
```

```
    {    if ( current->content->Pole.theta >= 0.)
            current->content->Pole.theta -= PI;
        else
            current->content->Pole.theta += PI;
    };

current->content->rotate_by_dip (new_dip);
  };

 current = current->next_pol; };

return;
};



// Procedure to mark the polygons according to the strike of a CUBIC surface.
// If the polygon is either sub parallel or sub orhtogonal to the strike
// of the surface, its orientation is kept. Otherwise, the polygon is
// rotated to be sub parallel if the character option is C, or sub orthogonal
// if the character option is R

void ListPolygons :: mark_by_strike (Cubic& C, double angleR, double angleC,
double angle2, char option, double Pr, double Pc)
{
Node* current = head_pol;
int i=0;
Polar SD;

for (i=0; i<Npol; ++i)
{
SD = C.strike_dip_at_point (current->content->center);
if (!(current->content->mark_orthogonal_to_strike (SD.theta, angleR, Pr)) &&
!(current->content->mark_parallel_to_strike (SD.theta, angleC, Pc)))
   {
double new_strike;

if (option == 'c')
new_strike = RandomBC(-angle2, angle2) + SD.theta;

if (option == 'r')
new_strike = RandomBC(-angle2, angle2) + SD.theta + HalfPI;

if (new_strike > PI)
   new_strike -= TwoPI;
if (new_strike < -PI)
   new_strike += TwoPI;

current->content->rotate_by_strike (new_strike);
  };
 current = current->next_pol; };

return;
};
```

stat.C

```
// *******************************************************************
// *                         G E O F R A C                           *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                    (contact einstein@mit.edu)                   *
// *******************************************************************

stat.h
```

surface.C

```
// *******************************************************************
// *                          G E O F R A C                          *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *           Don't use or modify without written permission        *
// *                   (contact einstein@mit.edu)                    *
// *******************************************************************

#include "surface.h"
#include "line.h"

boolian Surface::is_point_on_surface (Point& p) {
double zz=A*p.X*p.X+B*p.X*p.Y+C*p.Y*p.Y+D*p.X+E*p.Y+F;
double dif = p.Z-zz;
if (dif>=-0.000001 && dif<=0.000001)
  {return TRUE;}
else
  {return FALSE;};
};


// Procedure to calculate if a point P(X,Y,Z) is above or below a surface.
// If Z<z on the surface at (X,Y), the point is below and the function returns
-1.
// If Z>z on the surface at (X,Y), the point is above and the function returns
1.

int Surface :: is_point_above_below (Point& P)
{
if (is_point_on_surface (P))
   return 0;

double zz = find_Z_on_surface (P.X, P.Y);

if (P.Z>zz)
  {return 1; };      // The point is above the surface
if (P.Z<zz)
  {return -1; };      // The point is below the surface
};


boolian Surface::is_line_on_surface (Line& l) {
if ((is_point_on_surface(l.end1)) && (is_point_on_surface(l.end2)))
   return TRUE;
else
   return FALSE;
};


// Procedure to find out if a line lies above or below a surface, or if the
line
```

```
// intersects the surface. 1 is returned if both ends of the line are above
the
// surafce, -1 if they are below the surface. If one end is above and the
other
// one is below the surface, 0 is returned to inticate that the line
// intersects the surface.


int Surface :: how_line_intersect (Line& L)
{
int a1 = is_point_above_below (L.end1);
int a2 = is_point_above_below (L.end2);
if (a1*a2 < 0)                // The line intersects the surface
    return 0;
else if (a1+a2 > 0)           // Both ends are on the same side of the surface
  return 1;                   // The line is above the surface
else
  return -1;                  // The line is below the surface
};


// This function calculates the coordinates of the normal vector
// to a quadratic surface at a point p(X,Y,Z) on the surface.
// The normal vector is defined by the first derivative of the
// surface equation f(x,y,z)=0, i.e. n=(df/dx,df/dy,df/dz).
// For quadratic surface (see definiton of class Surface)
// the normal vector is (-2*A*X-B*Y-D, -2*C*Y-B*X-E, 1)

Cartesian Surface::normal_at_point(Point& p) {
double xx=2*A*p.X+B*p.Y+D;
double yy=2*C*p.Y+B*p.X+E;

Cartesian* vector = new Cartesian(-xx, -yy, 1.);
return (*vector);
};


// This function finds the Z coordinate of an (X, Y) point such that
// point (X,Y,Z) is on the given surface

double Surface::find_Z_on_surface (double x, double y) {
double Z=A*x*x+B*x*y+C*y*y+D*x+E*y+F;
return Z;
};


// This function finds the azimuth and latitude of a surface at a point.
// Latitude is calculated as the angle from north (axis X). Positive values
// are to the east, negative values are to the west. Axis Y is east.
// Azimuth is calculated as the angle between the normal vector at the point
// and the vertical axis Z. A Polar (latitude, azimuth) is returned.
// The coordinate system (X,Y,Z) is left-handed.

Polar Surface::polar_normal_at_point (Point& p) {
Cartesian N = normal_at_point(p);
double theta, phi;
double l3d = sqrt(N.X*N.X+N.Y*N.Y+N.Z*N.Z);
double l2d = sqrt(N.X*N.X+N.Y*N.Y);

phi = asin(l2d/l3d);
theta = acos(N.Y/l2d);
if (N.X<0.)
    theta*=-1.;

Polar* polar_normal = new Polar(theta, phi);
return (*polar_normal);
};


// This function returns the strike and dip of a surface at a point,
// calculated as the strike and dip of a tangent plane. Dip is between zero
// and pi/2 measured from the horizontal to the slope in a vertical plane.
// Strike is between -pi to pi, negative values are west from north,
// positive values are east from north.
// The system NTB is left-handed, where N is the normal vector, T is the
// strike tangent vector, and B=NxT is binormal pointing upward along
// the dip.

Polar Surface::strike_dip_at_point (Point& p) {
Polar SD = polar_normal_at_point(p);
SD.theta-=HalfPI;

if (SD.theta < -PI)
    SD.theta+=TwoPI;
return SD;
};

// This function calculates the volume enclosed under the surface over the
// rectangular area (x, y) where x=[-xm, xm] and y=[-ym, ym]

double Surface::find_enclosed_volume (double xm, double ym) {

double volume = xm*ym*(4./3.*(A*xm*xm+C*ym*ym)+B*xm*ym+2.*(D*xm+E*ym)+4.*F);
return volume;
};


// Function to find the maximum Z value of a surface above a rectangular
// area bounded by X, -X, Y, and -Y. Zmax is chosen to be the maximum value
// of: 1) the values at the four end points; 2) the local maximum of the
// function along the edges, if such a maximum exists; 3) the local maximum
// inside the area (X,Y), if such a maximum exists. A local maximum of f(x)
// exists at the  point where the df/dx is zero, and d^2f/dx^2 is negative.
// For f(x, y) the function has a local maximum if df/dx=df/dy=0, and
// d^2f/dx^2*d^2f/dy^2-(d^2f/dxdy)^2<0. This procedure ignores some cases
// for which the test is inconclusive (see calculus references). For an
// upward convex surface such cases are unlikely.

double Surface::Zmax_over_XY (double X, double Y) {
double Zmax, zz, x, y;
Zmax = find_Z_on_surface (X, Y);
zz = find_Z_on_surface (-X, Y);
     if (zz>Zmax)
```

```
         Zmax=zz;
zz = find_Z_on_surface (-X, -Y);
     if (zz>Zmax)
         Zmax=zz;
zz = find_Z_on_surface (X, -Y);
     if (zz>Zmax)
         Zmax=zz;

double f = 4*A*C-B*B;
if (f<0.)                         // possible maximum inside (X, Y)
       { x=(B*E-2*D*C)/f;
       y=(B*D-2*A*E)/f;           // point where 1st derivatives are 0.
zz = find_Z_on_surface(x, y);
       if (zz>Zmax)
         Zmax=zz;
       };


if (C<0.)                         // possible maximum along x=X or x=-X
  {
    y=(-E-B*X)/(2*C);             // checking max along x=X
        if (y>-Y && y<Y)
            zz = find_Z_on_surface(X, y);
        if (zz>Zmax)
            Zmax=zz;
     y=(-E+B*X)/(2*C);           // checking max along x=-X
        if (y>-Y && y<Y)
            zz = find_Z_on_surface(-X, y);
        if (zz>Zmax)
            Zmax=zz;
  };


if (A<0.)                         // possible maximum along y=Y or y=-Y
  {
    x=(-D-B*Y)/(2*A);            // checking max along y=Y
        if (x>-X && x<X)
            zz = find_Z_on_surface(x, Y);
        if (zz>Zmax)
            Zmax=zz;
     x=(-D+B*Y)/(2*A);           // checking max along y=-Y
        if (x>-X && x<X)
            zz = find_Z_on_surface(x,-Y);
        if (zz>Zmax)
            Zmax=zz;
  };

return Zmax;
};
```

volume.C

```
// ******************************************************************
// *                           G E O F R A C                        *
// *    Copyright Massachusetts Institute of Technology 1995-1998   *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein        *
// *                                                                *
// *          Don't use or modify without written permission        *
// *                   (contact einstein@mit.edu)                   *
// ******************************************************************

#include "volume.h"

double Random01 ();
double Random0a (double a);
double  RandomBC (double  b, double c);
extern double Xm, Ym;


// Constructor for the total volume enclosed under a quadratic surface above
// a rectangular area defined by the vertical planes x=Xm, x=-Xm, y=Ym,
// and y=-Ym. The origin of the coordinate system is assumed to be
// in the middle of the rectangular area, on the horizontal plane bounding
// the modeling volume from below.

Volume :: Volume (Surface& ground)
  {
    top=ground;
    P[0].X=Xm; P[0].Y=Ym; P[0].Z = top.find_Z_on_surface(Xm, Ym);
    P[1].X=-Xm; P[1].Y=Ym; P[1].Z = top.find_Z_on_surface(-Xm, Ym);
    P[2].X=-Xm; P[2].Y=-Ym; P[2].Z = top.find_Z_on_surface(-Xm, -Ym);
    P[3].X=Xm; P[3].Y=-Ym; P[3].Z = top.find_Z_on_surface(Xm, -Ym);
    volume=top.find_enclosed_volume(Xm, Ym);
    Zmax=top.Zmax_over_XY(Xm, Ym);
  };


// Procedure to generate a random point inside the modeling volume.
// The coordinates of the point are calculated uniformly as x=U(-Xm, Xm),
// y=U(-Ym, Ym), z=U(0, Zmax). After that it is calculated if it is inside
// the modeling volume (if Z of the point is below the top surface);
// if not, a new point is generated.

Point Volume :: random_point() {
Point* p = new Point;
do {
p->X = RandomBC (-Xm, Xm);
p->Y = RandomBC (-Ym, Ym);
p->Z = Random0a (Zmax);
}
while (!is_point_inside (*p));

return (*p);
};
```

```
boolian Volume :: is_point_inside (Point& p) {
if (p.X>=-Xm && p.X<=Xm && p.Y>=-Ym && p.Y<=Ym)
  {double zz = top.find_Z_on_surface(p.X, p.Y);
    if ( p.Z<=zz )
        return TRUE;};
return FALSE;
};

// Function to return the minimum z-coordinate of the 4 corners of an
// object Volume.

double Volume :: corner_min_z()
{
int i;

double min_z=P[0].Z;
for(i=1; i<4; ++i)
  {
  if(P[i].Z < min_z)
    min_z=P[i].Z;
  };
return min_z;
};
```

zones.C

```
// *******************************************************************
// *                        G E O F R A C                            *
// *    Copyright Massachusetts Institute of Technology 1995-1998    *
// *         Violeta Ivanova, Thomas Meyer, Herbert Einstein         *
// *                                                                 *
// *          Don't use or modify without written permission         *
// *                  (contact einstein@mit.edu)                     *
// *******************************************************************

#include "polygon.h"
#include "listpol.h"

#include <math.h>
#include <iostream.h>
#include <stdlib.h>
#define FALSE 0
#define TRUE 1


double Random01 ();


// Procedure to mark a polygon with a zone probability Pi. The shortest
// distance D to a list of polygons is found (the faults). The polygon is
// considered for marking only if it is located in front of the fault which
// is exactly at distance D. Then this distance is compared
// to distances defined by the array zones to find out in which zone i
// is the polygon. Then the polygon is marked as fracture with a
// marking probability Pi for zone i, defined by the array marks. The function
// returns TRUE if a generated random number is smaller than Pi, or FALSE
// if the number is larger than Pi. Pi and the random number are between 0.
// and 1.


boolian ListPolygons :: if_zone_mark_polygon (Polygon& pol, int Nzones,
double* zoneDmax, double* zoneP)
{
double D = shortest_distance_from_polygon (pol);
double mark = *(zoneP+Nzones-1);
int MARKED = FALSE;
int i;
int RESULT=FALSE;
double disttest;

Node* current=head_pol;
while (current)
  {
  disttest=D - current->content->distance_from_polygon(pol);
  if(current->content->if_front_of_polygon(pol) && disttest*disttest<0.0001)
    RESULT=TRUE;
  current=current->next_pol;
  };
```

```
if(RESULT)
  {
for (i=0; i<Nzones-1 && !MARKED; ++i)
  {
if (D < *(zoneDmax+i))
  { mark = *(zoneP+i);
  MARKED = TRUE; };
  };

double RN = Random01();
if (RN <= mark)
 RESULT=TRUE;
else
 RESULT=FALSE;
  };
return RESULT;
};


// Procedure to mark a polygon if he belongs to a zone defined by a box. The
// polygon can be either inside or outside the box, with associated marking
// probabilities.

boolian Box::if_box_mark_polygon(Polygon& pol, double* zoneP)

{
int MARKED=FALSE;
double RN=Random01();
double mark=*zoneP;

if(is_point_inside(pol.center) && RN<mark)
  MARKED=TRUE;

mark=*(zoneP+1);

if(!(is_point_inside(pol.center)) && RN<mark)
  MARKED=TRUE;

return MARKED;
};


// Procedure to mark the polygons in a list of polygons with zone
// probabilities according to the distances to another list of polygons.
// The argument ListPolygons are teh polygons (usually major faults) the
// distances from which define the zones. The list of polygons that calls
// the function (usually new fractures) are the ones that are being marked.

void ListPolygons :: mark_by_zones (ListPolygons& lp, int Nzones, double*
zoneDmax, double* zoneP)

{
Node* current = head_pol;
while (current -> next_pol)
  {
if (!(lp.if_zone_mark_polygon (*current->next_pol->content, Nzones, zoneDmax,
zoneP)))
```

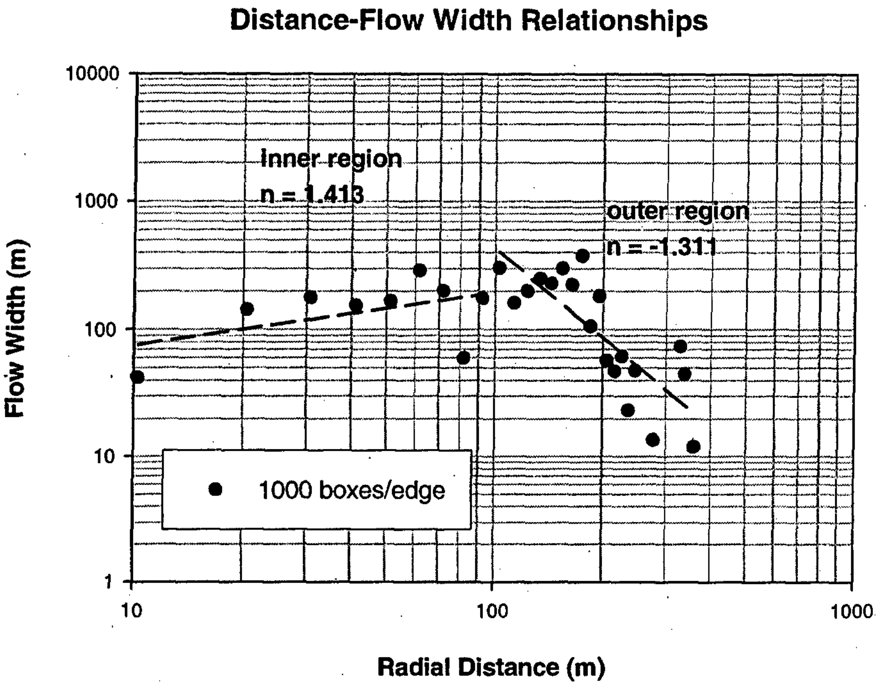
```
  { if (current->next_pol->next_pol)
                  delete current->next_pol->content;
  current->next_pol = current->next_pol->next_pol;
  -- Npol; }
else
  current = current->next_pol;
  };

if (!(lp.if_zone_mark_polygon (*head_pol->content, Nzones, zoneDmax, zoneP)))
  {head_pol = head_pol->next_pol;
  -- Npol; };

return;

};


// Procedure to mark polygons of a list of polygons with zone probabilites
// according to their location with respect to a box. If the centers of the
// polygons are outside of the box, they are discarded, and if the centers are
// inside the box, the polygons are retained only if a randomly generated
// number is lower than a given probability.

void ListPolygons::mark_by_box(Box& ZB, double* zoneP)

{
Node* current = head_pol;
while (current -> next_pol)
  {
  if(!(ZB.if_box_mark_polygon(*current->next_pol->content, zoneP)))
    {
    if (current->next_pol->next_pol)
      delete current->next_pol->content;
    current->next_pol = current->next_pol->next_pol;
    -- Npol;
    }
  else
    current = current->next_pol;
  };

if (!(ZB.if_box_mark_polygon(*head_pol->content, zoneP)))
  {
  head_pol = head_pol->next_pol;
  -- Npol;
  };

return;
};


// Function to consider the size of a polygon in the pdf of fracture sizes.
// The function finds the specified "zone" by polygon size and increases the
// number of polygons that have such size specified by Amin < Apol < Amax.

void Polygon :: include_in_size_pdf (int Nzones, int* Nin_zones, double*
maxAratio, double MeanA)
{
```

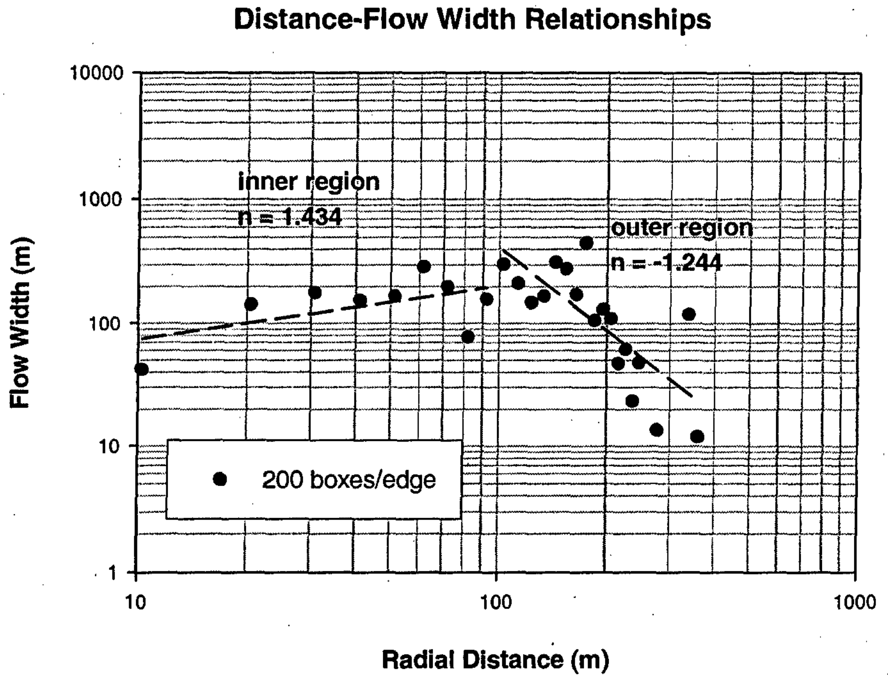
```
int i;

for (i=0; i<Nzones; ++i)
  {
if ((area/MeanA) < *(maxAratio+i))
  { ++ *(Nin_zones+i);
    return;};
  };

++ *(Nin_zones + Nzones);
return;

};



// Function to find the size distribution of a list of polygons

void ListPolygons:: size_distribution (int Nzones, int* Nin_zones, double*
maxAratio, double MeanA)
{
int i;
for (i=0; i<Nzones; ++i)
 Nin_zones[i] = 0;

Node* current = head_pol;

while (current)
  {
current->content->include_in_size_pdf (Nzones, Nin_zones, maxAratio, MeanA);
current = current->next_pol;
  };
return;

};
```

Cubical Element Approach Results (Radial Distance-Flow Width Relationships)

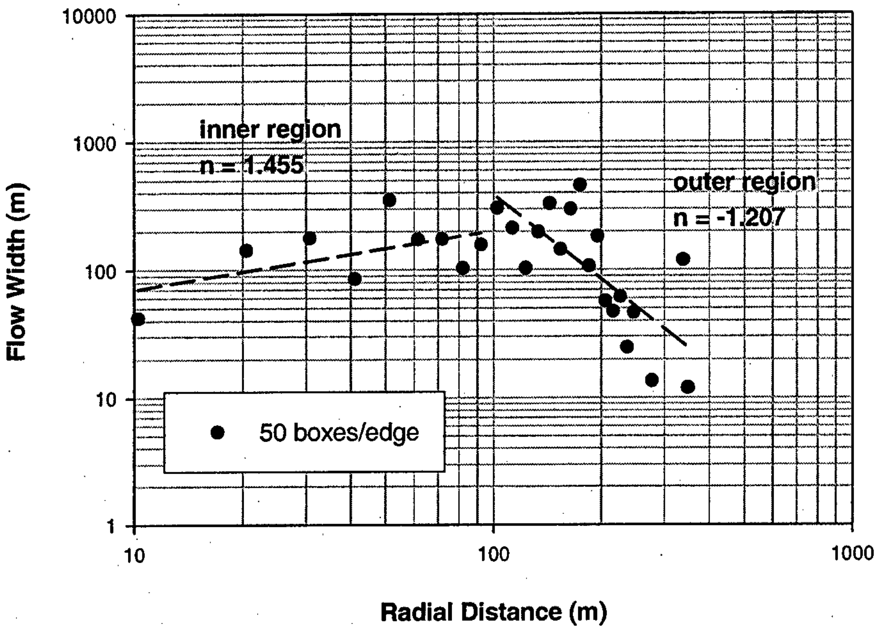
Short Well (10 m long)

Network 1

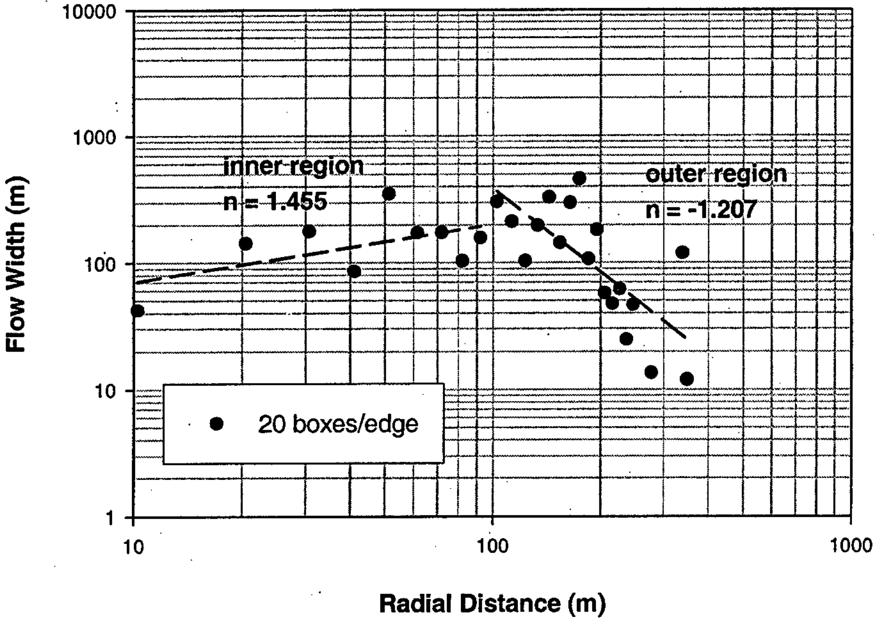




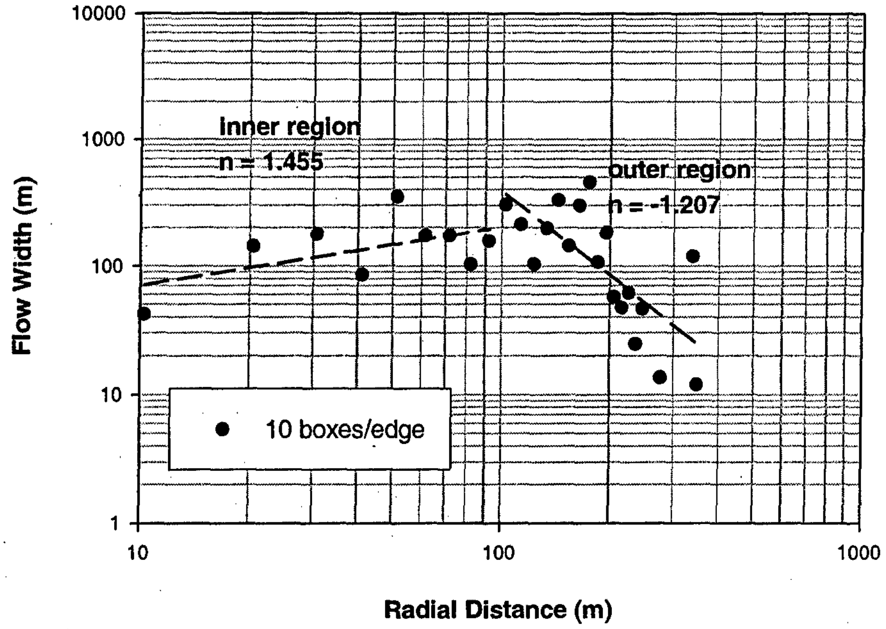
Distance-Flow Width Relationships



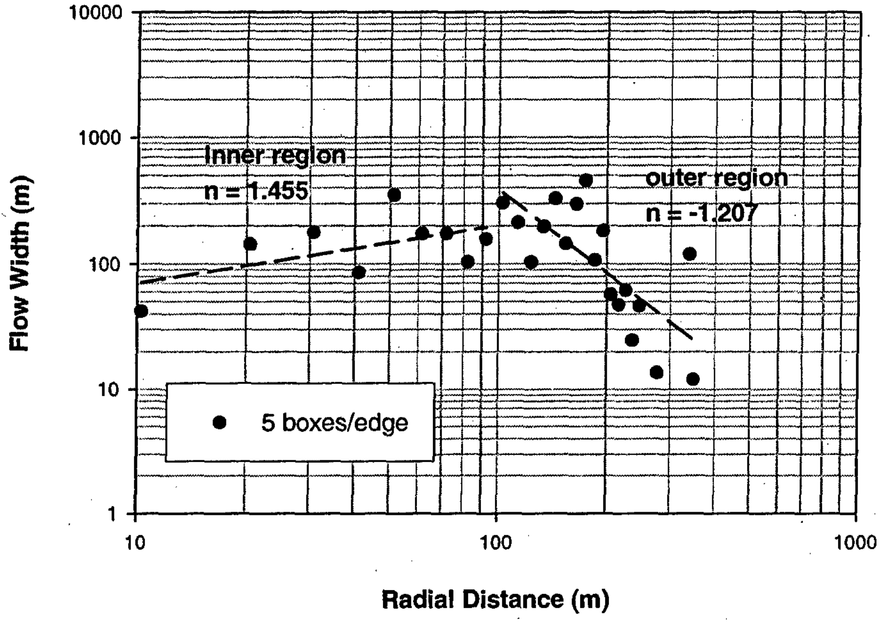
Distance-Flow Width Relationships



Distance-Flow Width Relationships

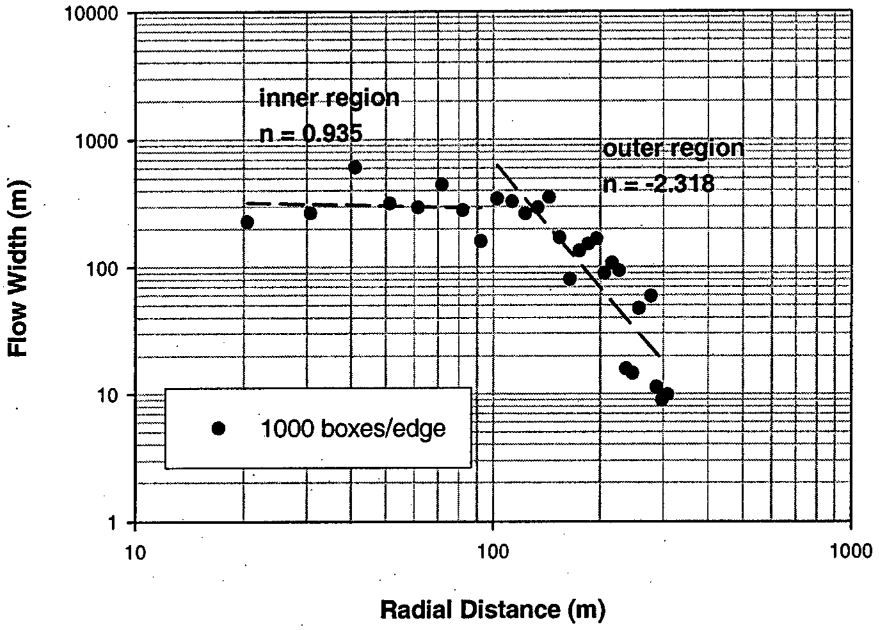


Distance-Flow Width Relationships

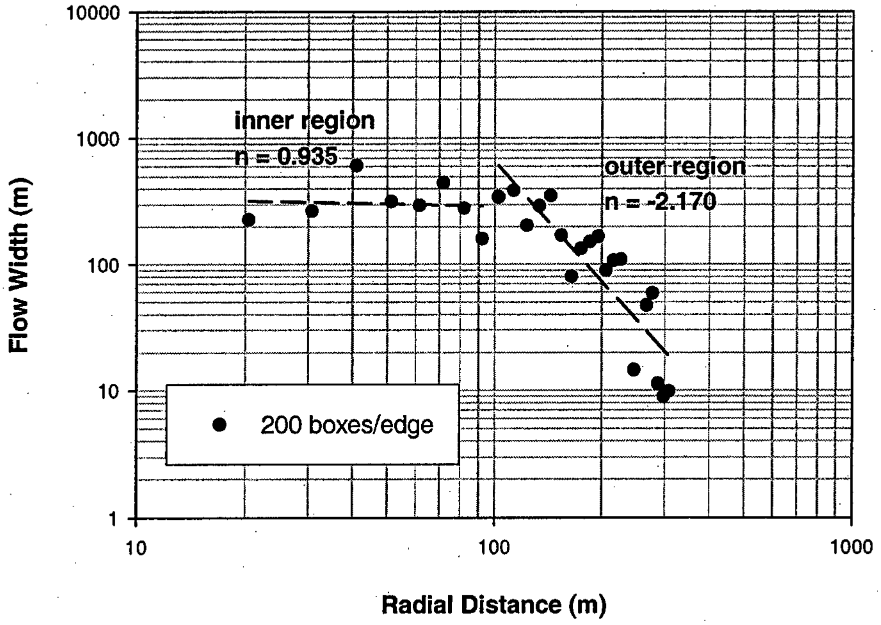


Network 2

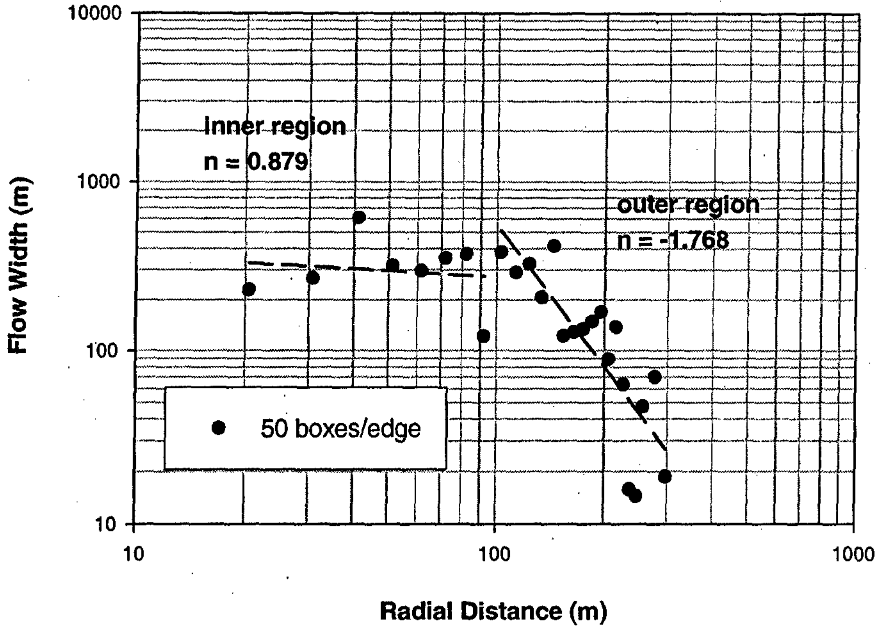
Distance-Flow Width Relationships



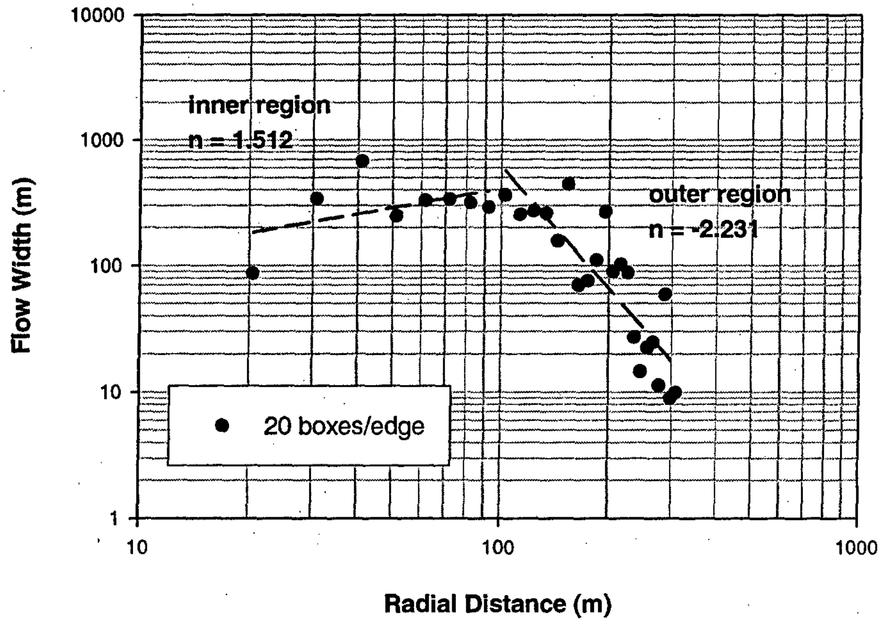
Distance-Flow Width Relationships

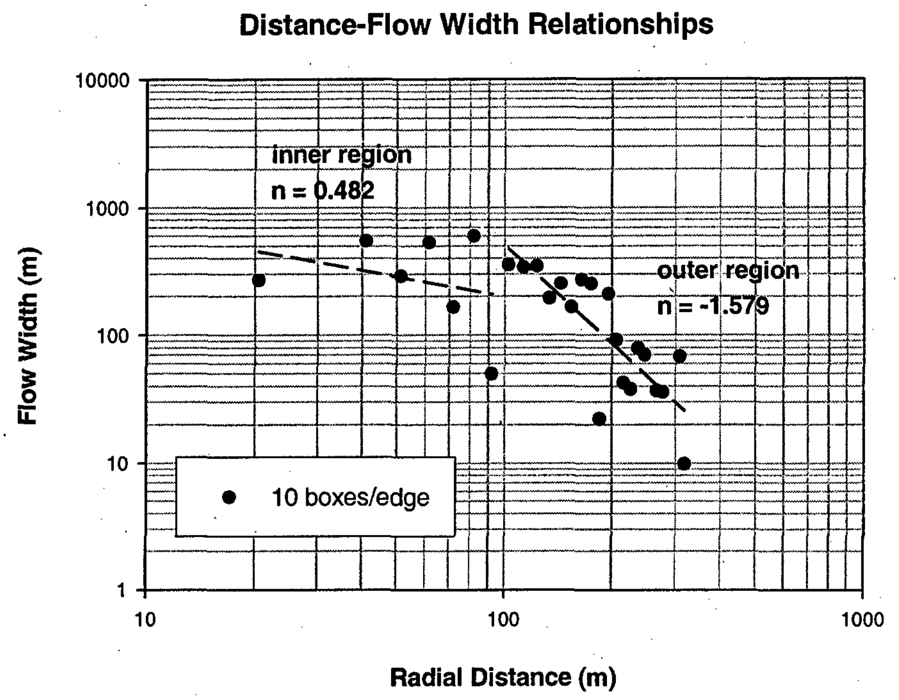


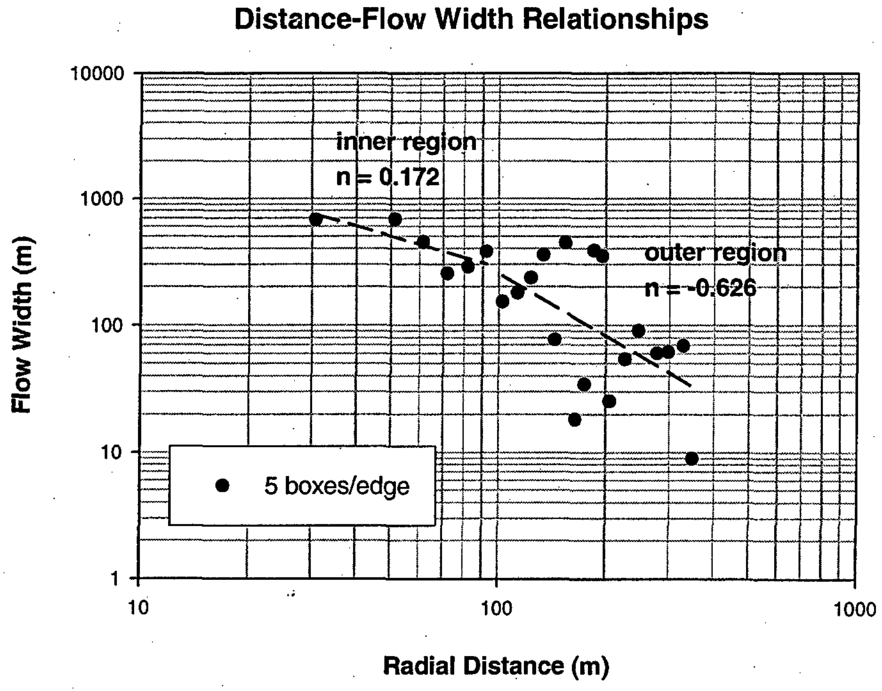
Distance-Flow Width Relationships



Distance-Flow Width Relationships

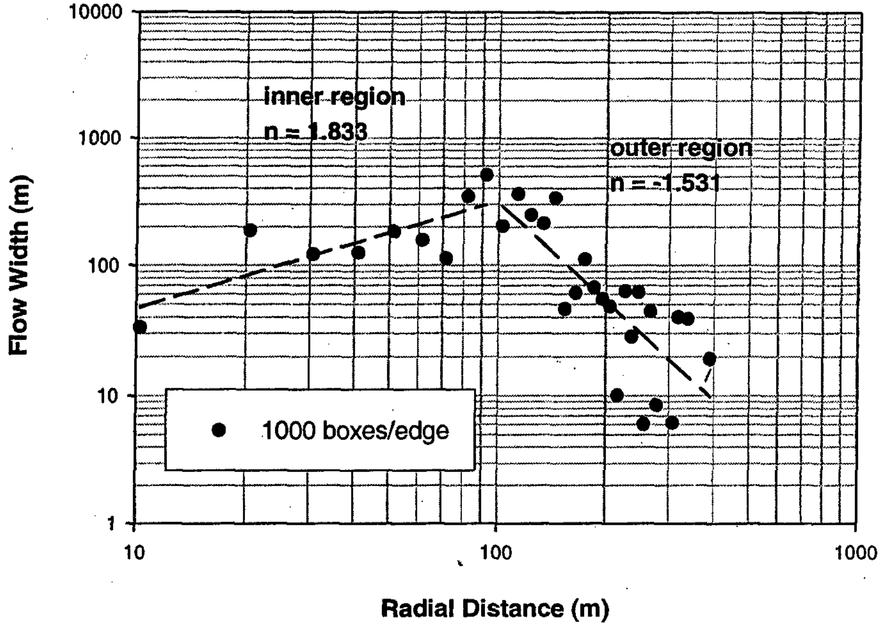



Distance-Flow Width Relationships
inner region
n = 0.482
outer region
n = -1.579
10 boxes/edge
Flow Width (m)
10000
1000
100
10
1
10
100
1000
Radial Distance (m)

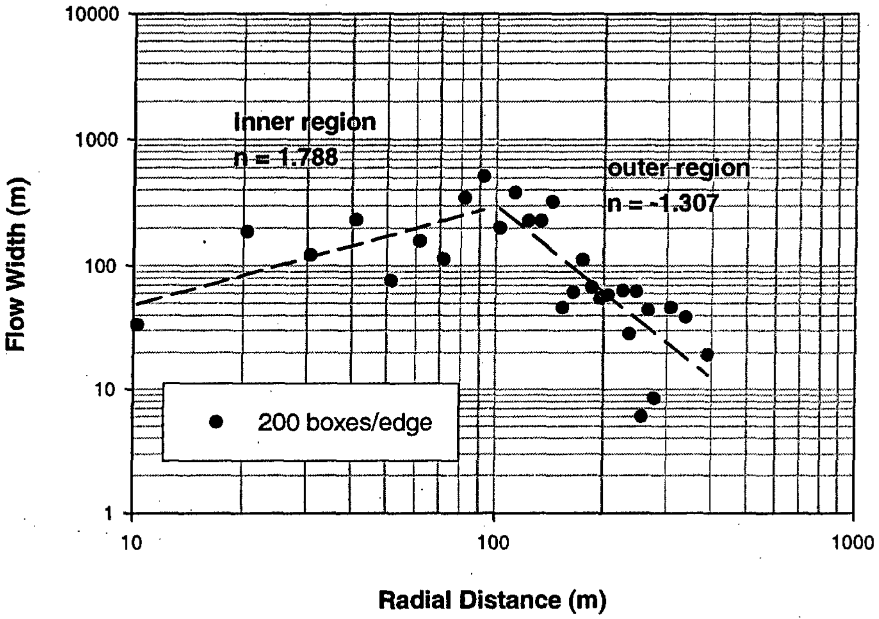

Distance-Flow Width Relationships
inner region
n = 0.172
outer region
n = -0.626
5 boxes/edge
Flow Width (m)
10000
1000
100
10
1
10
100
1000
Radial Distance (m)

Network 3

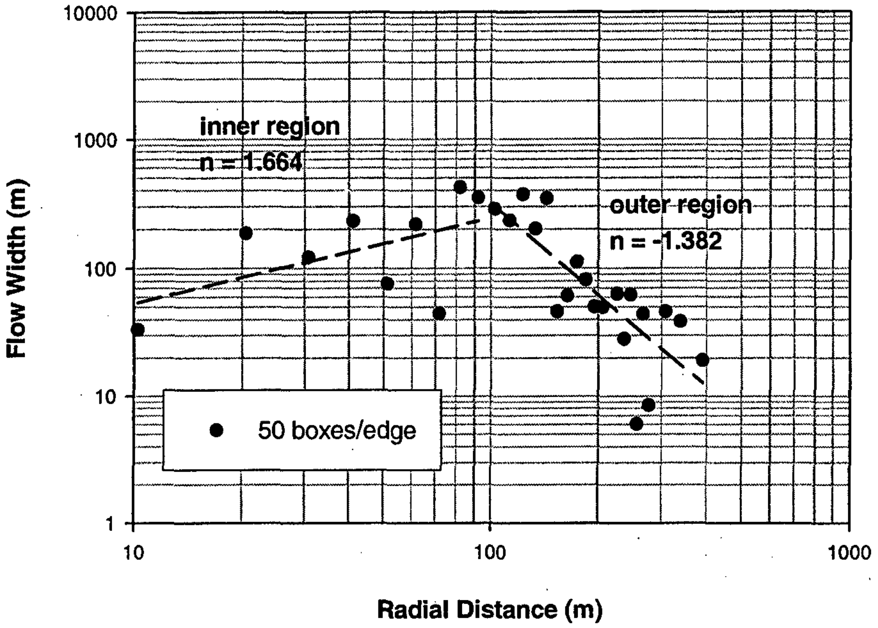
Distance-Flow Width Relationships



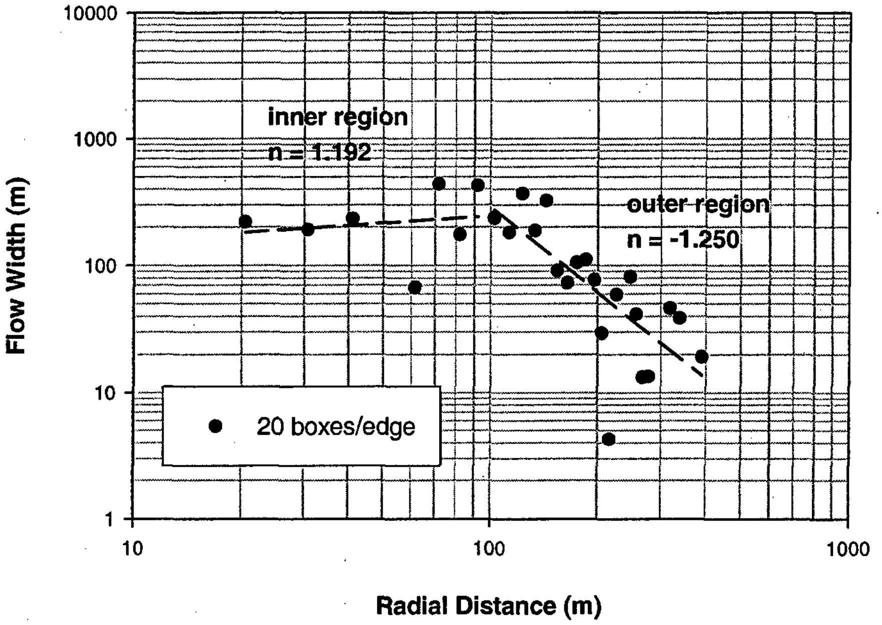
Distance-Flow Width Relationships



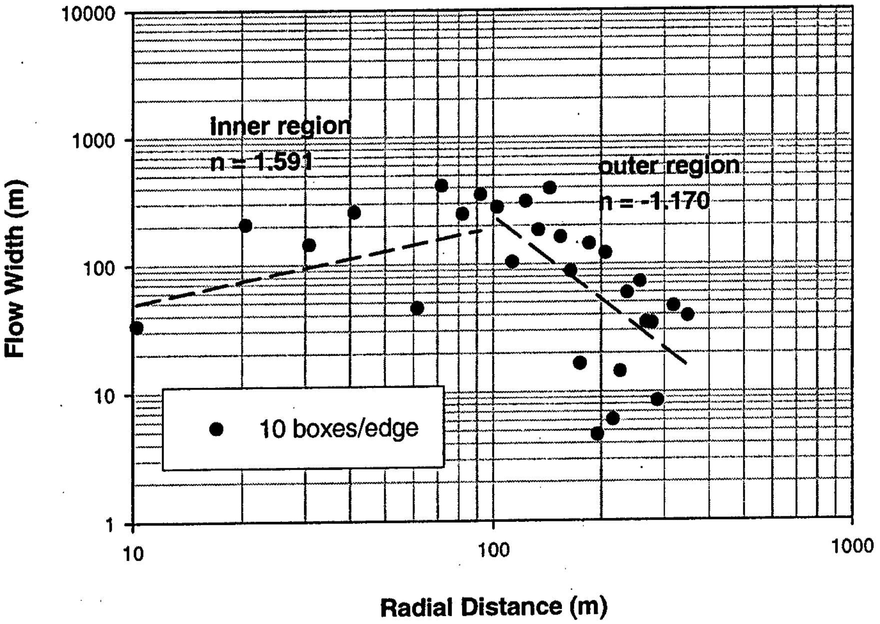
Distance-Flow Width Relationships



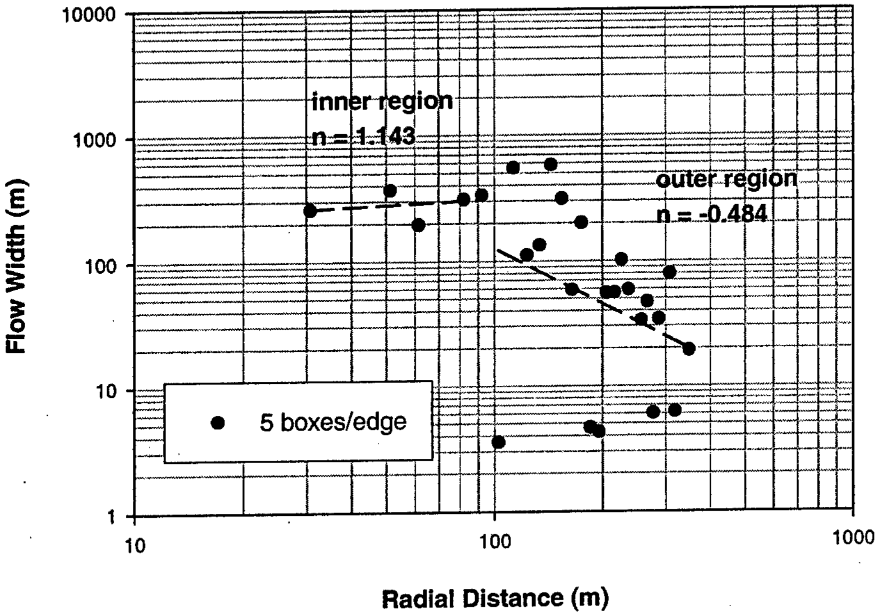
Distance-Flow Width Relationships



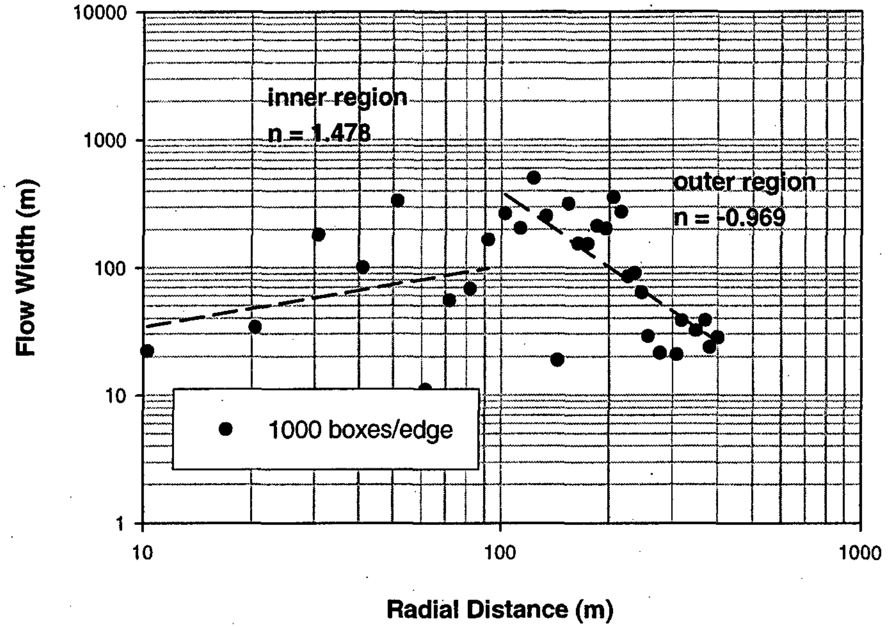
Distance-Flow Width Relationships



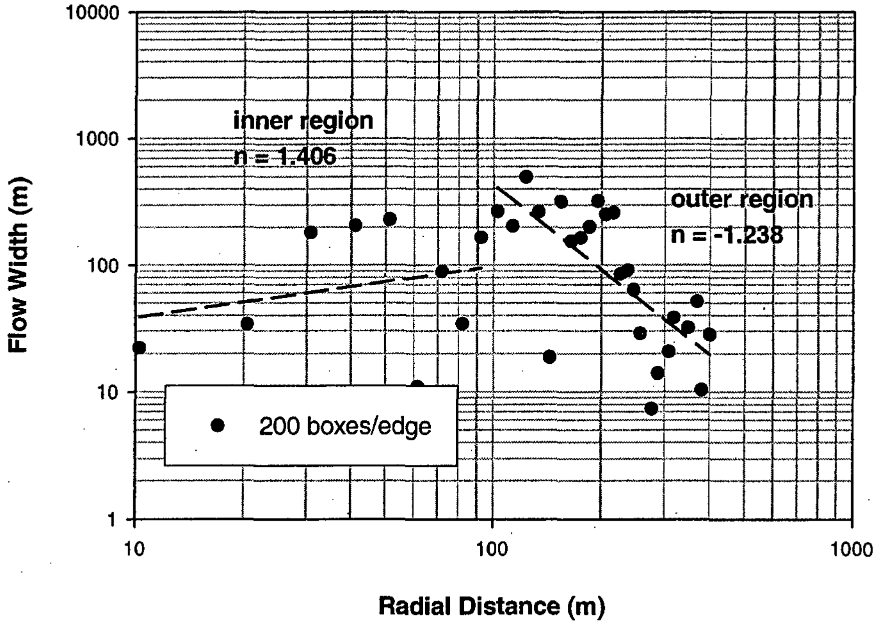
Distance-Flow Width Relationships



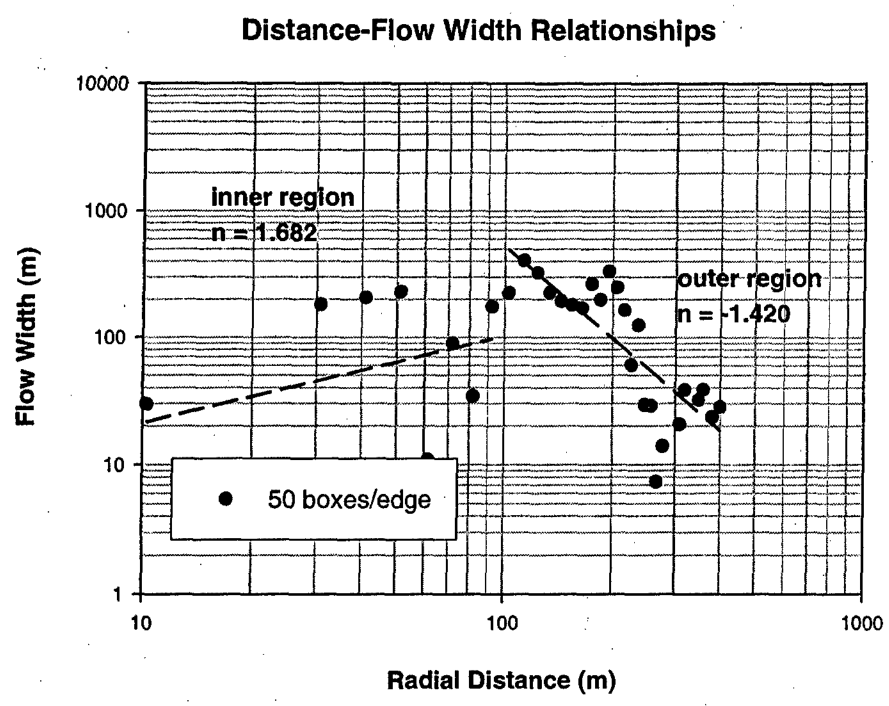
Network 4



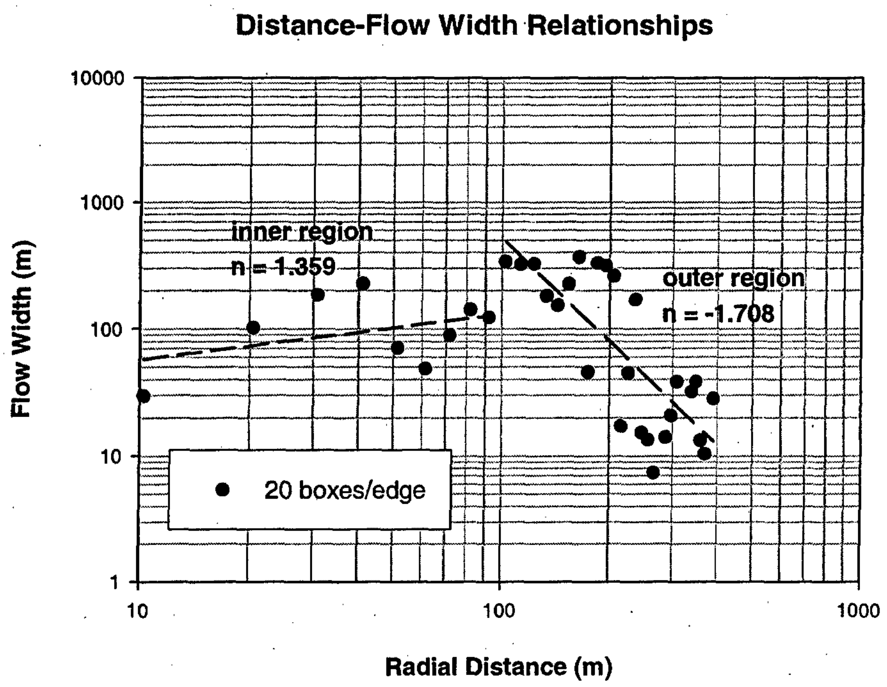
Distance-Flow Width Relationships



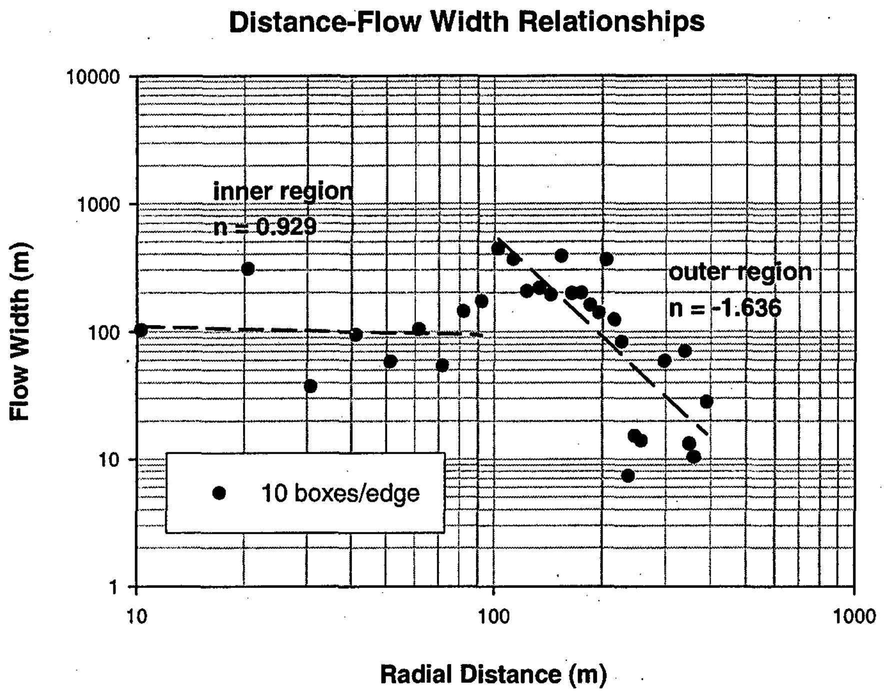
Distance-Flow Width Relationships

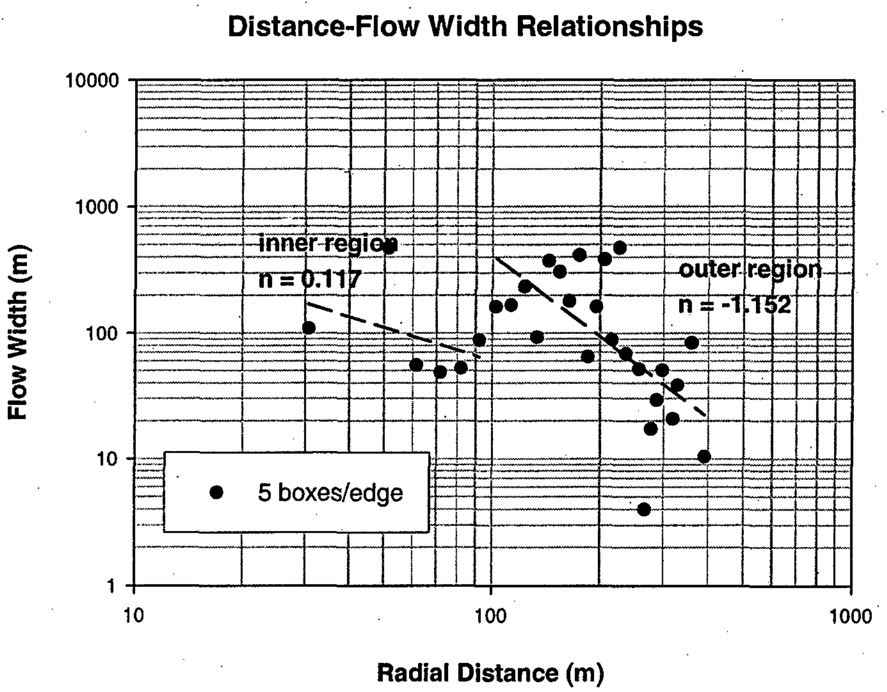


Distance-Flow Width Relationships

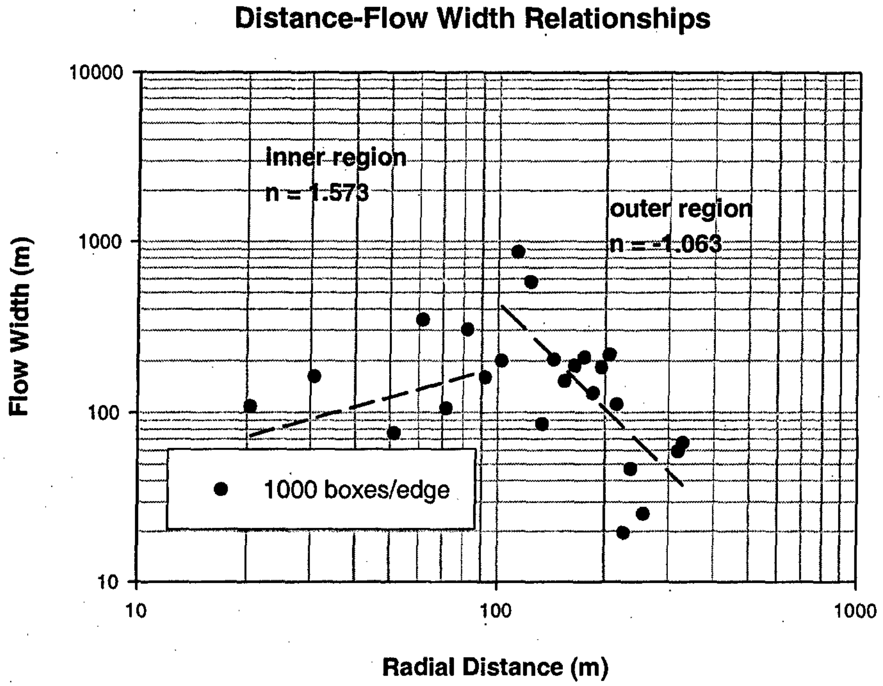


Distance-Flow Width Relationships

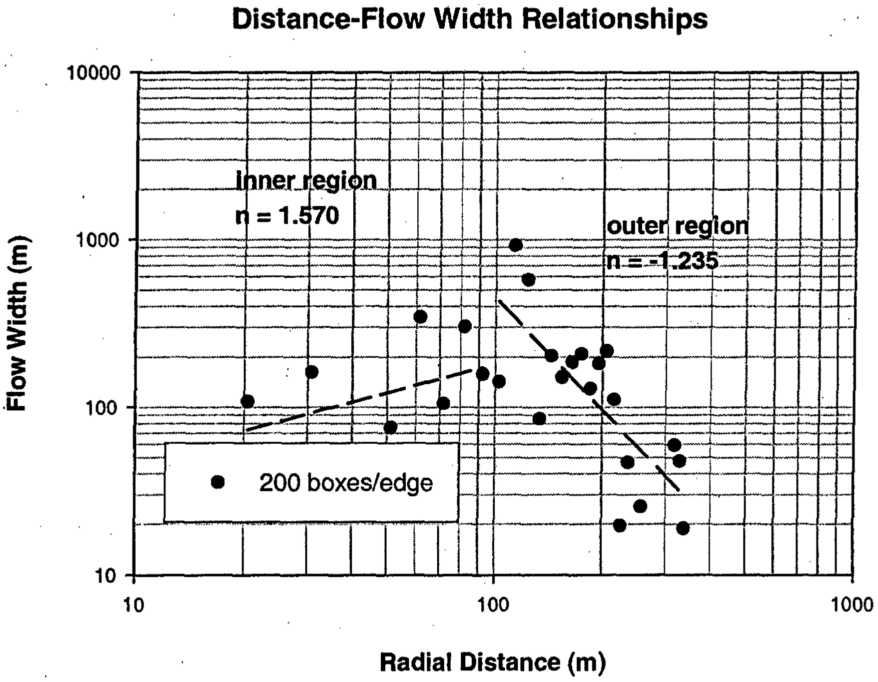

Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
Radial Distance (m)
10
100
1000
inner region
n = 0.929
outer region
n = -1.636
● 10 boxes/edge


Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
Radial Distance (m)
10
100
1000
inner region
n = 0.117
outer region
n = -1.152
● 5 boxes/edge

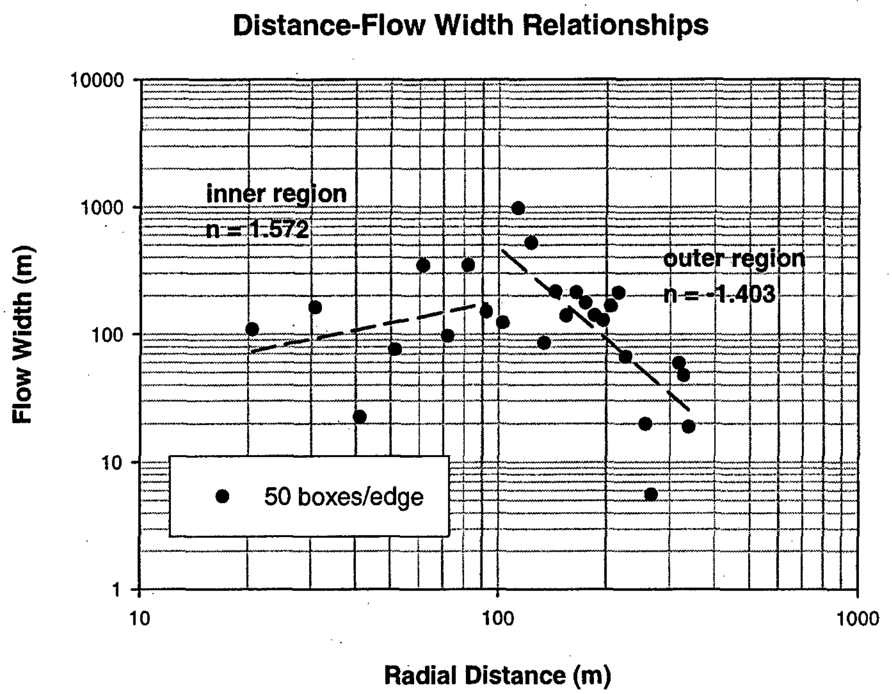
Network 5

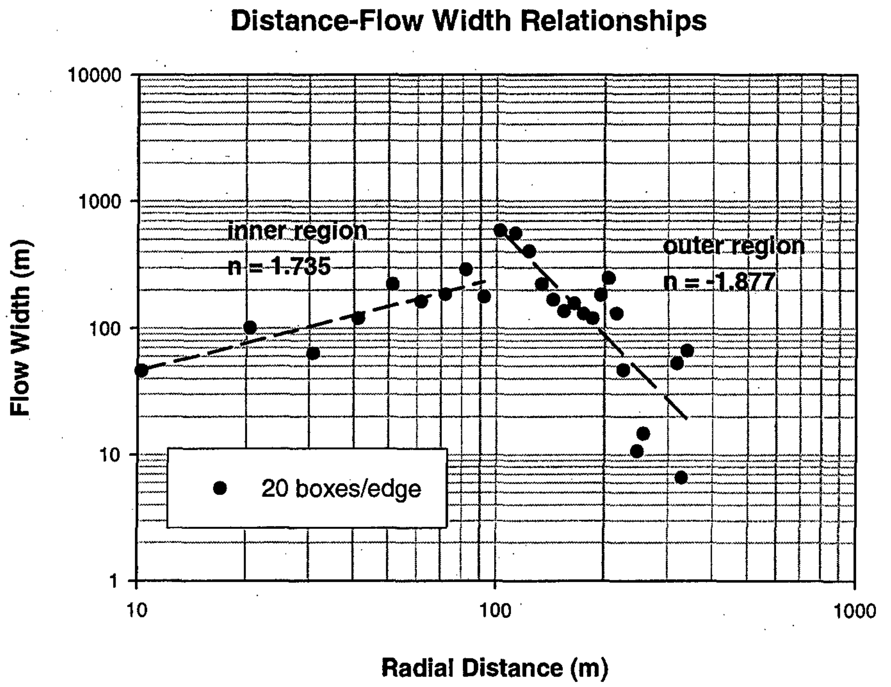


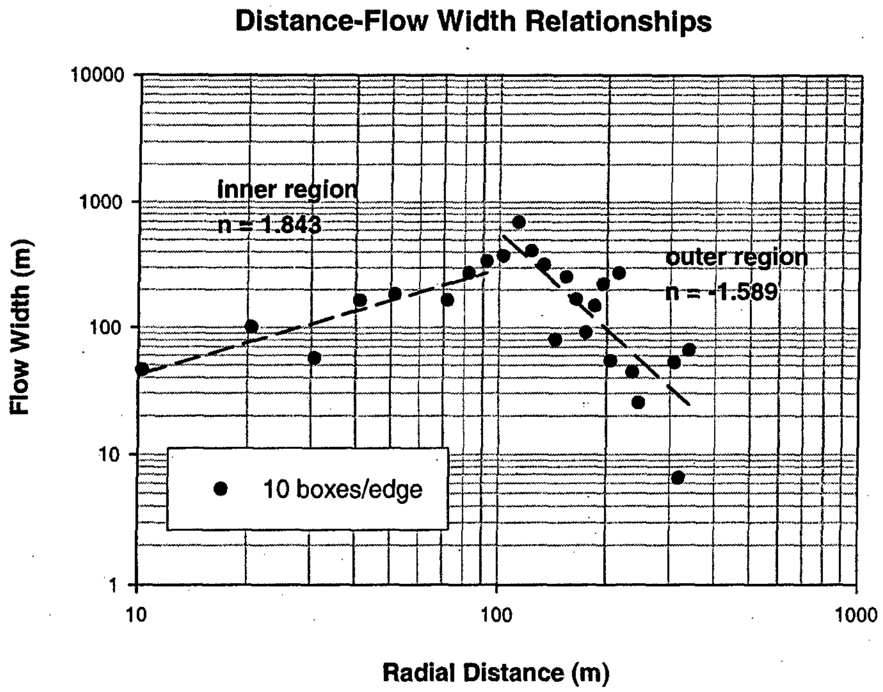
Distance-Flow Width Relationships
10000
1000
100
10
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 1.573
outer region
n = -1.063
● 1000 boxes/edge

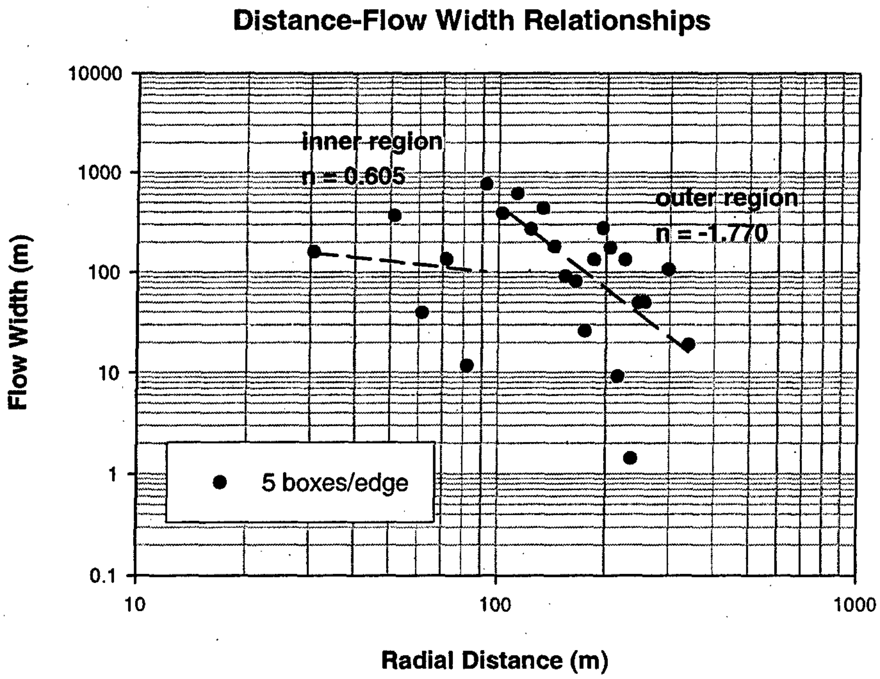


Distance-Flow Width Relationships
10000
1000
100
10
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 1.570
outer region
n = -1.235
● 200 boxes/edge


Distance-Flow Width Relationships
Flow Width (m)
Radial Distance (m)
10000
1000
100
10
1
10
100
1000
inner region
n = 1.572
outer region
n = -1.403
● 50 boxes/edge

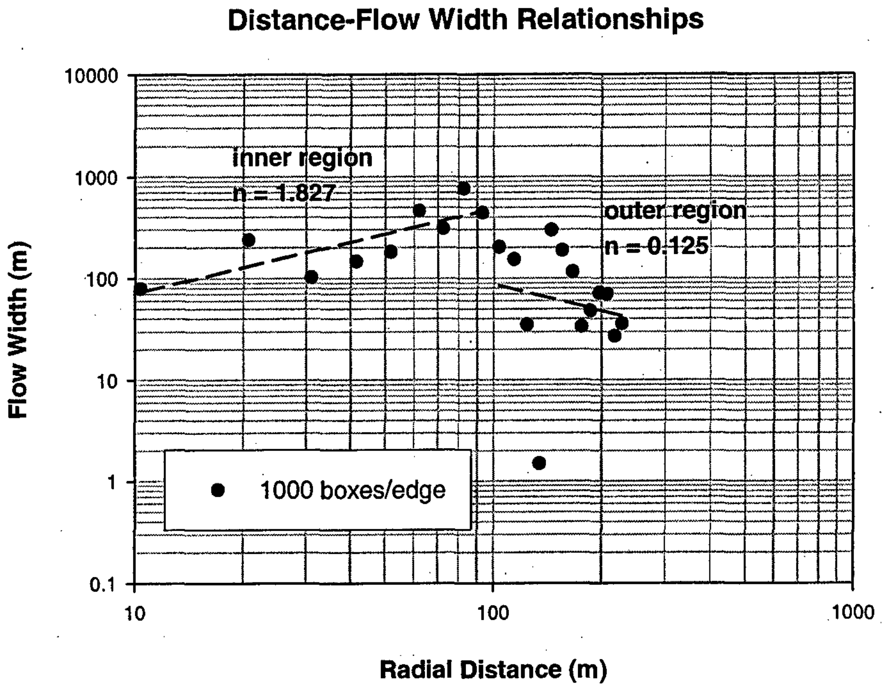

Distance-Flow Width Relationships
Flow Width (m)
Radial Distance (m)
10000
1000
100
10
1
10
100
1000
inner region
n = 1.735
outer region
n = -1.877
● 20 boxes/edge

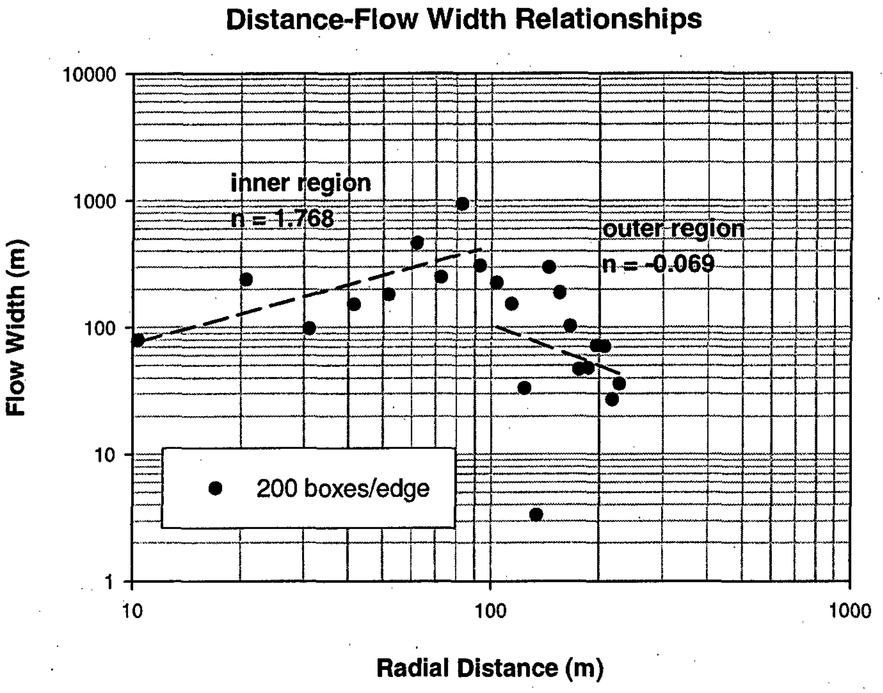

Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 1.843
outer region
n = -1.589
● 10 boxes/edge

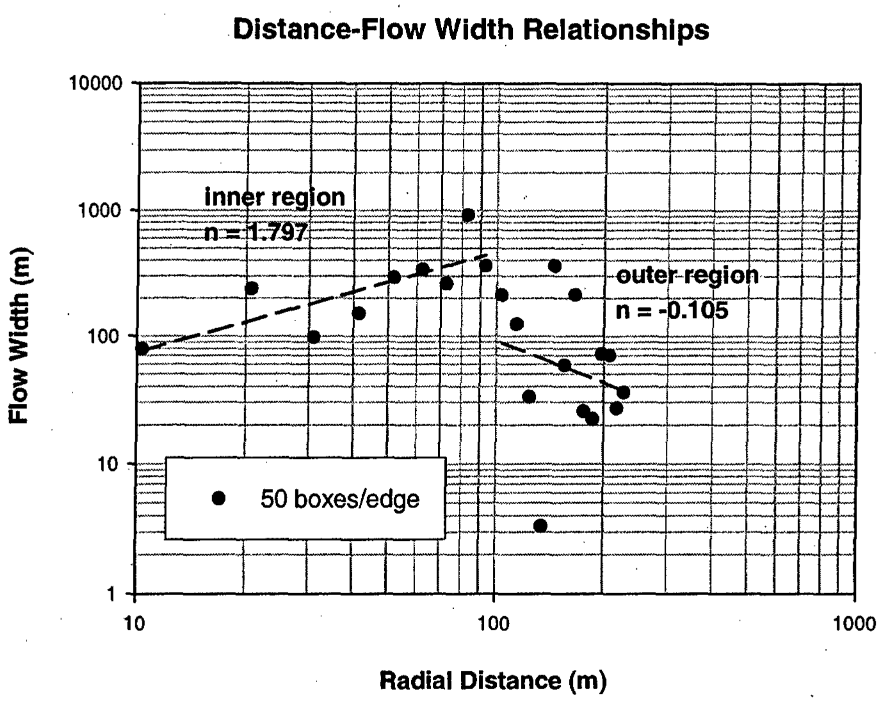

Distance-Flow Width Relationships
10000
1000
100
10
1
0.1
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 0.605
outer region
n = -1.770
● 5 boxes/edge

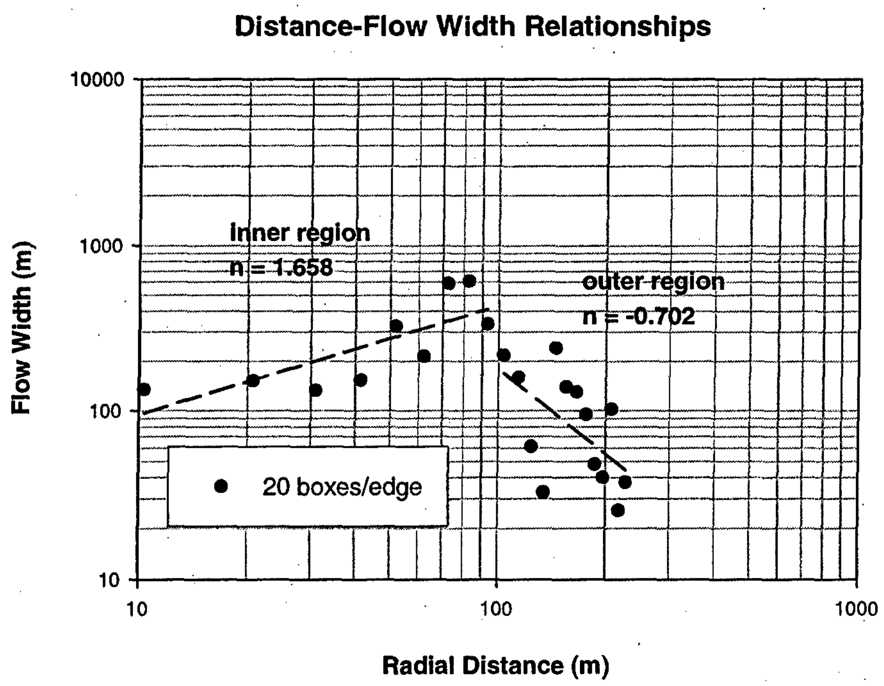
Long Well (180 m long)

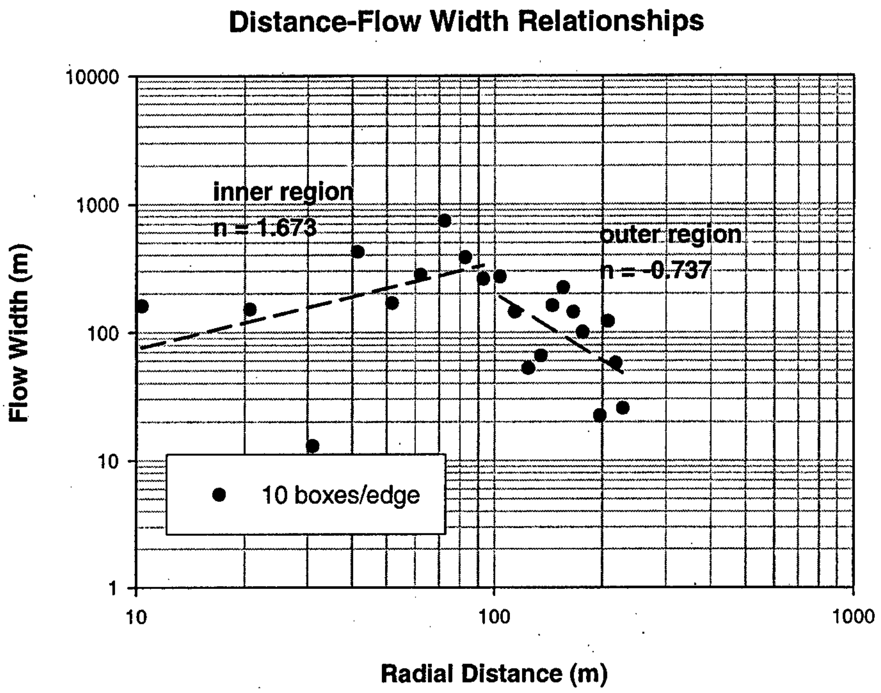
Network 1

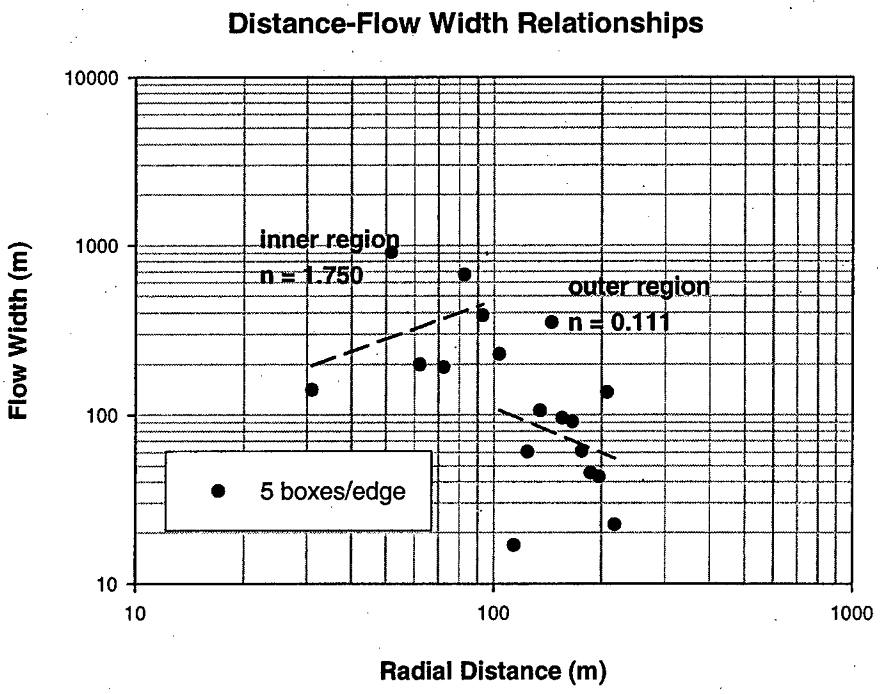





Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
inner region
n = 1.797
outer region
n = -0.105
● 50 boxes/edge
10
100
1000
Radial Distance (m)

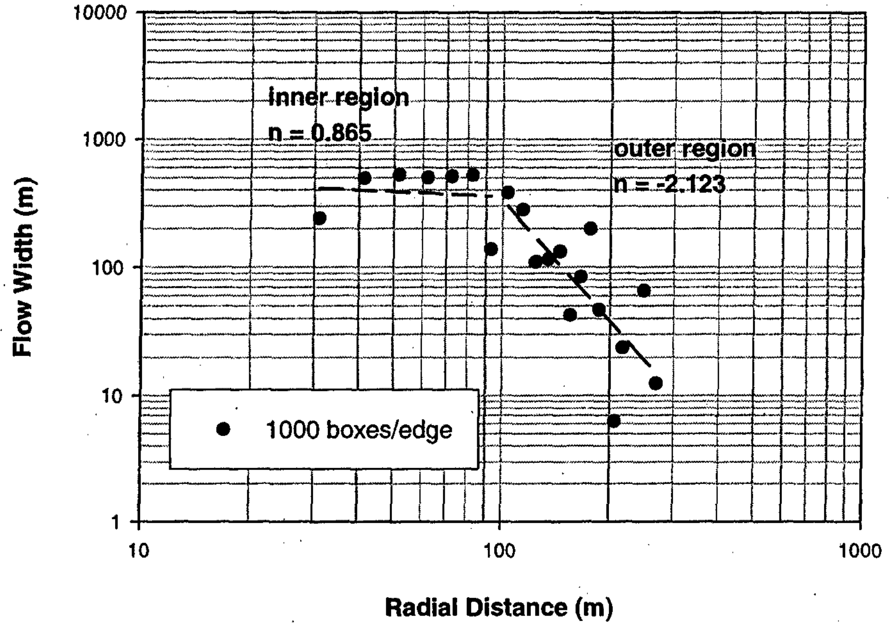

Distance-Flow Width Relationships
10000
1000
100
10
Flow Width (m)
inner region
n = 1.658
outer region
n = -0.702
● 20 boxes/edge
10
100
1000
Radial Distance (m)


Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
inner region
n = 1.673
outer region
n = -0.737
10 boxes/edge
10
100
1000
Radial Distance (m)

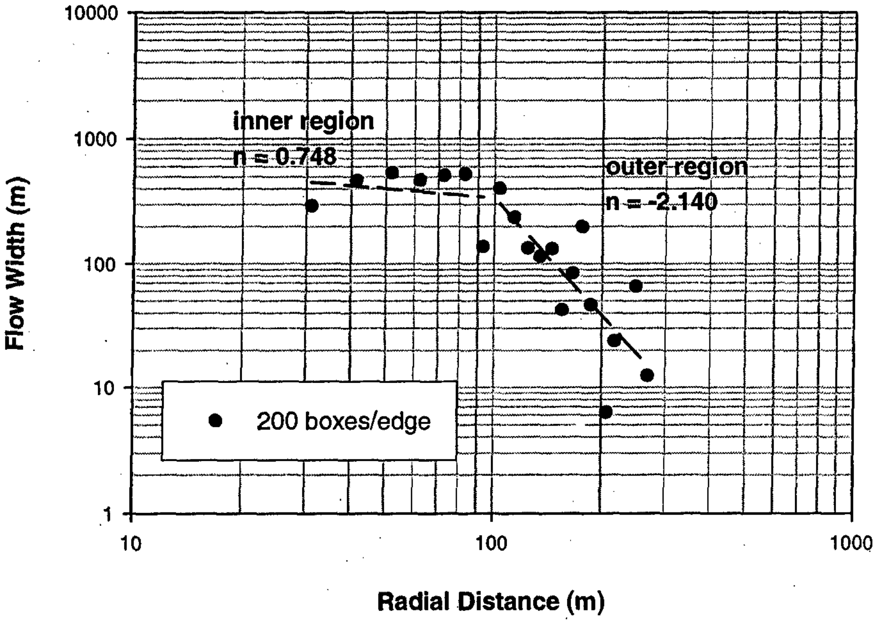

Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
inner region
n = 1.750
outer region
n = 0.111
5 boxes/edge
10
100
1000
Radial Distance (m)

Network 2

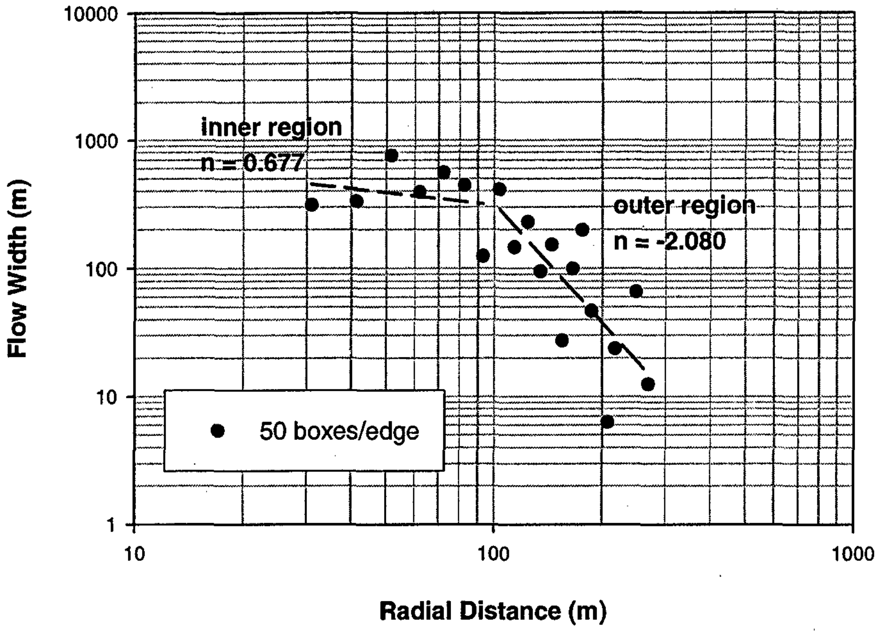
Distance-Flow Width Relationships



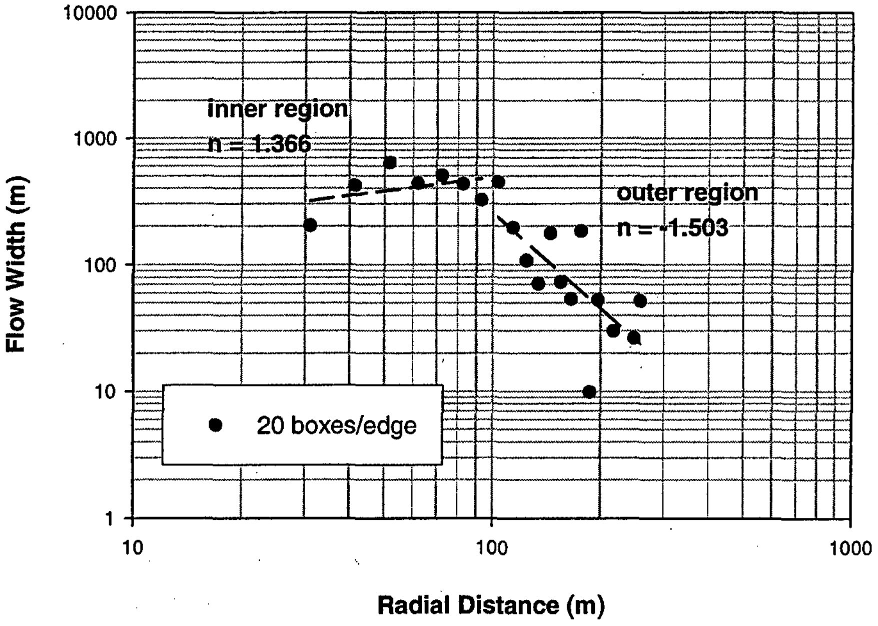
Distance-Flow Width Relationships

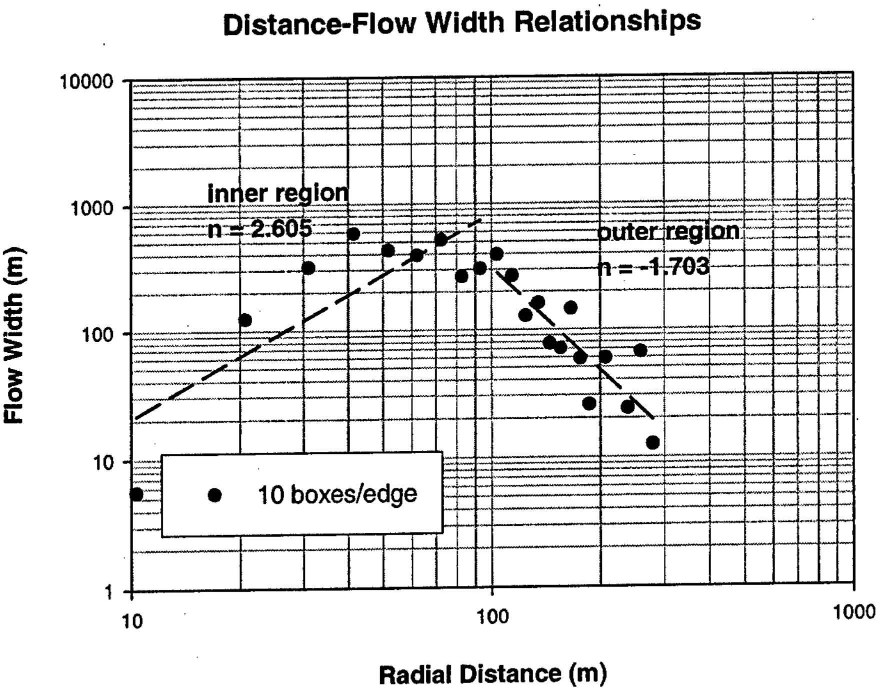


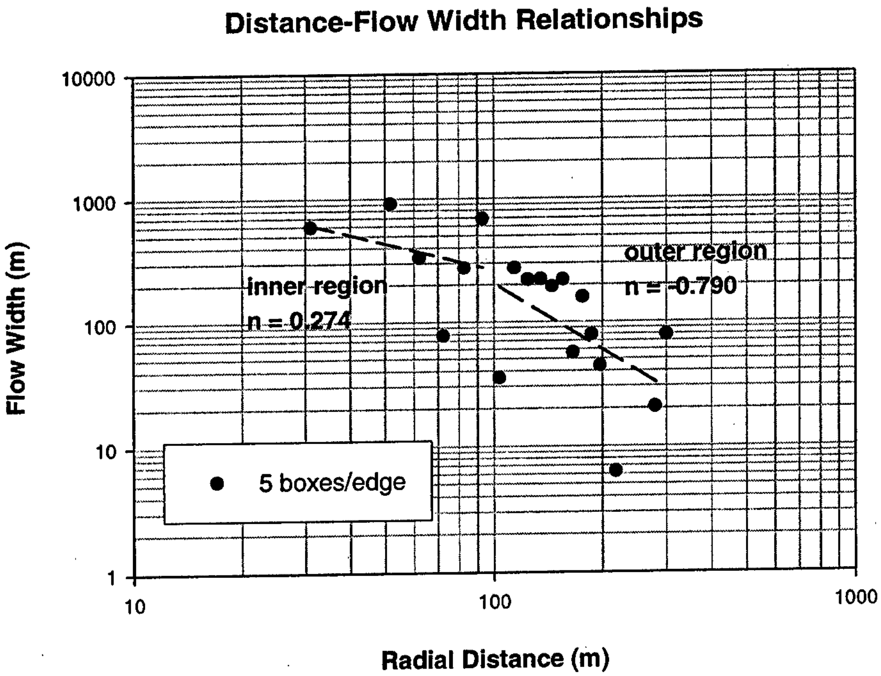
Distance-Flow Width Relationships



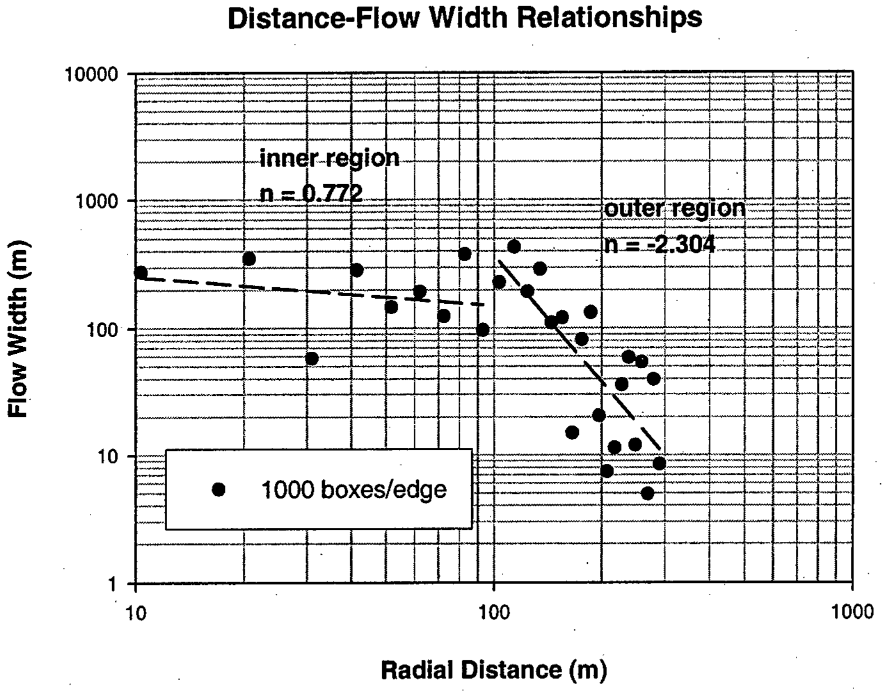
Distance-Flow Width Relationships

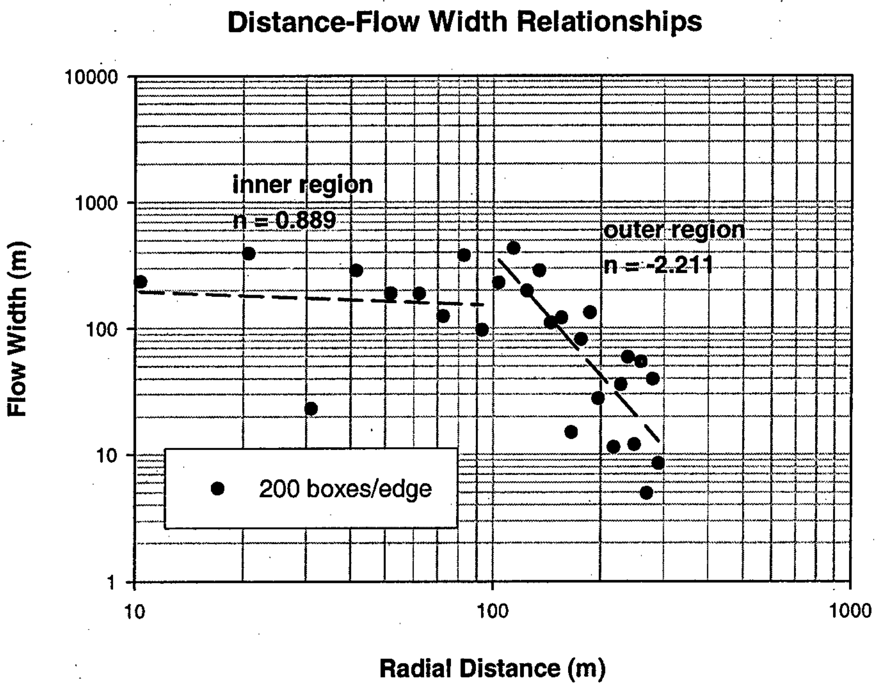


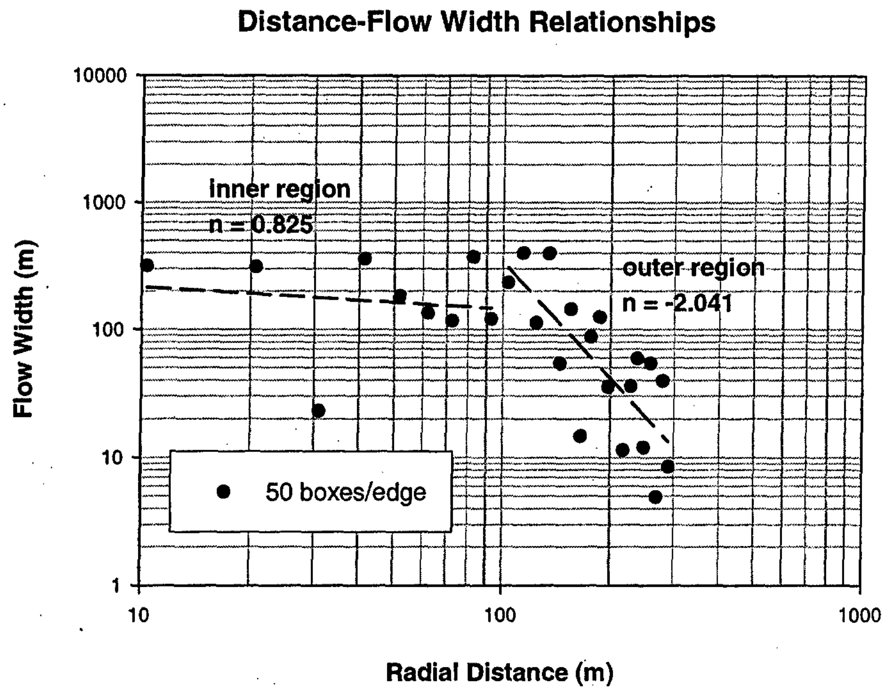

Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
inner region
n = 2.605
outer region
n = -1.703
● 10 boxes/edge
10
100
1000
Radial Distance (m)


Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
inner region
n = 0.274
outer region
n = -0.790
● 5 boxes/edge
10
100
1000
Radial Distance (m)

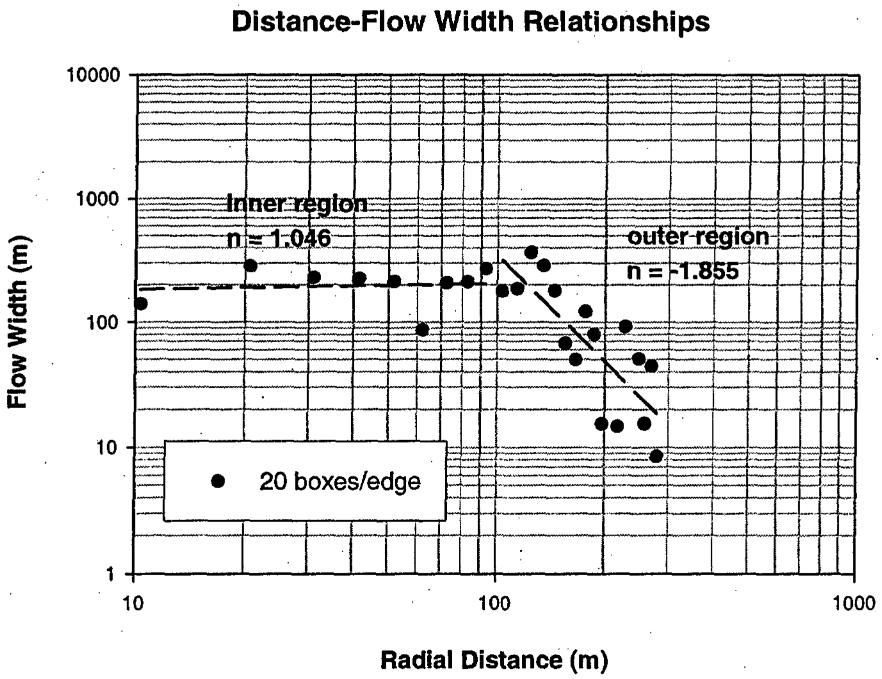
Network 3


Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 0.772
outer region
n = -2.304
● 1000 boxes/edge

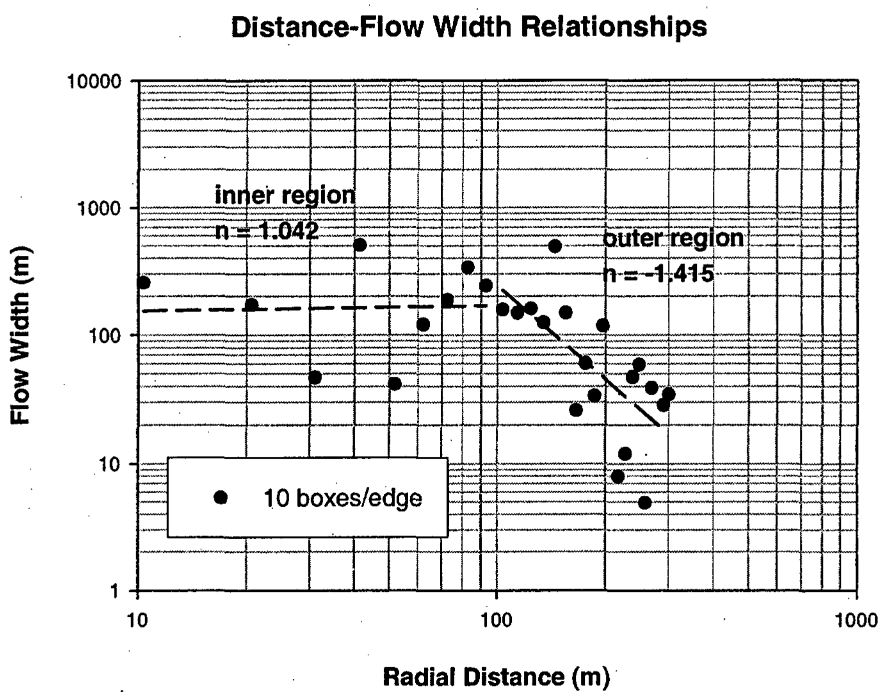

Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 0.889
outer region
n = -2.211
● 200 boxes/edge

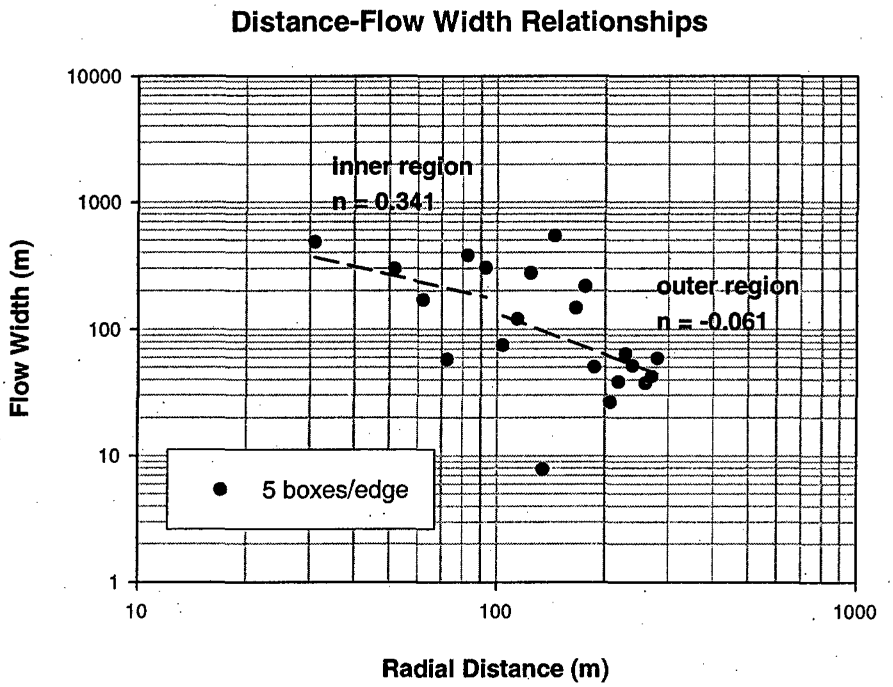


Distance-Flow Width Relationships

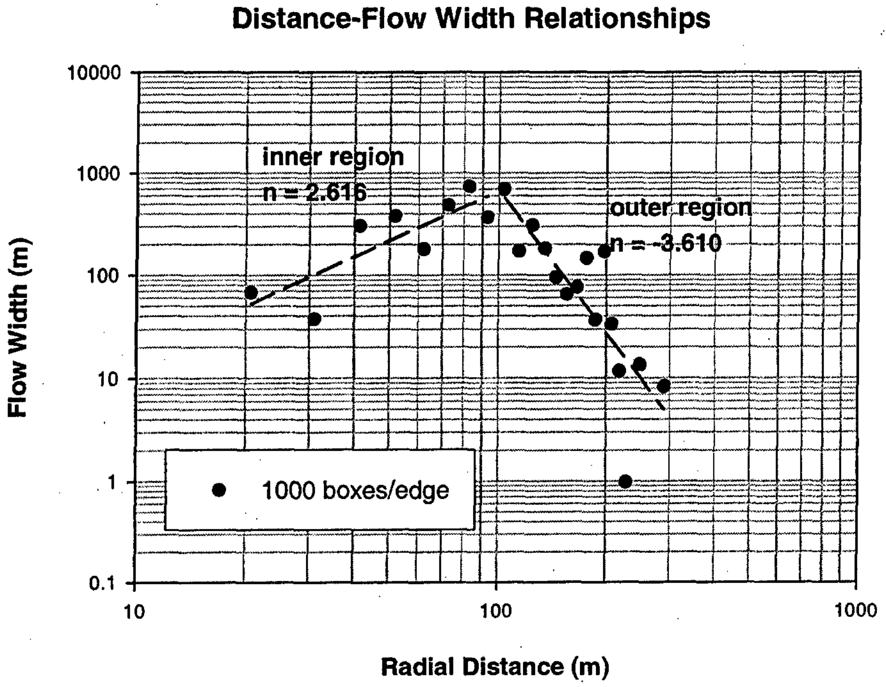


Distance-Flow Width Relationships

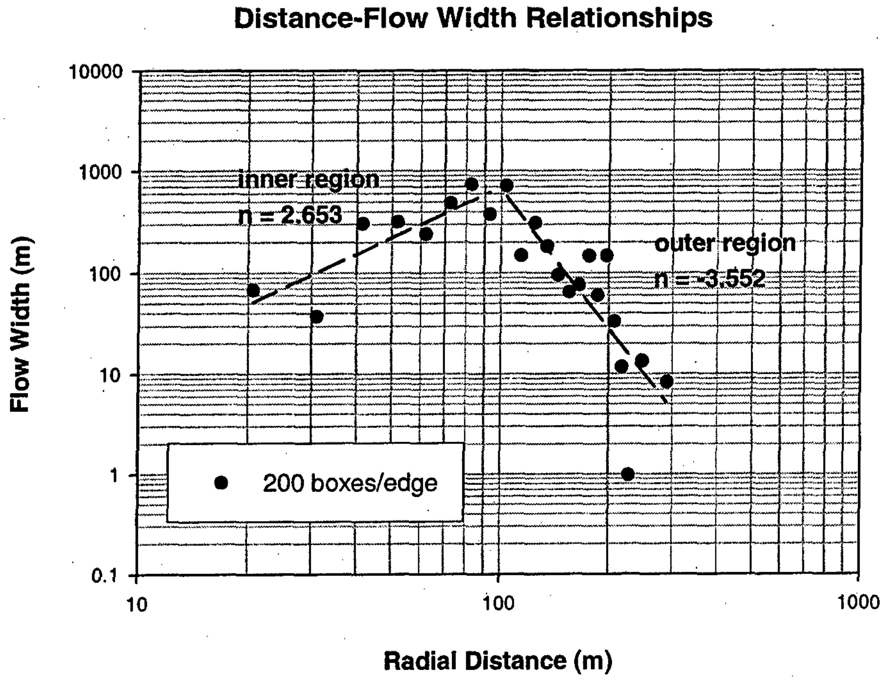

Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
inner region
n = 1.042
outer region
n = -1.415
● 10 boxes/edge
10
100
1000
Radial Distance (m)


Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
inner region
n = 0.341
outer region
n = -0.061
● 5 boxes/edge
10
100
1000
Radial Distance (m)

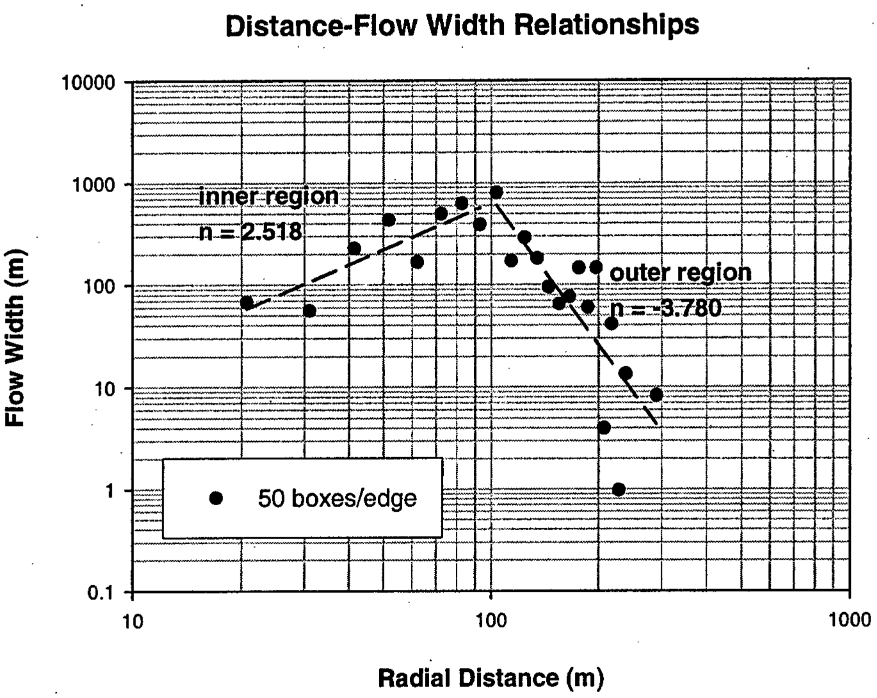
Network 4

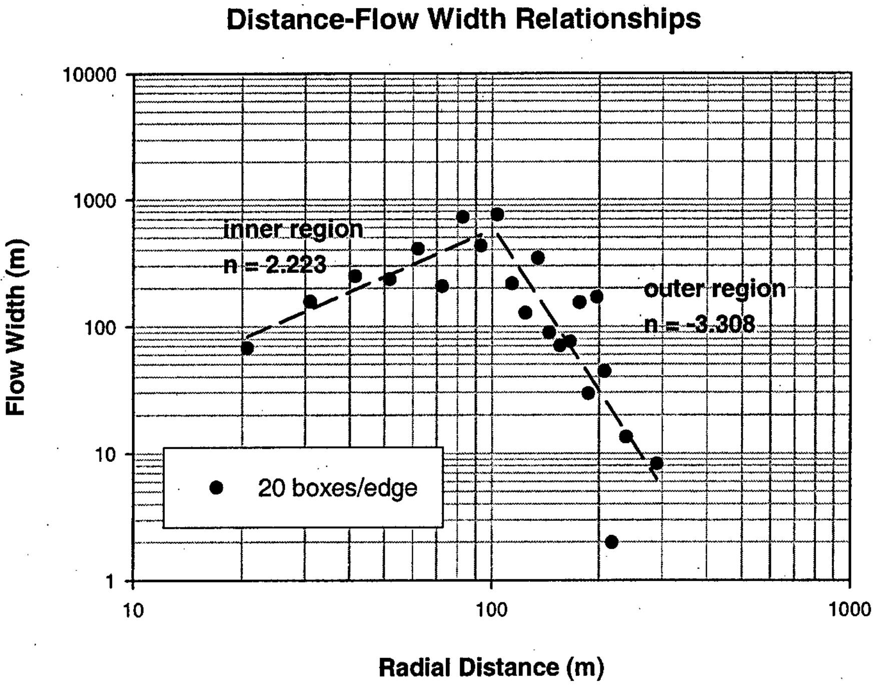


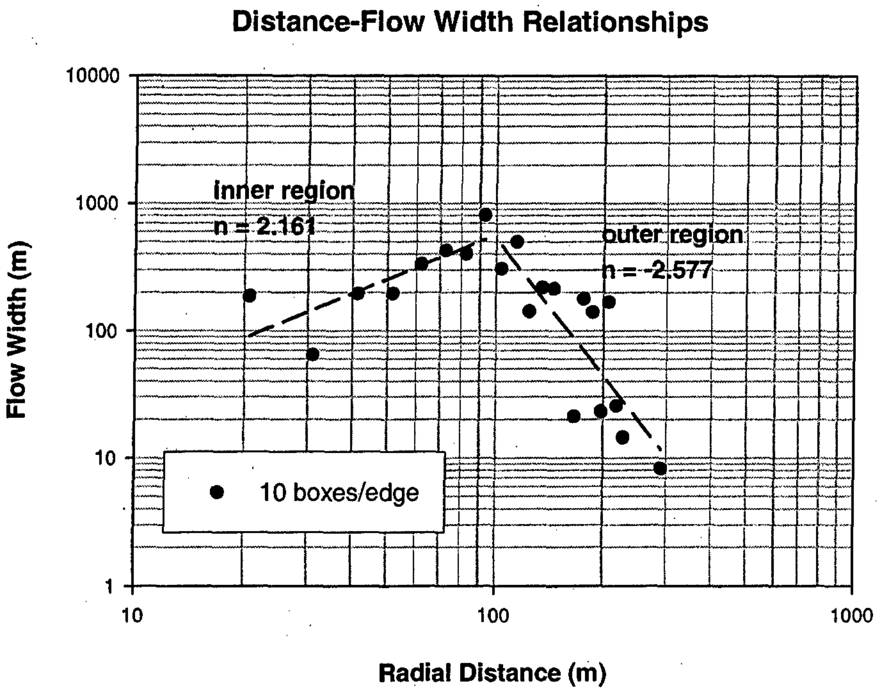
Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
0.1
10
100
1000
Radial Distance (m)
inner region
n = 2.616
outer region
n = -3.610
● 1000 boxes/edge

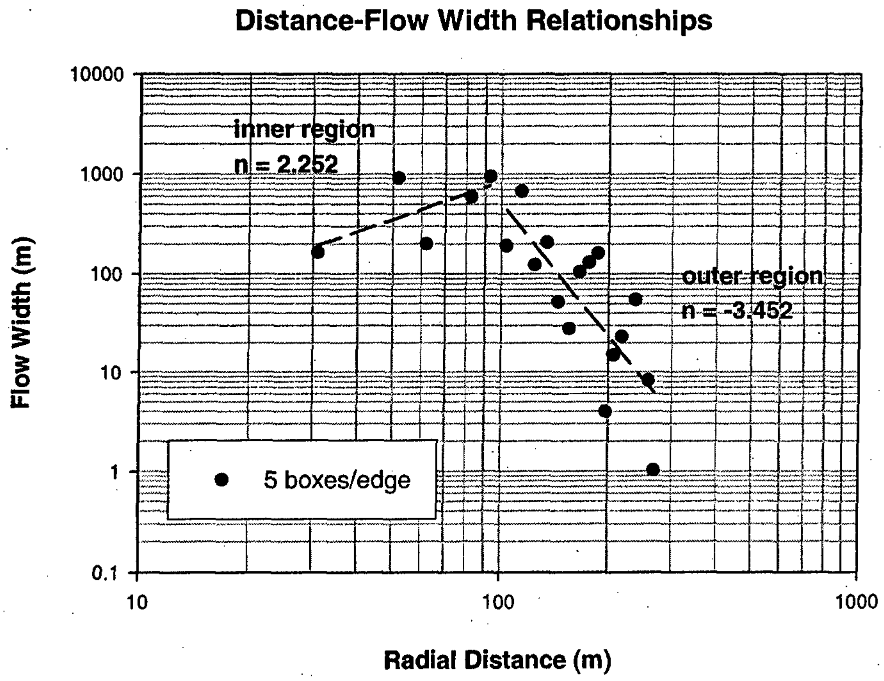


Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
0.1
10
100
1000
Radial Distance (m)
inner region
n = 2.653
outer region
n = -3.552
● 200 boxes/edge

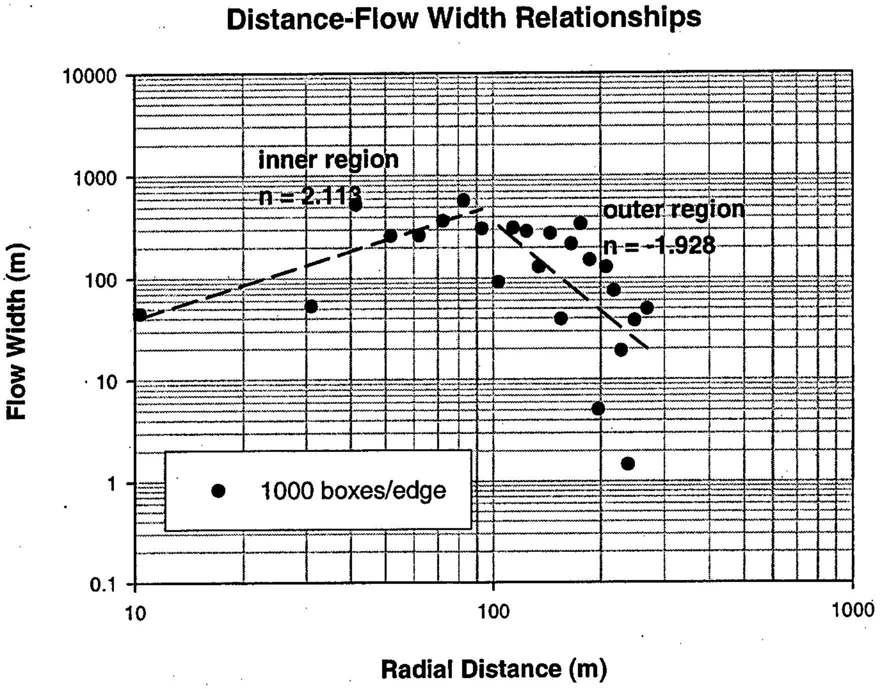

Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
0.1
10
100
1000
Radial Distance (m)
inner region
n = 2.518
outer region
n = -3.780
● 50 boxes/edge

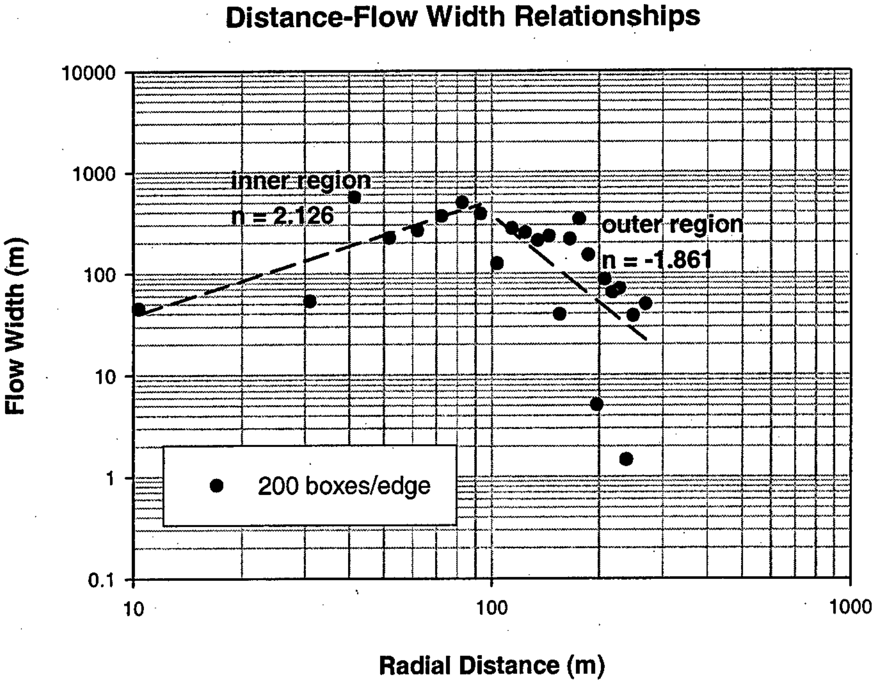

Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
10
100
1000
Radial Distance (m)
inner region
n = 2.223
outer region
n = -3.308
● 20 boxes/edge

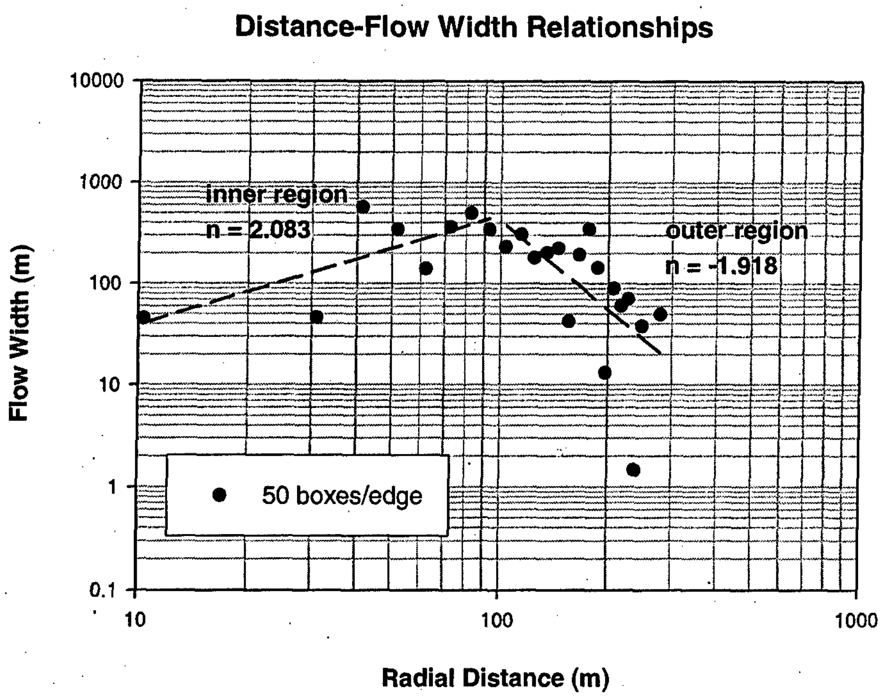

Distance-Flow Width Relationships
10000
1000
100
10
1
Flow Width (m)
inner region
n = 2.161
outer region
n = -2.577
● 10 boxes/edge
10
100
1000
Radial Distance (m)

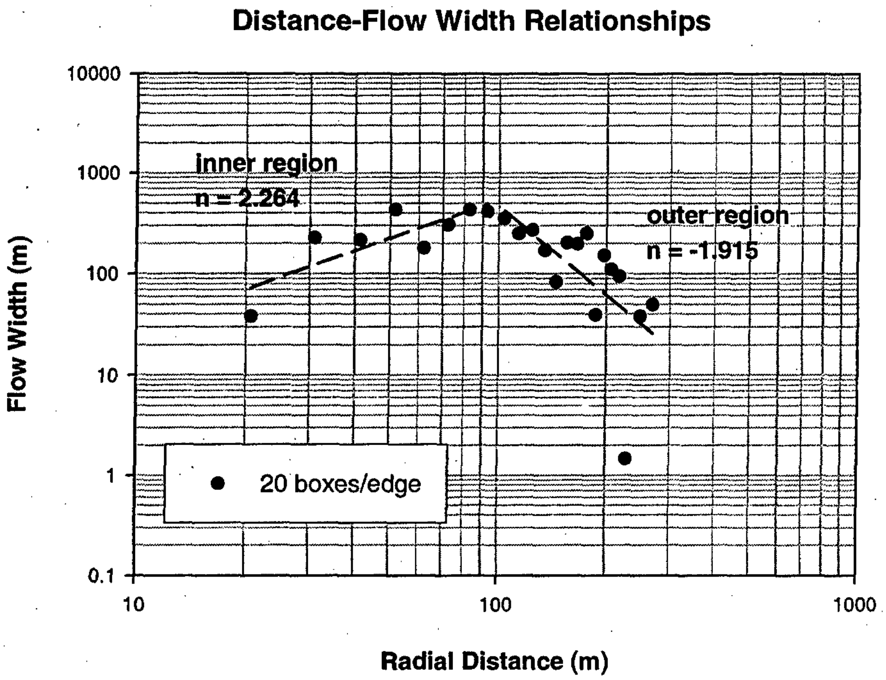

Distance-Flow Width Relationships
10000
1000
100
10
1
0.1
Flow Width (m)
inner region
n = 2.252
outer region
n = -3.452
● 5 boxes/edge
10
100
1000
Radial Distance (m)

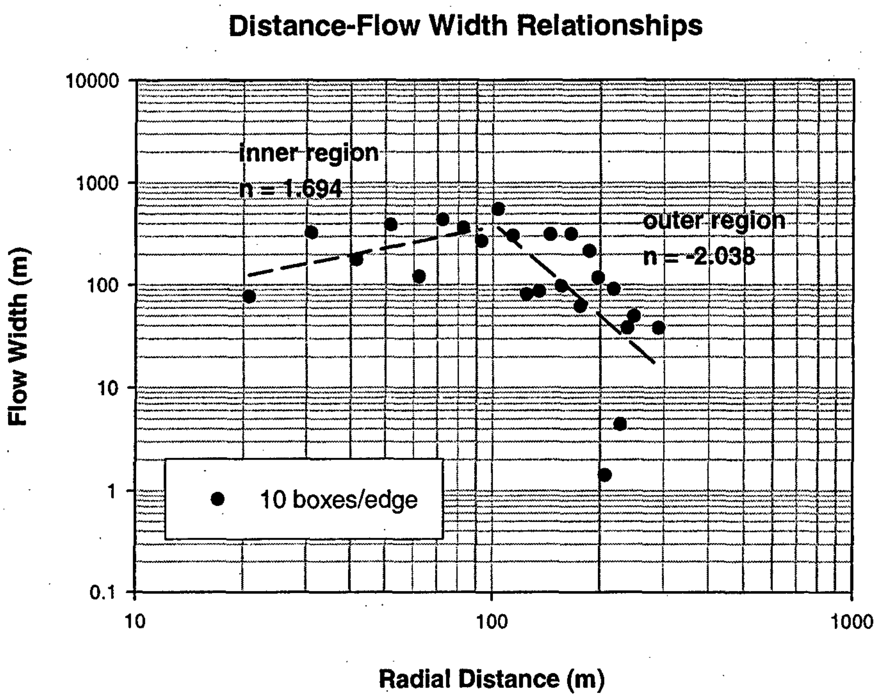
Network 5

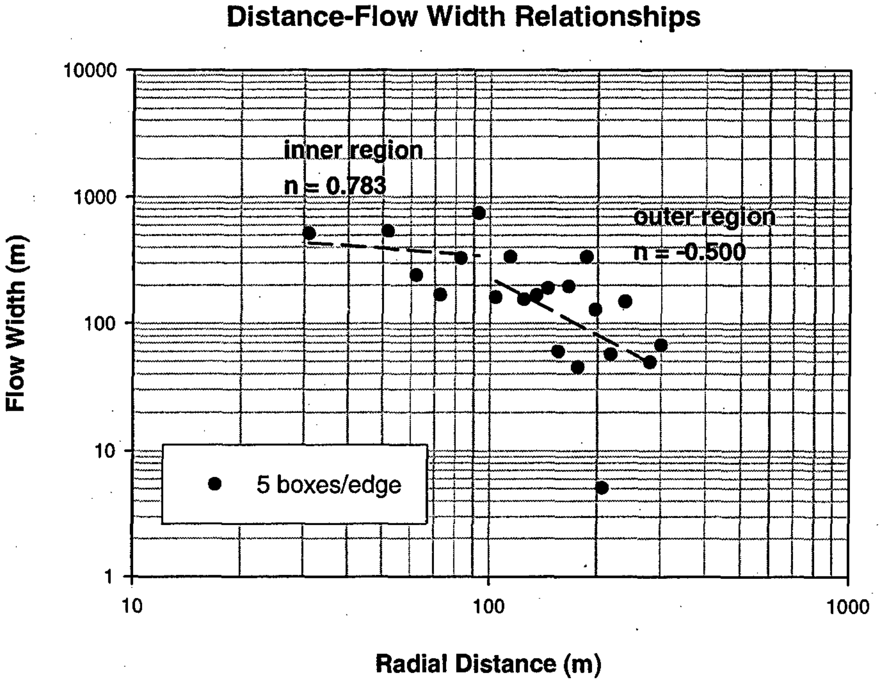

Distance-Flow Width Relationships
10000
1000
100
10
1
0.1
Flow Width (m)
inner region
n = 2.113
outer region
n = -1.928
● 1000 boxes/edge
10
100
1000
Radial Distance (m)


Distance-Flow Width Relationships
10000
1000
100
10
1
0.1
Flow Width (m)
inner region
n = 2.126
outer region
n = -1.861
● 200 boxes/edge
10
100
1000
Radial Distance (m)


Distance-Flow Width Relationships
10000
1000
100
10
1
0.1
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 2.083
outer region
n = -1.918
● 50 boxes/edge


Distance-Flow Width Relationships
10000
1000
100
10
1
0.1
Flow Width (m)
10
100
1000
Radial Distance (m)
inner region
n = 2.264
outer region
n = -1.915
● 20 boxes/edge


Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
0.1
10
100
1000
Radial Distance (m)
inner region
n = 1.694
outer region
n = -2.038
● 10 boxes/edge


Distance-Flow Width Relationships
Flow Width (m)
10000
1000
100
10
1
10
100
1000
Radial Distance (m)
inner region
n = 0.783
outer region
n = -0.500
● 5 boxes/edge