

Preliminary Report

on

Fractal Modeling of Fractures
in
Tight Gas Reservoirs

supported by

The Department of Energy

Grant #DE-FG21-94MC31182

staff

Martin V. Ferer - Professor of Physics

Bruce H. Dean - Physics Ph.D. Student

Charles Mick - Physics Undergraduate Senior

Foreword

This preliminary report from the results of Task 1 is grouped into three sections:

- I) an introduction
- II A) box-counting
 - i) a discussion of the general procedure and advantages
 - ii) results for the MWX fracture network
- B) evidence from the literature that real fracture networks are fractal
- III) a description of the computer codes for generating simulated fracture networks.

I Introduction

Recovery from naturally fractured, tight-gas reservoirs is controlled by the fracture network.[1] Reliable characterization of the actual fracture network in the reservoir is severely limited. The location and orientation of fractures intersecting the borehole can be determined, but the length of these fractures cannot be unambiguously determined. Fracture networks can be determined for outcrops, but there is little reason to believe that the network in the reservoir should be identical because of the differences in stresses and history. Seismic techniques do provide some large scale (resolution of tens or hundreds of feet) information about the fracture density and average fracture orientation, although there is some controversy about interpretation of the multi-component surface seismic data, especially regarding which layer is being probed.

At the very least, our assumption of fractal behavior is a good approximation to the real data for the MWX site, and the assumption of fractal behavior enables us to produce, a real visual similarity to the actual clustering found in the MWX fracture distribution in a very routine (i.e. easily programmable) × fashion.

Furthermore, independent of the assumption of fractal behavior, it is known that typical fractures ✓ in the second set should begin and end at fractures of the first set.[2] This effect is commonly observed in real fracture networks from outcrop studies, for example 92% of the secondary fractures in the MWX outcrop satisfy this criterion.[3] We have imposed this constraint upon our secondary fractures which ✓ increases the visual similarity between our networks and the real network over simulated networks from other fractal modeling schemes. [4]

I Introduction

1.A Modeling the Fracture Network

Because of the lack of detailed information about the actual fracture network, modeling methods must represent the porosity and permeability associated with the fracture network, as accurately as possible with very little apriori information. Three rather different types of approaches have been used: i)

dual porosity simulations, ii)'stochastic' modeling of fracture networks, and iii) fractal modeling of fracture networks. The dual porosity approach is a natural extension of the gridding schemes widely used in describing reservoirs, however in assuming mesoscopic scale (tens or hundreds of feet) averages of fracture porosities and permeabilities, they may be smoothing the very heterogeneities which control the recovery, which may limit reliability for strongly anisotropic fracturing. That is, even if fractures are located randomly throughout the grid-block so that an average porosity may be sensible, the conductivity of similar fractures differ widely invalidating assumptions of an average permeability.

Stochastic models which assume a variety of probability distributions of fracture characteristics have been used with some success in modeling fracture networks. [5], [6], [7] The advantage of these stochastic models over the dual porosity simulations is that real fracture heterogeneities are included in the modeling process. On the other hand these stochastic models need information about all features of the actual fracture network to provide the most accurate modeling. In the highest level (most accurate) model for each set of fractures with a given orientation, one needs to determine the probability distribution of i) the location of independent fractures ii) the location of fracture clusters or swarms iii) locations of fractures within clusters, iv) cluster lengths, v) fracture lengths, vi) fracture apertures, and vii) fracture orientations. The less reliable the information determining these probability distributions; the less reliable the fracture network. Reliable information about many aspects of the real fracture network is impossible to determine; the assumption of self-similar fractal behavior (if valid) enables us to predict features of one aspect of the distribution from other aspects of the distribution; i.e. i), ii), and iii) result from the box-counting along the borehole which, in turn, predicts features of the distributions for iv), v), and vi) for self-similar fractal networks.

L B) Introduction - Advantages of Fractal Modeling

Aspects of fractal geometry have been applied to mimic the heterogeneity associated with layering in real reservoirs for a number of years with some success. In these cases, the variation in permeability with height at the borehole was found to obey fractal statistics,[8] and the correlations implicit in fractal geometries allowed them to interpolate between the known permeabilities at the borehole in such a way that results from flow models agreed with analyses of production logs and tracer breakthrough.[9] Examples in the open literature reporting the use of fractal geostatistics to treat naturally fractured reservoirs are less common.[10], [4] If a set of natural fractures is described by a self-similar fractal geometry, the self-similar, scale invariance of the fracture network implies relationships among the number of fractures, and the various length scales: fracture correlation or clustering, fracture aperture, and fracture length. Therefore, if fracture networks obey a self-similar fractal geometry, borehole data locating orientational sets of fractures, will enable a determination of the fractal dimension and 'lacunarity'. This along with relatively generic information about the typical aperture size and length

of fractures,[1] will allow us to produce a self-similar fractal network. The clustering occurs naturally in the fractal network because of the correlations inherent in fractal geometries. The fractal parts of the aperture size and length distributions (even the fracture shape distributions) should be the same as the fractal parts of the fracture location vs. scale distributions.

In the sections following this introduction(I), we will II.A) present 'fractal' analysis of the MWX site, using the box-counting procedure[11], [12] , II.B) review available evidence testing the fractal nature of fracture distributions and discuss the advantages of using data from our 'fractal' analysis over data from a stochastic analysis III) present an efficient algorithm for producing a self-similar fracture network which mimics the real MWX outcrop fracture network.

II Fractal Analysis of the MWX Site, Outcrop Fracture

Before discussing our analysis of the MWX site (Fig. 1), it is important to understand the box-counting procedure used in these tests as well as in our method for generating the fracture networks. As discussed in the conclusion of this section (II), this box-counting procedure automatically reproduces the random aspects of the distribution of fractures in addition to reproducing the clustering obvious in Fig. 1.

II.A) i) An Example of the Box-counting Procedure

Fig. 2 shows fractures of one orientation intersecting a length of borehole. To determine the fractal dimension as well as the range of size scales over which the distribution is fractal, one covers the array of fractures by successively smaller and smaller rulers (one-dimensional 'boxes'), and then one counts the number of 'boxes' or rulers covering one or more fractures. If the distribution has a fractal dimension D_f over a range of sizes, then

$$\# \text{ of boxes containing fractures} = A (\text{size of boxes})^{D_f},$$

where the constant A is called the "lacunarity." Specifically, if one covers the 24 fractures by a ruler of length L , (shown at the bottom of Fig. 2) one ruler covers the fractures; with two ruler of length $L/2$ (near the bottom of Fig. 2) both cover fractures; with four rulers of length $L/4$ all 4 cover fractures, but with 8 rulers of length $L/8$ only 6 cover fractures. This is continued down to 128 rulers of length $L/128$ as shown in Table 1.

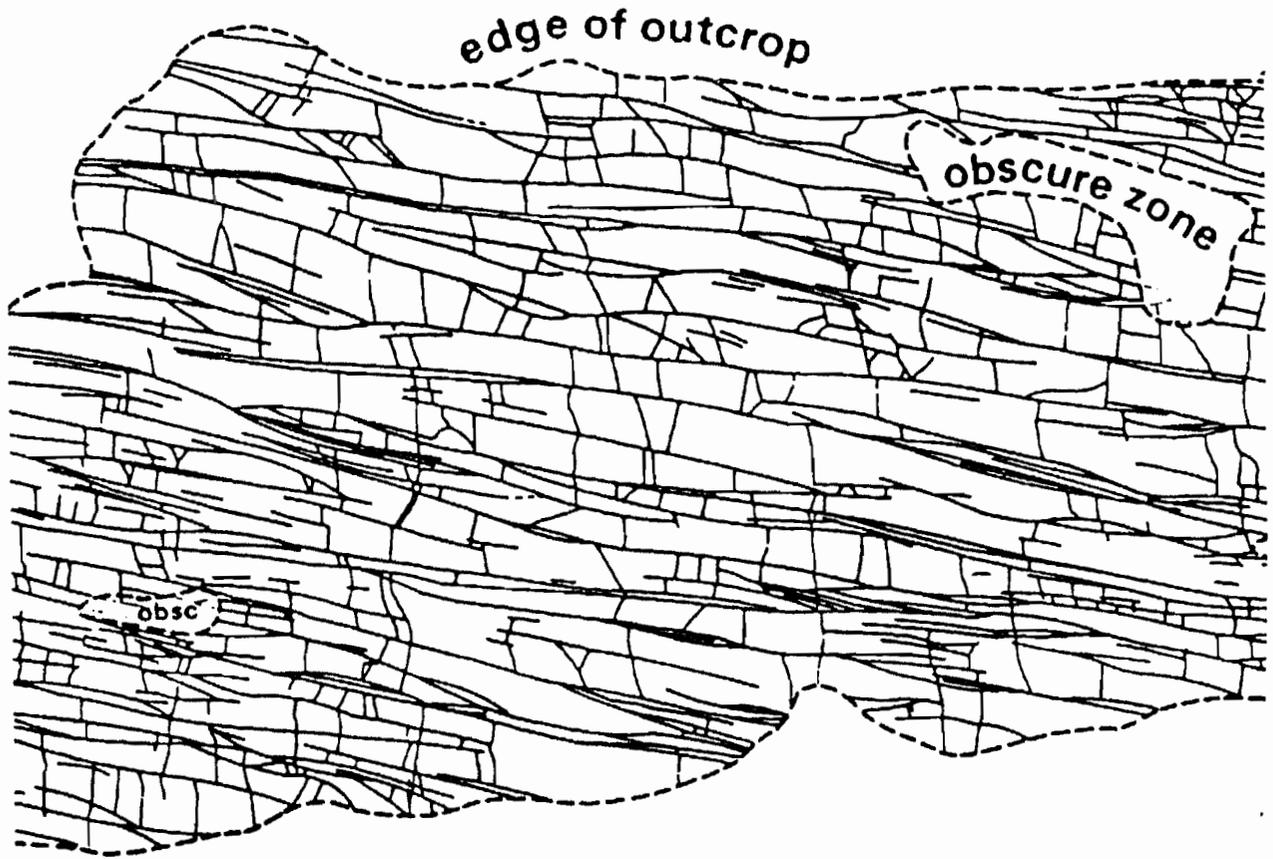


Fig. 1 Outcrop Fracture Network at MWX site.

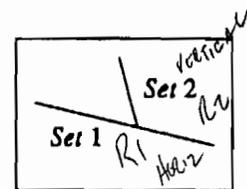




Fig. 2a Fractures in a particular orientational set which intersect the borehole.

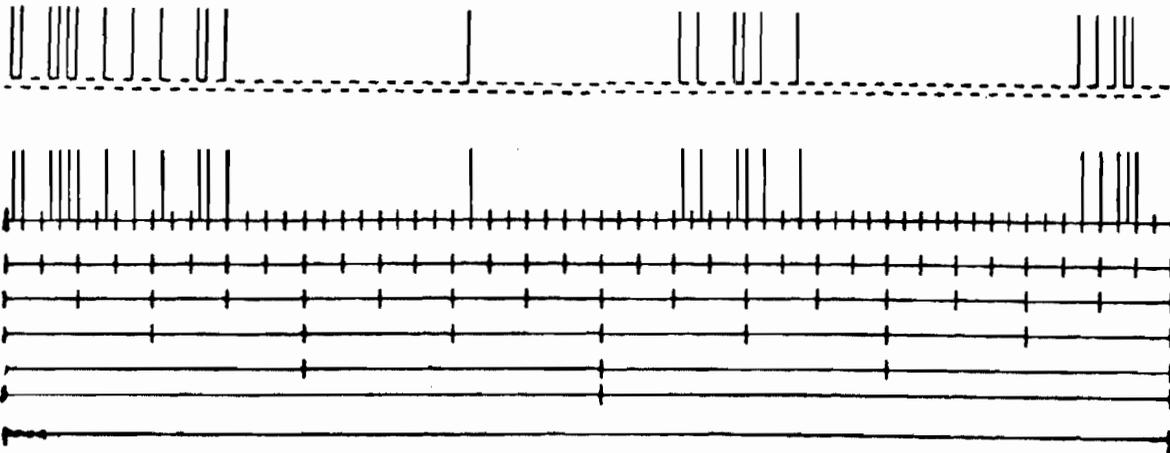


Fig. 2b The lower half of the figure shows the fractures in Fig. 1 with the scale rulers 'covering' the set of fractures from a ruler of length L , proceeding upwards to rulers of length $L/64$ just below the fractures. The top half shows the same set of 'covering' rulers of length $L/128$. The rulers are left-justified so that the fractures at the right-end of the ruler are covered by the ruler.

Length of Rulers ($\frac{L}{\Delta}$)	Number of Rulers Covering Fractures (N)
L	1
$L/2$	2
$L/4$	4
$L/8$	6
$L/16$	8
$L/32$	13
$L/64$	17
$L/128$	24
$L/256$	24
$L/1024$	24

Table 1

Since there are only 24 fractures, at scales smaller than $L/128$, there will only be 24 rulers covering fractures. A log-log plot of the box-counting for Fig. 2 is shown below

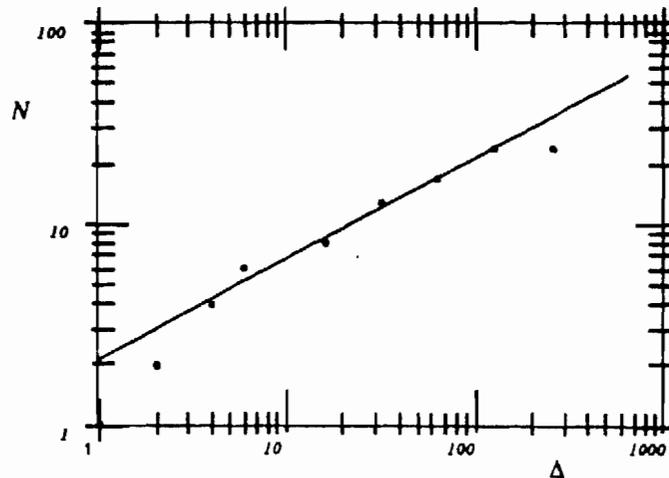


Fig. 3 Fractal Plot for Fig. 2.

The fractal relationship is given by the solid line $N = (2.12) \Delta^{0.5}$, except at large and small scales for the reasons that follow. At small Δ , (coarse scales L , $L/2$ and $L/4$), N equals the number of rulers ($N = \Delta$) because all the rulers cover fractures; in later sections, we refer to this as the initial covering. At very large Δ (very fine scales $L/256$ and $L/1024$), only 24 rulers are covered because there are only 24 fractures and there is no more detail in the fracture pattern, so that the box-counting 'cuts-off' or 'saturates' at 24. Therefore, for this fracture pattern, the pattern is fractal between the initial covering and cutoff regimes (over the range of scales $\Delta = 8$ to 128) with a fractal dimension of 1.5 and a lacunarity of 2.12. The ruler counting gives an exponent $D_f - 1$, i.e. the 2d fractal dimension minus one.

Before we continue it should be pointed out that this fracture pattern was generated by our algorithm to have a lacunarity of 2.12 and a fractal dimension of 0.5 over the range of scales from $L/8$ to $L/128$. The algorithm which generated this pattern is described and used in section III.

It is important to realize that if the distribution of fractures in Fig. 2 were completely random (i.e., if there were no clustering of fractures), the points from the box-counting would obey a linear relationship ($N = \Delta$) up to cutoff. That is, on the average, each box would contain one fracture up to the total number of fractures (in this case $N_{total} = 24$); at finer scales, the one fracture would randomly occupy one of the smaller boxes. However, because of clustering groups of fractures are much closer together than average. Therefore, when box-counting, the linear regime ends before $N = N_{total}$; and one enters the 'fractal' or clustering regime where some boxes are empty and others have several fractures much closer together than average. The box-counting provides a routine procedure for characterizing (and, thus, for reproducing) this clustering.

IIA ii) Box-counting Results for the MWX Network

First the primary set of fractures (the horizontal fractures in Fig. 1 were analyzed. A series of eight lines (boreholes) of length L (the length was 16 cm on a figure enlarged by 141%) were drawn through the set of primary fractures, and the box-counting procedure was used on each of these boreholes. The results for the number of boxes covering fractures vs. the scale Δ is shown in Fig. 4. The initial covering regime persisted until scale 16. The cutoff regime began at scale 80. In-between the data is well represented by the fractal power law $N = 4.9 \Delta^{0.425}$, indicating a fractal dimension $D_f = 1.43$. It should be noted that scales intermediate to the simple doubling rule, $\Delta = 2^n$, (used in Figs. 2 & 3 and Table 1) were used to provide more data in the fractal regime.

The secondary fractures (the vertical fractures of Fig. 1) were analyzed in the same way. A series of horizontal lines of length L were drawn through the secondary fractures, the box counting was performed and the values $N(\Delta)$ were averaged.

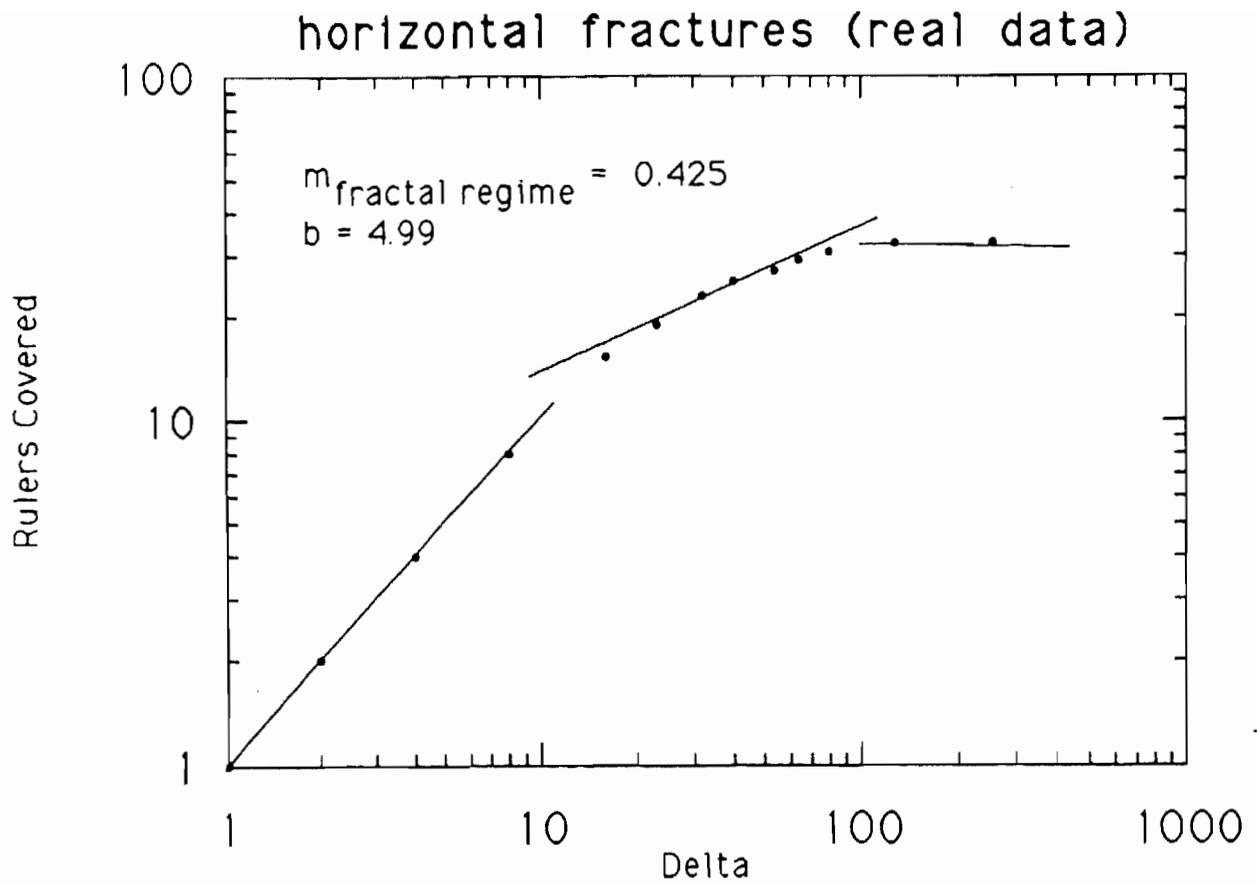


Fig 4. For the primary (horizontal) fractures, the box-counting from the 'boreholes' on the MWX outcrop (Fig. 1), shows the initial covering (the characteristic linear increase, $N = \Delta$, up to the clustering or fractal regime), the fractal regime, and the cutoff regime.

Fig. 5 shows the plot of N vs. Δ and shows that for these secondary fractures the initial covering regime persists until scale 6 and that the cutoff regime begins at scale 40. In-between the number of boxes obey the fractal power law $N = 3.47 \Delta^{0.346}$, indicating a fractal dimension $D_f = 1.34$. Again, intermediate scales were used in the fractal regime to provide more data in the fractal regime.

To determine the length distribution from the data provided by M. McKoy, we plotted the total number of fractures with lengths greater than a given length L , $N(L)$, vs. L . It should be noted that this total number $N(L)$ with lengths $l > L$ is the integral of the number density of fractures $n(l)$ with length l integrated from $l = L$ up to the one fracture of maximum length l_{\max} : $N(L) = \int_L^{l_{\max}} n(l) dl$. This graph of the data is shown in Fig. 6. It is convincingly fit by the characteristic exponential cutoff for the greatest lengths ($L > 14$), and by a fractal power law for the smallest lengths ($4 < L < 14$). For a self-similar fractal fracture network, the number density should be given by as $n(l) = \eta l^{-D_f}$ so that the total number should be given by $N(L) = \left(\frac{\eta}{1-D_f} \right) l^{1-D_f}$. [11] Therefore, that data is consistent with a fractal dimension $D_f = 1.48$.

This data does not decide unambiguously whether or not the clustering/fractal regime is rigorously fractal. That is, this data does not unambiguously favor a strictly power law regime (i.e. fractal behavior) between the linear, initial covering regime and cutoff. However, the power law assumption used to draw the lines does represent a good fit to the box-counting data. Therefore, at worst, by assuming that the intermediate regime is fractal, we may be providing merely good approximation to the data. If the assumption of fractal clustering only provides a good approximation to the true clustering, our simulated fracture networks will represent a good approximation to the actual fracture network; this is all that is necessary.

On the other hand, it is encouraging that the power laws from the box counting and length distributions are all consistent with the same fractal dimension, $D_f = 1.4 \pm 0.1$, to within a realistic accuracy from the data fitting. This equality of fractal dimensions from all length measures is the hallmark of self-similar fracture networks.

II.A iii) Programs for Box-counting of Borehole Fractures

A program to carry out the box-counting procedure and return the fractal dimension and lacunarity has been developed in order to process multiple sets of data from various boreholes. To test these programs as well as the routines for simulating the fracture networks, numerous trial runs have been performed to analyze the "borehole fractures" from simulated networks. For example, Fig. 7 shows the results from the box counting program analysis of a simulated fracture set which was generated assuming a fractal dimension of 1.5 and a lacunarity of 2.12.

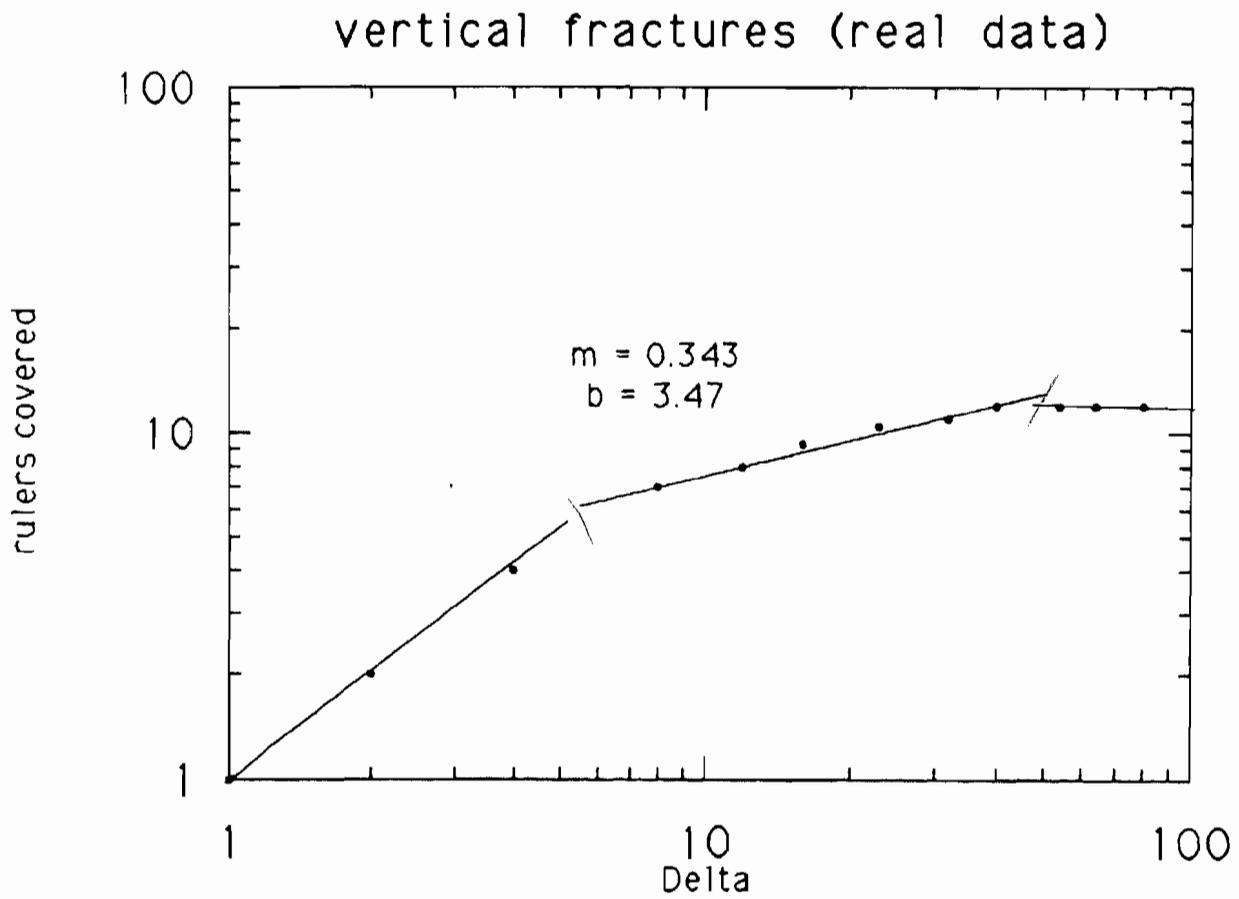


Fig. 5. For the secondary (vertical) fractures, the box-counting from the 'boreholes' on the MWX outcrop (Fig. 1), shows the initial covering (the characteristic linear increase, $N = \Delta$, up to the clustering or fractal regime), the fractal regime, and the cutoff regime.

- $L = 4 \rightarrow 14$
- * $L = 14 \rightarrow 96$

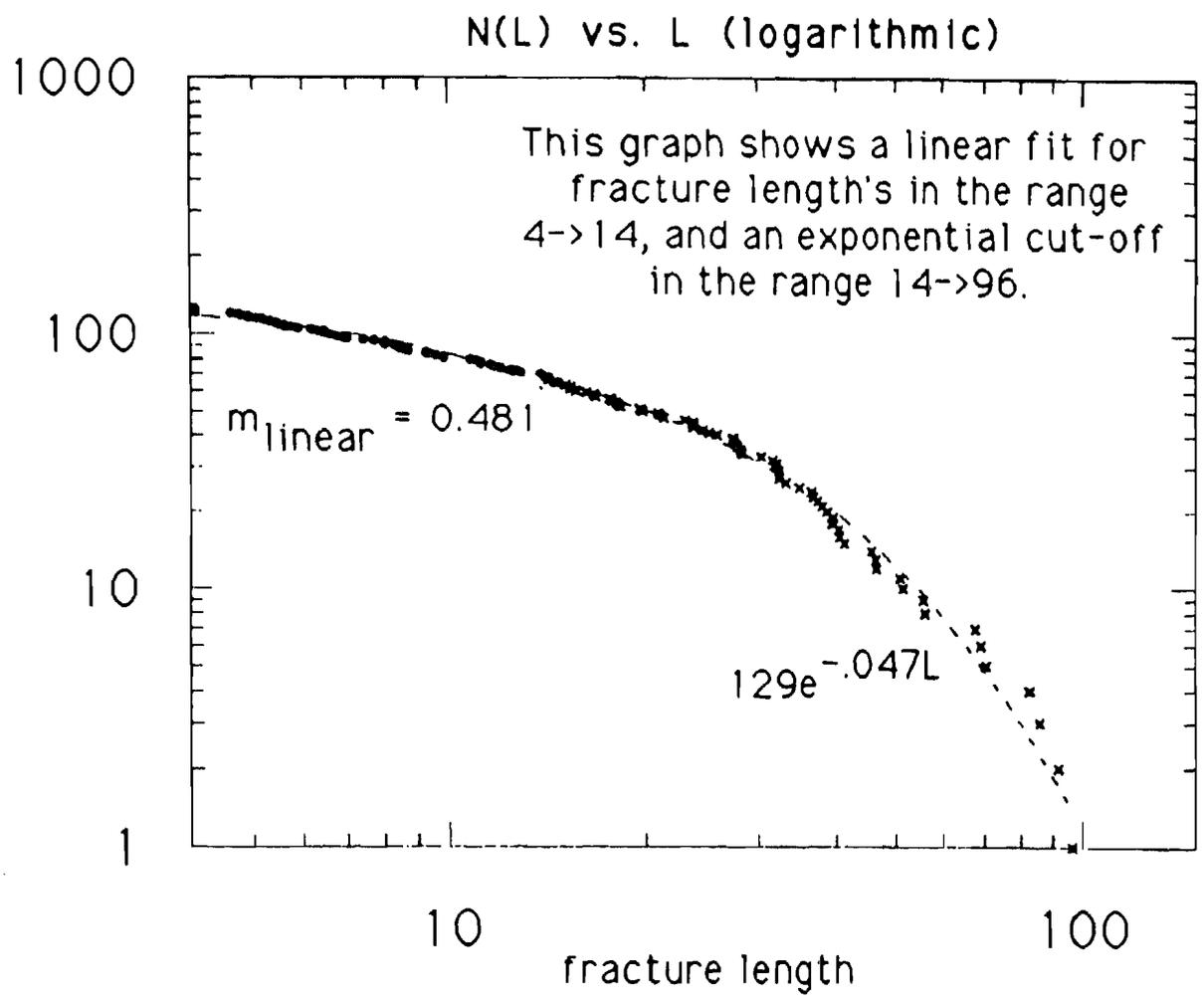


Fig. 6. The number of fractures $N(L)$ with lengths greater than L plotted against L . This shows the exponential cutoff for the larger lengths and the fractal regime for the smaller lengths.

- Rulers Covered

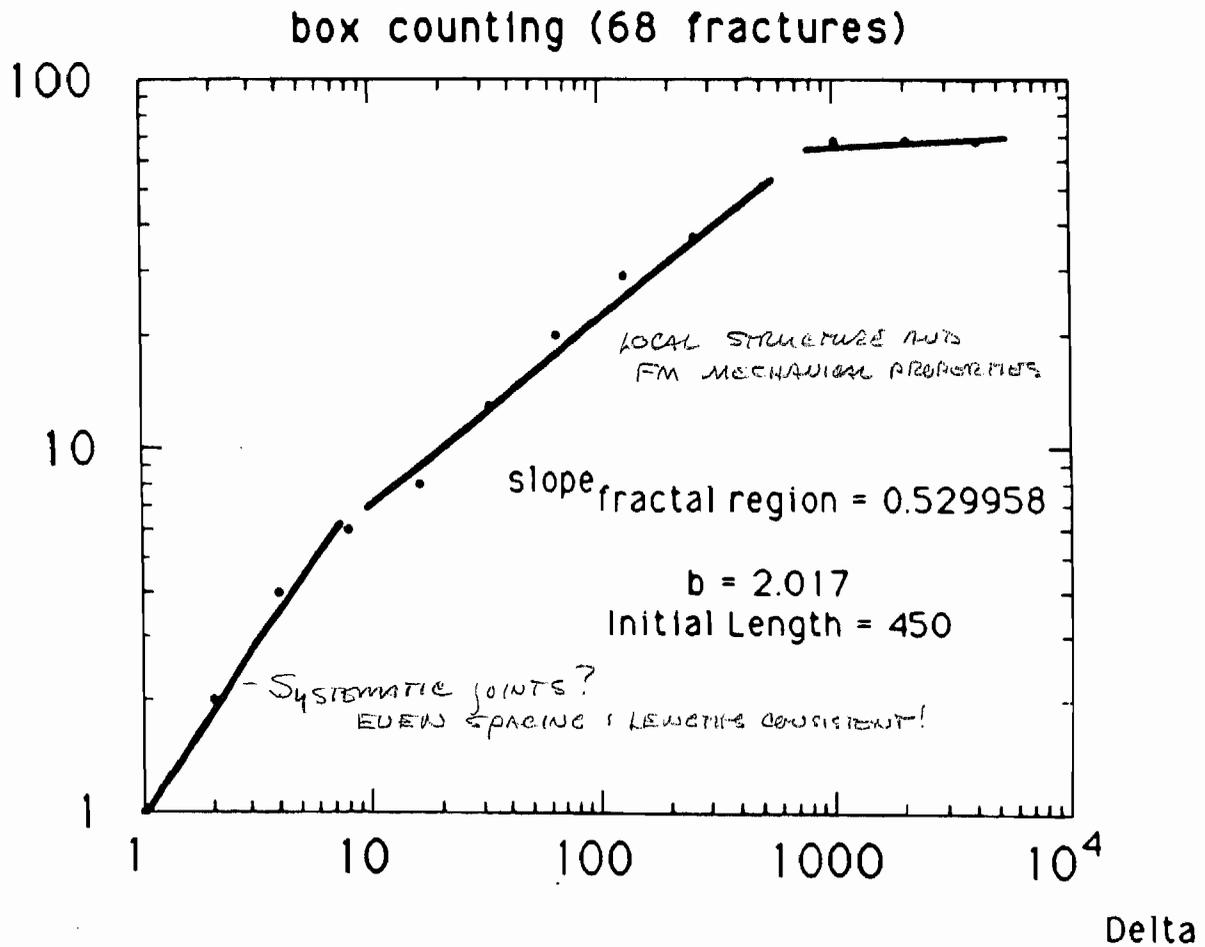


Fig. 7. Box counting results from a simulated borehole.

The least squares analysis of the box-counting results yielded a fractal dimension of 1.52 ± 0.015 and a lacunarity of 2.02 ± 0.121 indicating a consistency within the various aspects of the project.

The programs for the borehole analysis are presented in Appendix A. The driver program serves the function of reading in data from a particular fracture set (section I in source code). The data was assumed to consist of a single column of numbers giving the location of each fracture from an origin of zero. The user is asked to input an initial ruler length which gives the length of the borehole. Control is then passed to the 'box-count' function (section II). This 'box-count' function is the heart of the program; it returns the number of rulers being covered and the size of these rulers. When this data has been obtained, the fractal regime is isolated from the initial covering and cutoff regimes in section III and is sent to a least squares analysis function in section IV. The values sent to the least squares analysis function are the log values of the original data from the fractal regime. The least squares function determines the slope and y-intercept of the best straight line through all the points. The standard deviations are also determined.

II.B Are Real Fracture Networks Fractal

There is evidence that real fracture networks are fractal both in outcrops where Barton and others have found a fractal dimension of $D_f \approx 1.55$ (i.e. $1.50 < D_f < 1.6$ for different fracture systems) [13] as well as from underground data in the Fanay-Augères uranium mine [10] where they found a varying fractal dimension. It seems possible that the variation in their fractal dimension may result from use of too great a range of scales. As we saw for very large scales, all the rulers are covered so their finding a 'fractal dimension' of 1 at large scales is not surprising. Similarly, at very small scales one approaches a limit where the number of 'boxes' covered equals the number of fractures so the 'fractal dimension' approaches 0; this may be an artifact of the neglect of small aperture fractures (micro-cracks which may be significant in determining number at their 0.005 meter scale).

The length of the fractures has been found to be fractal, [14] and the shape of the fractures has also been determined to be fractal [15], [16], [17]. This suggests that all features of the fractures may be fractal: i) distributions of centers, ii) distributions of lengths, iii) distribution of widths, and iv) shapes. The evidence that the shapes are fractal suggests that porosities and permeabilities may also obey fractal statistics. If all geometrical aspects of the fracture distribution are fractal with the same fractal dimension, the fracture distribution is self-similar. This may seem to be a very unusual occurrence, but in fact many examples of development (or growth) which occur in random media (like the development of fractures in stressed rock formations) have a self-similar geometry. The first level of our geostatistical modeling will assume a self-similar fractal geometry for the fracture distribution. Higher levels of our geostatistical modeling will use actual measurements to determine the fractal distribution of (e.g.) the fracture widths.

III A 2-d Fracture Generation Algorithm

In this section we describe the implementation and design of an algorithm that was developed to generate a fracture network in 2-dimensions. As we have discussed in Section I, the primary assumption in our model is that the network geometry is fractal - i.e. has a self-similar or scale invariant geometry. Using this information we have developed a program to generate complete 2-d fracture outcrop networks using only the *lacunarity*, *fractal dimension*, *initial covering*, and *cutoff* parameters obtained from[†] MWX data.

The PASCAL programming language was chosen to emphasize both modularity and structure in the development and design stages of the algorithm. Since the PASCAL syntax is completely analogous to *pseudocode* used in general algorithm descriptions, the program can be easily modified by others or converted to another programming language at a later time.

The algorithm is most easily described by reference to the procedure flowchart in Fig. 8. The body of the program (Appendix A pp.26-38) consists primarily of variable and procedure declarations whose execution begin on p. 38. The procedures listed on p. 38 (and in Fig. 8) define the highest level of program hierarchy. All other procedures declared in the body of the program are called from within these procedures.

The most general description of the program is obtained by examining Fig. 8. In the broadest sense the program performs 2 tasks (separated by the dotted line):

- (1) Generates a horizontal fracture set.
- (2) Generates a vertical fracture set - consistent with the fracture set in (1)

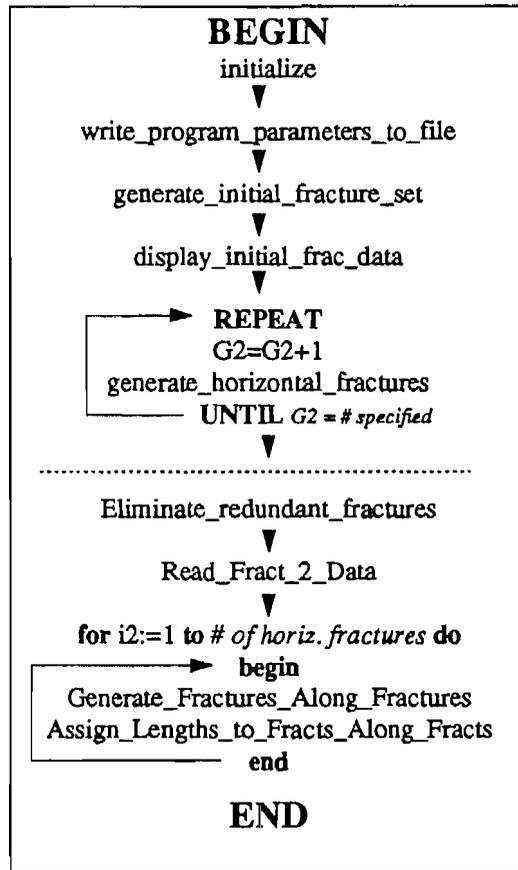


Fig. 8. Procedure Flowchart for *2d_frac.pas*

[†] Using the box-counting programs developed by C. Mick.

To generate the horizontal fracture set the program first generates a 1-dimensional fracture set along a left-justified line[†] extending downward in the vertical direction (see Fig. 10). This is accomplished by the procedure *GENERATE_INITIAL_FRACTURE_SET* whose flowchart is given in Fig. 1. The first step in the procedure initializes the first row in the 2-dimensional ruler array $L[j,k]$, where $i=1,2$ and k can range from 1 to $2^{13} = 8192$ as declared using the TYPE and VAR clauses at the beginning of the program. The range of the for loop given by the variable Ri is the initial number of rulers chosen to cover the fracture set in a 1-1 ratio. If a fracture is covered by a ruler, then the value of the array corresponding to this specific ruler is given the value 1. Conversely, an empty ruler site is given the value 0.

Having initialized the $L[1,i]$ array the *GENERATE_INITIAL_FRACTURE* procedure then divides each ruler into two new rulers (by mapping each ruler variable in $L[1,i]$ to two new ruler variables in $L[2,i]$). To accomplish this the procedure begins the *repeat...until* loop shown in Fig. 9 and increments the counting variable G (initially = 0) to the value of 1. The *GENERATION_PARITY* procedure then determines if G is odd or even and assigns the variables e and f the values (1 and 2) or (2 and 1) respectively, depending on whether G is odd or even.

Next, the procedure randomly chooses one of the 2 new rulers in $L[2,k]$ for each of the rulers in $L[1,i]$ and assigns this ruler a value of 1 while giving the other ruler a value of 0. In this way, the covered fractures in the initial level are brought down to the next level of $2Ri$ rulers. The remaining rulers are assigned fractures according to the distribution:

$$N = l \Delta^{D_f - 1}, \quad (1)$$

where N is the number of fractures, l is the lacunarity, D_f is the fractal dimension, and $\Delta = \frac{(\text{total length of fracture set} = 1)}{\text{number of rulers}}$. The progress of the algorithm is then checked by displaying the

fracture locations graphically as the program is running. After a single pass through the loop the number of rulers, R , is doubled and the whole process begins again. At the next iteration the parity of G will

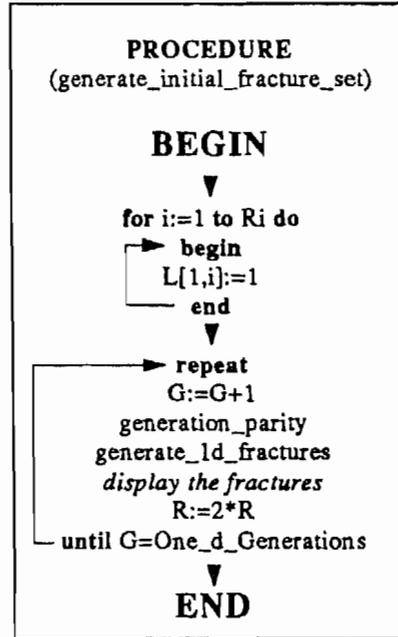


Fig. 9 Flowchart for the *generate_initial_fracture_set* procedure.

[†] we initially began the fracture generation process along a left justified line. This approach is being modified to produce outcrops originating from an arbitrary line within a distribution.

change as will the values of e and f according to the previous assignment. Using the mapping $L[e,i] \rightarrow L[f,j]$, the values of $L[2,j]$ are used as input for the next iteration and the values of $L[1,i]$ are overwritten with new fracture assignments in the next ruler doubling. The output at this stage of the program is shown in Fig. 10 and is analogous to Fig. 2a. Using $D_f = 1.5$, $l = 2.12$, with an initial covering of 4 we obtain 24 fractures with $4 \cdot 2^5 = 128$ rulers after 5 generations.

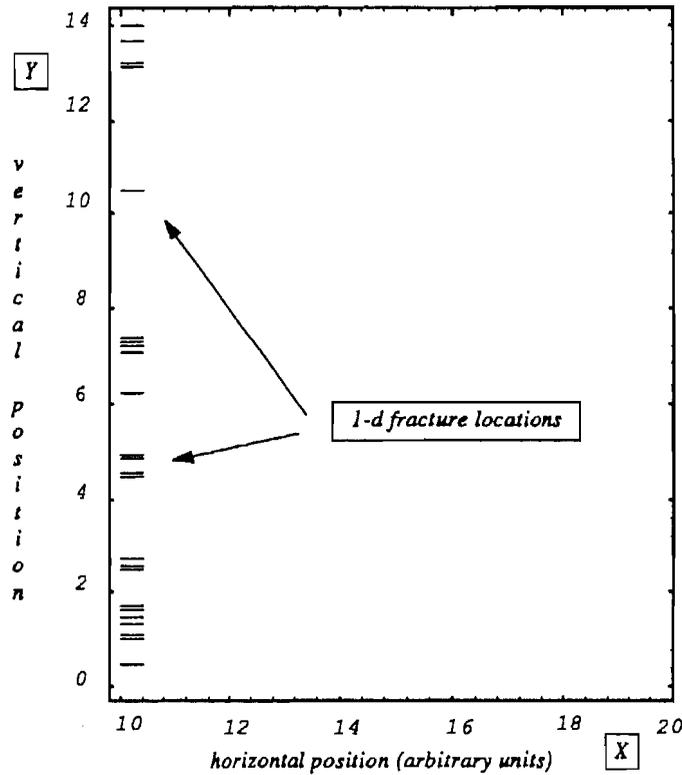


Fig. 10. 1-d fracture generation output. The data is represented graphically by a series of small line segments with extension in the x direction. In this example, the extension is arbitrary and is only used to illustrate the fracture locations.

Continuing with the generation of the horizontal fracture set the program enters the *repeat...until* loop shown directly above the dotted line in Fig. 1 and increments the counting variable $G2$ (initially = 0) to the value of 1. The loop executes the *GENERATE_HORIZONTAL_FRACTURES* procedure to produce a vertical fracture set for each value of the grid step in the x direction. Fig. 11 shows the procedure flowchart:

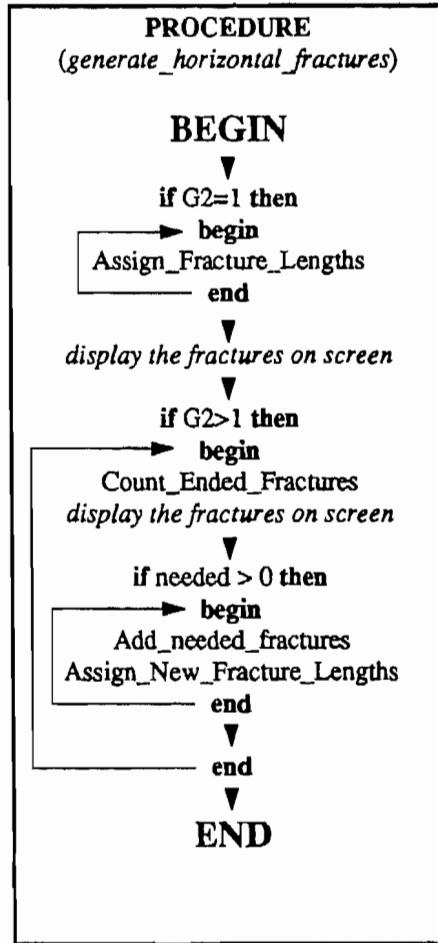


Fig. 11. Procedure flowchart for *generate_horizontal_fractures*.

In the first iteration ($G2=1$) the procedure assigns a length (extension in the x -direction) to each fracture site. To obtain the fracture length we assume a probability density function given by

$$p(L) = A L^{1-D_f}, \quad (2)$$

where L is the fracture length and A is a constant. The probability that a given fracture will have a length $\leq L'$ (greater than 2 arbitrary units) is then given by the distribution function:

$$P(2 < L \leq L') = \int_2^{L'} A L^{1-D_f} dL = \frac{A}{2-D_f} \left[L'^{(2-D_f)} - 2^{(2-D_f)} \right]. \quad (3)$$

Since the fracture must have a length between 2 and 100 (according to our assumed distribution) then the constant A is determined from

$$P(2 < L \leq 100) = 1 = \int_2^{100} A L^{1-D_f} dL, \quad (4)$$

so that

$$A = \frac{(2-D_f) 100^{(D_f-2)}}{\left[1 - \left(\frac{2}{100}\right)^{(2-D_f)}\right]} . \quad (5)$$

Substituting (5) into (3) then gives

$$P(L') = \left[\frac{1 - \left(\frac{2}{L'}\right)^{(2-D_f)}}{1 - \left(\frac{2}{100}\right)^{(2-D_f)}} \right] \left(\frac{L'}{100}\right)^{(D_f-2)} . \quad (6)$$

Generating a random number s_3 between $0 \rightarrow 1$ (labeled as s_3 in the `ASSIGN_FRACTURE_LENGTHS` procedure) and then setting this equal to (6), we can solve for the length L' to obtain:

$$L' = 10 \left[s_3 \left(1 - \left(\frac{2}{100}\right)^{(2-D_f)}\right) + \left(\frac{2}{100}\right)^{(2-D_f)} \right]^{\frac{1}{(2-D_f)}} . \quad (7)$$

Next, we generate a second random number s_4 between $0 \rightarrow 1$ and calculate the final fracture length from

$$L = s_4 L' . \quad (8)$$

In this way, the fracture sites are assigned lengths in the horizontal direction. Using the parameters $D_f = 1.5$, $l = 2.7$, and $R_f = 16$ we obtain 29 fractures with $16 \cdot 2^3 = 128$ rulers after 3 generations giving the output shown in Fig. 12.

Referring to Fig. 6 we notice that to the left of $L \approx 14$ (*linear regime*) the length assignments may be made using the procedure outlined above. To the right of $L \approx 14$ (*exponential cutoff*) we have a non-linear distribution and so we must use:

$$P_{II}(L) = 129 e^{-0.047L} . \quad (9)$$

To incorporate the data from region II into our fracture generation program (while avoiding having to solve a non-linear function for L') we are currently modifying the program by reading in the values from (2) and (9) into an array for each fracture length L' between 2 and 100. Generating a random number between 2 and 100 - we can then determine the corresponding fracture length from the array.

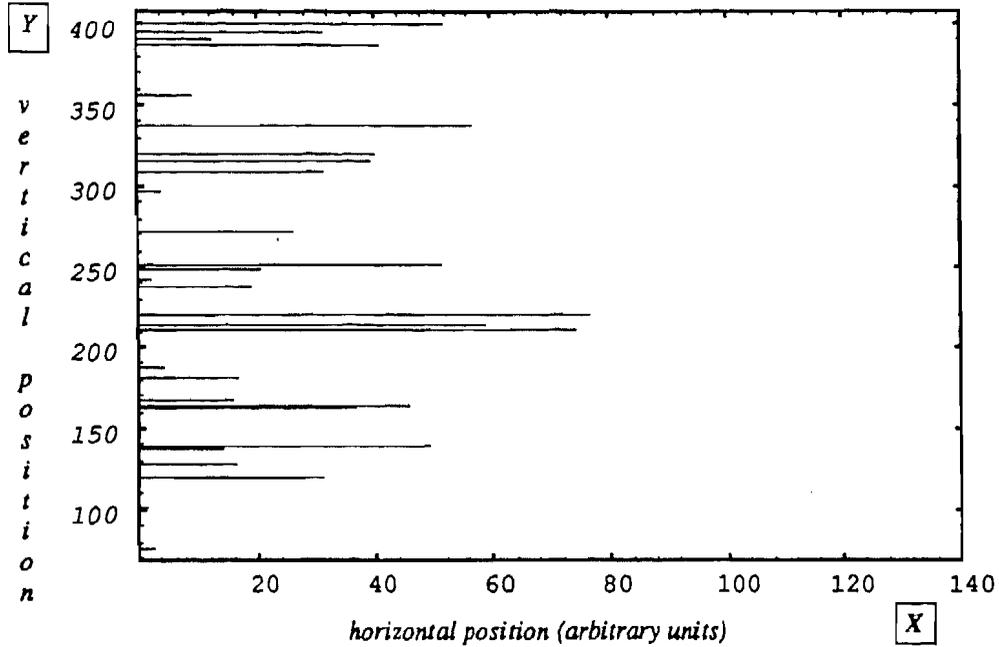


Fig. 12. Lengths are assigned to the initial fracture set.

At the next iteration, $G2$ is greater than 1 and the program will step forward by a specified amount in the x -direction ($= (G2 - 1) \times step$) to determine (using the previous length assignments) how many fractures extend past this point. If fractures have ended, new fracture assignments must be made to maintain the distribution in (1). The number of fractures that have not crossed the grid point are counted by the procedure *COUNT_ENDED_FRACTURES* and stored in the variable *needed*. If fractures have ended, the procedure *ADD_NEEDED_FRACTURES* is executed as shown in Fig. 11. To guarantee that the new fracture assignments produce a fractal distribution, we must reverse the ruler doubling process and re-assign fractures that have crossed the specified grid point to half as many rulers used in the final step of the initial 1-d fracture generation process. The unoccupied fracture sites are then assigned new fractures following the same procedure described for the initial fracture generation.

After adding new fractures (beginning from $x = (G2 - 1) \times step$) the *ASSIGN NEW FRACTURE LENGTHS* procedure uses (7) and then (8) to determine their length. The x_1 and x_2 coordinates (endpoints) for each of the fractures are stored in the arrays $Lfx1[i]$ and $Lfx2[j]$ and the whole process continues until the distribution is generated for the specified number of horizontal site locations. The endpoints of the fractures along with their vertical position are written to the file *FRACT_1.DAT* for each value of the gridstep x given above by the *DISPLAY_FRAC_EXTENSIONS* procedure. The resulting output is given in Fig. 13 and can be compared with the MWX horizontal fracture data shown in Fig. 14. The parameters used were $D_f = 1.5$, $l = 2.7$, and $R_i = 16$ which were determined from the MWX outcrop using the box counting procedures described in Section II.

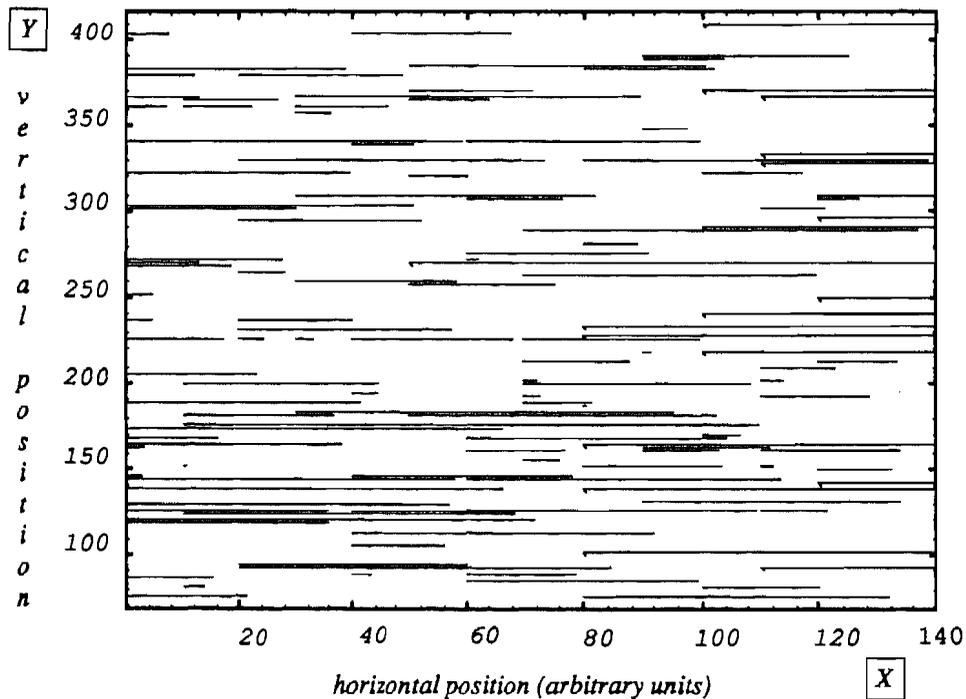


Fig. 13 Horizontal fracture outcrop

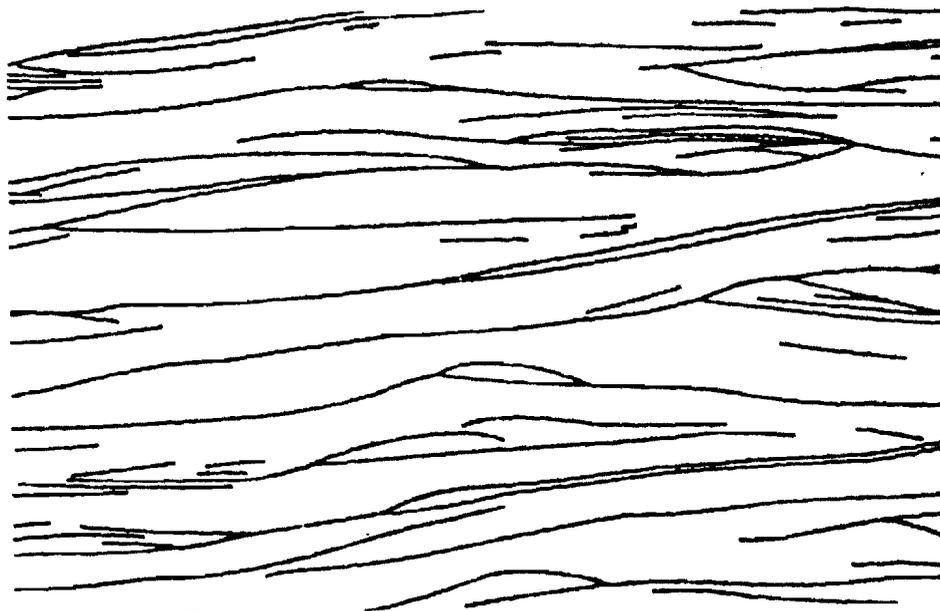


Fig. 14 MWX Horizontal Fracture Outcrop Data

To generate the vertical fracture set we first generate a fracture distribution along each of the horizontal fractures by applying our 1-d generation algorithm to each fracture in the datafile FRACT_1.DAT. Since the horizontal fracture positions were previously stored at each value of the gridstep, a fracture crossing n gridpoints is stored n times by the *DISPLAY_FRAC_EXTENSIONS* procedure. Therefore, before we can assign vertical fractures along each of the horizontal fractures we must first eliminate all duplicate fractures from the data set. This is accomplished by the *ELIMINATE_REDUNDANT_FRACTURES* procedure listed below the dotted line in Fig. 8. The result of this operation is then stored in a new file: FRACT_2.DAT. After obtaining a unique set of horizontal fractures we reinitialize our variables by reading in the FRACT_2.DAT values with the *READ_FRAC_2_DATA* procedure as shown in Fig. 8.

Starting in the upper left hand corner of Fig. 13 and proceeding downward vertically, the program produces a fractal distribution (using a parameter set determined from the vertical fracture data) along the first fracture in the data set. In our model we assume that vertical fractures can only begin or end along a horizontal fracture. In this case, we need only find the next horizontal fracture below each vertical fracture site to determine the fracture endpoint and therefore its' length. To begin the process the program enters the for loop below the *READ_FRAC_2_DATA* procedure in Fig. 8. If the $i2$ -th horizontal fracture has a length greater than a certain number of units, the program executes the *GENERATE_FRACTURES_ALONG_FRACTURES* procedure to generate a fracture set along the $i2$ -th fracture. The flowchart for this procedure is completely analogous to the flowchart given in Fig. 2 except that in this case we use a slightly different 1-d fracture generation procedure

(*GENERATE_ID_FRACTURES2*) to incorporate the vertical fracture parameters and new fractal distribution function.

After producing a fracture distribution along the *i2-th* horizontal fracture, the program executes the *ASSIGN LENGTHS TO FRACTURES ALONG FRACTURES* procedure whose flowchart is given in Fig. 15. The outer for loop in the procedure scans through all rulers of the fracture distribution just produced by the *GENERATE FRACTURES ALONG FRACTURES* procedure. If a fracture site is occupied then the vertical position of the horizontal fracture is stored in the variable *y1*. The location of the fracture along the *i2-th* horizontal fractures' length is then stored as *xf1*. Now that we have the *x* and *y* values of the vertical fractures' starting point - we scan the fracture set (using the *SORT FRACTURES* and *CHOOSE_THE_NEXT_FRACTURE_BELOW* procedures) to find the vertical position of the next horizontal fracture beneath our given fracture. This position is then stored as *y2*. If the value of *y2* corresponds to a fracture within the boundaries of the network (and not at an adjacent grid site starting at the top of the screen) then the vertical fracture is displayed and its' position stored in the file: *FRAC_3.DAT*. The program terminates when the horizontal fractures have been scanned and vertical fractures are generated along their lengths. Using the identical parameters that were used for Fig. 13 along with the parameters $D_{f,vertical} = 1.2$,

$l_{vertical} = 1$, and $R_{i,vertical} = 4$ we obtain the output shown in Fig. 16 which can be compared with the MWX data in Fig. 17.

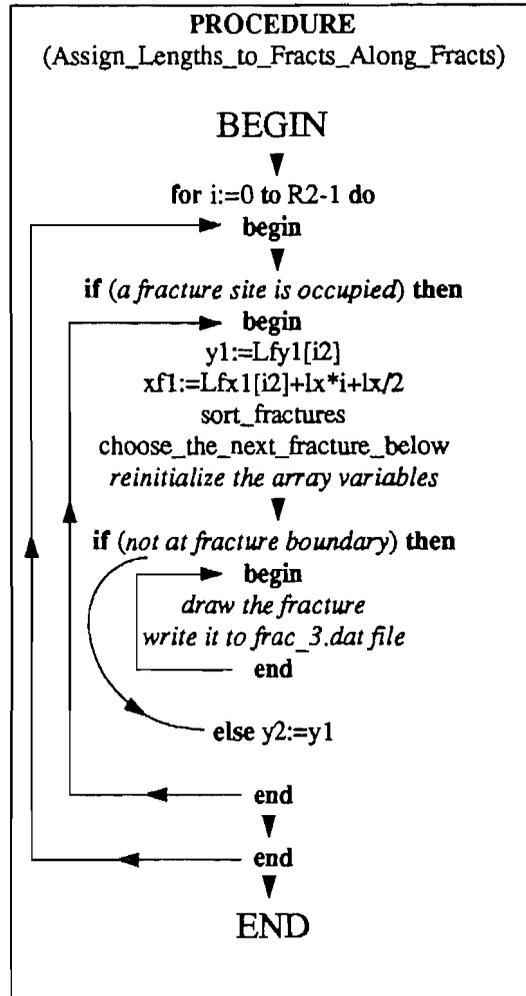


Fig. 15 Procedure flowchart for *ASSIGN_LENGTHS_TO_FRACTURES_ALONG_FRACTURES*.

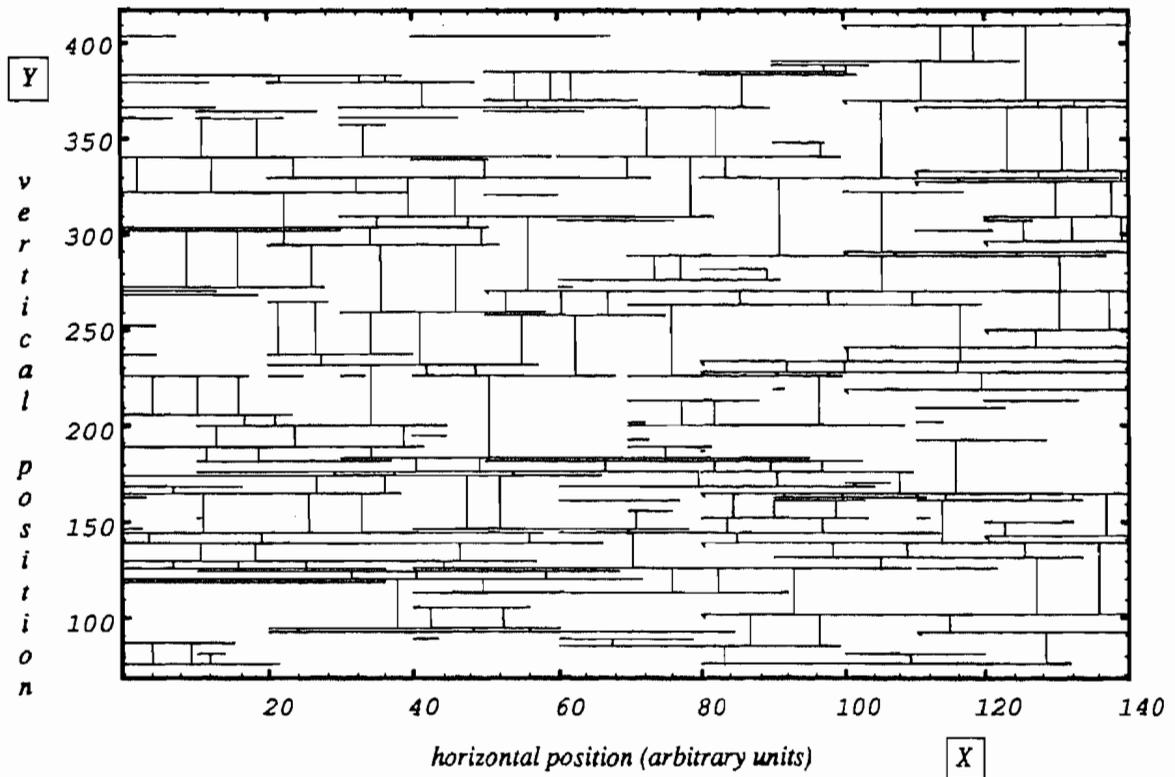


Fig. 16. 2-d Fracture Outcrop Data generated by *2d_frac2.pas*.

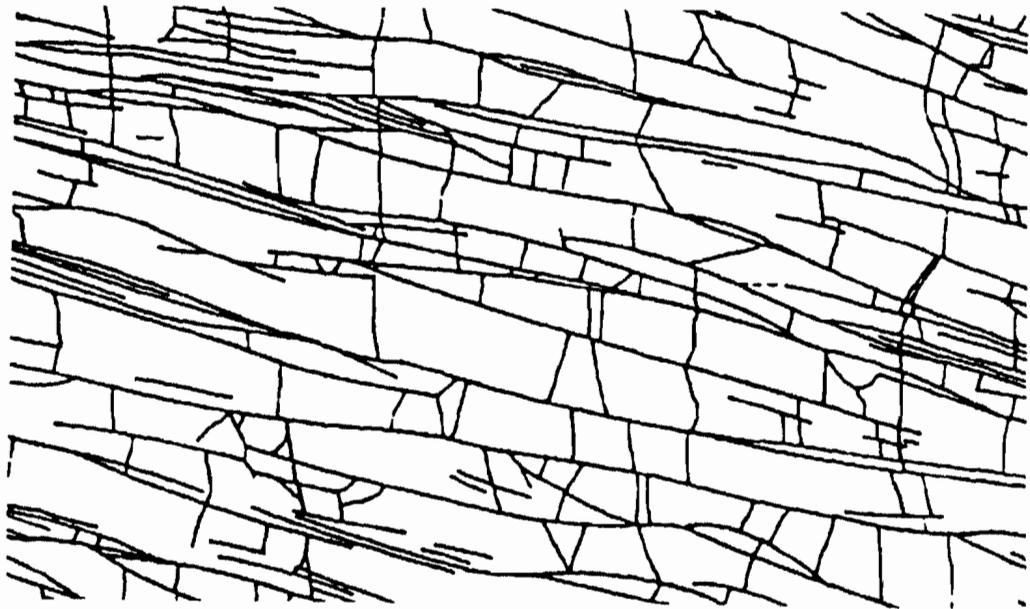


Fig. 17. MWX 2-d Fracture Outcrop Data

IV. Discussion

To model the fracture outcrop networks occurring in naturally fractured tight-gas reservoirs we have taken an approach that incorporates:

- I. Data Analysis: we characterize the MWX fracture data using four parameters (for both horizontal and vertical fractures):
 - (a) Lacunarity
 - (b) Fractal Dimension
 - (c) Initial Covering Scale (R_i).
 - (d) Cutoff - determined from the distribution of fracture lengths.
- II. Fracture Generation: we generate self-similar fracture networks using data from I.) with an algorithm that incorporates fractal geostatistics.

From our work we have found that there are several advantages in an approach that uses fractal statistics:

- 1) The networks produced by our model appear to be in agreement with actual fracture networks but do not require extensive a-priori knowledge of the network. Using data from isolated borehole sites we can generate entire networks with an algorithm that assumes a self-similar or scale invariant geometry.
- 2) We are able to generate horizontal and vertical fractures separately (although not independently) using distinct parameter sets in each case. The fractures can then be analyzed and combined later to produce complete self-consistent networks.
- 3) Since the data is generated using a statistical approach, the algorithms require relatively little computer time to produce complete networks (\approx 2-3 minutes on a 486DX2-50).
- 4) Evidence suggests that real fracture networks obey fractal statistics (see section II.B and part I of this report).

The characterization and analysis of the network data produced by our algorithms is not yet complete. By varying other parameters such as gridsize, fracture length, and the horizontal/vertical orientation of fractures, we believe that it will be possible to generate fracture distribution patterns that are 'optimally similar' in the fractal/statistical sense - to real fracture networks occurring in nature.

References

1. Skopec, R. A., JPT. December 1993, 1168, (1993).
2. Davidge, R. W. "Mechanical Behavior of Ceramics." 1979 Cambridge University Press. New York.
3. McKoy, M., private communication. (1994).
4. Xie, H. "Fractals in Rock Mechanics." Geomechanics Research Series. Kwasniewski ed. 1993 A. A. Balkema. Rotterdam.
5. McKoy, M., Development of Stochastic Fracture Porosity Models and Application to the Recovery Efficiency Test (RET #1) Well in Wayne County, West Virginia (1993).
6. Long, J. C. S. and D. M. Billaux, Wat. Resources Res. **23**, 1201, (1987).
7. Billaux, D., J. P. Chilès, K. Hestir and J. C. S. Long, Int. J. Rock Mech., Min. Sci. & Geomech. Abstr. **26**, 281, (1989).
8. Hewett, T. A. "SPE 15386 Fractal Geostatistics for reservoir hetero's." 1986 Soc. of Pet. Eng. Richardson, TX.
9. Matthews, J. L., A. S. Emanuel and K. A. Edwards, JPT. 1139, (1989).
10. Chilès, J. P., Math. Geol. **20**, 631, (1988).
11. Feder, J. "Fractals." 1988 Plenum Press. New York.
12. Mandelbrot, B. B. "The Fractal Geometry of Nature." 1982 W. H. Freeman Publishers. New York.
13. LaPointe, P. R., Int. J. Rock Mech., Min. Sci. & Geomech. Abstr. . **25**, 421, (1988).
14. Heffer, K. J. and T. G. Bevan, fracture length scaling (1990).
15. Roach, D. E., A. D. Fowler and W. K. Fyson, Geology. **21**, 759, (1993).
16. Roach, D. E. and A. D. Fowler, Computers & Geosci. **19**, 849, (1993).
17. Maloy, K. J., A. Hansen, E. L. Hinrichsen and S. Roux, Phys. Rev. Lett. **213**, (1992).

Appendix A: Program Listing: *2d_frac.pas*

(Pascal Source,† 670 lines)

```
PROGRAM td_frac2; {--- Program declaration ---}

USES crt,graph; {--- Libraries that will be used ---}

TYPE
  ruler=array[1..2,1..8192] of integer;
  one_d_array=array[1..1600] of real;

VAR      {--- Variables are explained as encountered ---}
  i,
  i1,
  i2,
  c,
  i5,
  j,
  k,
  m,
  s,
  G,
  G2,
  G3,
  e,
  f,
  R,
  Ri,
  R2,
  Ri2,
  count_f;
  c1,
  c2,
  n1,
  needed,
  Magnif,
  step,
  u,
  y_step,
  One_d_Generations,
  One_d_Generations2,
  Two_d_Generations:      integer;
  L                        :ruler;
  temp,
```

† compiled and developed using *Borland - Turbo Pascal* Version 7.0 under DOS 6.1 on a home-built 486DX2-50 with 16MB of memory. The fracture network figures were produced with *Mathematica* ver. 2.2 running under *Windows* 3.1.

```

xf1,
xf2,
cx,
cy,
sx,
sy,
lx,
ly,
Lf,
s3,
s4,
y,
s1,
s2,
y1,
y2,
y3,
Lacunarity,
Lacunarity2,
Fractal_Dim,
Fractal_Dim2,
f_lenth           :real;
Lfx1,
Lfx2,
Lfy1             :one_d_array;
x1,
x2,
Fractures,
Resolution,
Gnumber,
Covered          :string[6];
datafile,
datafile2,
datafile3        :text;
write1           :string[1];
write            :boolean;

```

```

FUNCTION N(d:integer):integer; {--- This function gives the distribution of the horizontal fractures---}
BEGIN
  N:=Round( Lacunarity*Exp( (Fractal_Dim-1)*ln(d) ) );
END;

```

```

FUNCTION N2(d:integer):integer; {--- This function gives the distribution of the vertical fractures---}
BEGIN
  N2:=Round( Lacunarity2*Exp( (Fractal_Dim2-1)*ln(d) ) );
END;

```

```

PROCEDURE initialize;           {--- Initialize graphics screen and scaling parameters ---}
BEGIN
  clrscr;randomize;G:=0;G2:=0;
  Magnif:=3;count_f:=0;
  c1:=detect;c2:=0;initgraph(c1,c2,'c:\tp\BGI');
  setbkcolor(3);

```

```

SetFillStyle(EmptyFill,0);
n1:=8; write1:=0;write:=false;
cx:=100;cy:=100*(getmaxy/getmaxx);
sx:=50;{getmaxx/n1;}sy:=0.8*getmaxy/n1;
setttextjustify(centertext,centertext);
outtextxy(trunc(getmaxx/2),10,'2-D Fractures');
delay(1000);
END;

```

```

PROCEDURE display_initial_frac_data; {--- Display various parameters on-screen ---}
BEGIN
  k:=0;
  for i:=1 to R do
    begin
      if L[f,i]=1 then k:=k+1;
    end;
  setcolor(4);
  outtextxy(Trunc(0.1*getmaxx),Trunc(0.95*getmaxy),'Rulers =');
  outtextxy(Trunc(0.14*getmaxx+22),Trunc(0.98*getmaxy),'Starting Fractures =');
  outtextxy(Trunc(0.14*getmaxx+13),Trunc(0.92*getmaxy),'Initial Covering =');
  str(R,Resolution);
  str(k,Fractures);
  setcolor(15);
  outtextxy(Trunc(0.175*getmaxx),Trunc(0.95*getmaxy),Resolution);
  outtextxy(Trunc(0.315*getmaxx),Trunc(0.98*getmaxy+0),Fractures);
  str(Ri,Resolution);
  outtextxy(Trunc(0.295*getmaxx),Trunc(0.92*getmaxy),Resolution);
END;

```

```

PROCEDURE generation_parity; {-- Determine if G is odd or even --}
BEGIN
  if (G/2-trunc(g/2)) >0 then
    begin
      e:=1;
      f:=2;
    end;
  if (G/2-trunc(g/2)) =0 then
    begin
      e:=2;
      f:=1;
    end;
END;

```

```

PROCEDURE display_1d_fractures; {--- Display Fract. along y at a previous Gen. ----}
BEGIN
  ly:=0.7*getmaxy/R;
  for i:=0 to R-1 do
    begin
      if (i/2-trunc(i/2)) >0 then setcolor(9);
      if (i/2-trunc(i/2)) =0 then setcolor(12);
      line(trunc(sx),trunc(cy+ly*i),
          trunc(sx),trunc(cy+ly*(i+0.95)));
      if (L[e,i+1]=1) then

```

```

    begin
      putpixel(trunc(sx),trunc(cy+ly*i+ly/2),15);
    end;
  end;
END;

PROCEDURE display_1d_fractures2; {--- Display Fract. along y at the final Gen. ---}
BEGIN
  ly:=0.7*getmaxy/(2*R);
  for i:=0 to 2*R-1 do
    begin
      if (i/2-trunc(i/2)) >0 then setcolor(9);
      if (i/2-trunc(i/2)) =0 then setcolor(12);
      line(trunc(sx),trunc(cy+ly*i),
          trunc(sx),trunc(cy+ly*(i+0.95)));
      if (L[f,i+1]=1) then
        begin
          putpixel(trunc(sx),trunc(cy+ly*i+ly/2),15);
        end;
    end;
  end;
END;

PROCEDURE display_1d_fractures3; {-- This backs up to R/2 and shows L[e,i] --}
BEGIN
  ly:=0.7*getmaxy/(R/2);
  for i:=0 to trunc(R/2)-1 do
    begin
      if (i/2-trunc(i/2)) >0 then setcolor(13);
      if (i/2-trunc(i/2)) =0 then setcolor(4);
      line(trunc(sx-15),trunc(cy+ly*i),
          trunc(sx-15),trunc(cy+ly*(i+0.95)));
      if (L[e,i+1]=1) then
        begin
          putpixel(trunc(sx-15),trunc(cy+ly*i+ly/2),15);
        end;
    end;
  end;
END;

PROCEDURE display_1d_fractures4; {--- Show fractures by pixel at each Gen. ---}
BEGIN
  ly:=0.7*getmaxy/(R);
  for j:=0 to R-1 do
    begin
      if (j/2-trunc(j/2)) >0 then setcolor(9);
      if (j/2-trunc(j/2)) =0 then setcolor(12);
      line(trunc(sx+(G2-1)*Step*Magnif),trunc(cy+ly*j),
          trunc(sx+(G2-1)*Step*Magnif),trunc(cy+ly*(j+0.95)));
      if (L[f,j+1]=1) then
        begin
          putpixel(trunc(sx+(G2-1)*Step*Magnif),trunc(cy+ly*j+ly/2),15);
        end;
    end;
  end;
END;

```

```

PROCEDURE display_fractures_y1; {--- Show fractures at a previous stage ---}
BEGIN
  lx:=Abs(Lfx2[i2]-Lfx1[i2])/(R2);
  for i1:=0 to R2-1 do
    begin
      if (i1/2-trunc(i1/2)) >0 then setcolor(1);
      if (i1/2-trunc(i1/2)) =0 then setcolor(1);
      line(trunc(sx+(Lfx1[i2]+lx*i1)*Magnif+3) ,trunc(Lfy1[i2]),
           trunc(sx+(Lfx1[i2]+lx*(i1+0.95))*Magnif+3),trunc(Lfy1[i2]));
      if L[e,i1+1]=1 then
        begin
          putpixel(trunc(sx+(Lfx1[i2]+lx*i1+lx/2)*Magnif+3), trunc(Lfy1[i2]),15);
        end;
      end;
    END;

PROCEDURE display_fractures_y2;
BEGIN
  lx:=Abs(Lfx2[i2]-Lfx1[i2])/(2*R2);
  for i1:=0 to 2*R2-1 do
    begin
      if (i1/2-trunc(i1/2)) >0 then setcolor(1);
      if (i1/2-trunc(i1/2)) =0 then setcolor(1);
      line(trunc(sx+(Lfx1[i2]+lx*i1)*Magnif+3) ,trunc(Lfy1[i2]),
           trunc(sx+(Lfx1[i2]+lx*(i1+0.95))*Magnif+3),trunc(Lfy1[i2]));
      if L[f,i1+1]=1 then
        begin
          putpixel(trunc(sx+(Lfx1[i2]+lx*i1+lx/2)*Magnif+3), trunc(Lfy1[i2]),15);
        end;
      end;
    END;

PROCEDURE display_frac_extensions;
BEGIN
  ly:=0.7*getmaxy/R;          {--This gives the ruler lengths in the y-direc. ---}
  if g2=1 then              {---Create datafile fract_1.dat ---}
    begin
      assign(datafile,'c:\tp\files\FRACT_1.DAT');
      rewrite(datafile);
    end;
  Bar(trunc(sx+490),trunc(cy), trunc(getmaxx),trunc(cy+340)); {--- Erase old data from the screen ---}
  for i:=0 to R-1 do
    begin
      if (L[f,i+1]=1) then
        begin
          count_f:=count_f+1;
          str((g2-1),Gnumber);
          Lfy1[i+1]:=cy+ly*i+ly/2;
          writeln(datafile,Lfx1[i+1]:3:2,',Lfx2[i+1]:3:2,',Lfy1[i+1]:3:2); {---write to fract_1.dat ---}
          setcolor(1);
          line(trunc(sx+Lfx1[i+1]*Magnif+3),trunc(Lfy1[i+1]), {--- display the fractures on screen ---}
              trunc(sx+Lfx2[i+1]*Magnif+3),trunc(Lfy1[i+1]));
        end;
    end;

```

```

setcolor(8);
if (G2-1)>0 then      {--- Draw the gridlines on the screen ---}
  line(trunc(sx+(G2-1)*Step*Magnif),trunc(cy),
       trunc(sx+(G2-1)*Step*Magnif),trunc(cy+ly*(R-1)+ly/2));
if Step>=10 then    {--- Print the grid values on the screen ---}
  begin
    setcolor(8);
    str((g2-1)*step,Gnumber);
    outtextxy(trunc(sx+(G2-1)*Step*Magnif+0),trunc(cy-20),Gnumber);
    str((g2-1),Gnumber);
    outtextxy(trunc(sx+(G2-1)*Step*Magnif+0),trunc(cy+345),Gnumber);
  end;
end;
end;
END;

PROCEDURE generate_1d_fractures;      {--- 1-d Algorithm - Generates Vertical Slices along x--}
BEGIN
  for i:=1 to R do  {-- Divide Measuring Scale and Bring down Fractures --}
  begin
    if L[e,i]=1 then {-- If fractures are present, add fractures below -}
    begin
      for j:=1 to 1 do
      begin
        s:=random(2)+1;
        if s=1 then
          begin
            L[f,2*i-1]:=1;
            L[f,2*i] :=0;
          end;
        if s=2 then
          begin
            L[f,2*i-1]:=0;
            L[f,2*i] :=1;
          end;
        end;
      end
    else      {-- If no fractures are present, add spaces ----}
    begin
      L[f,2*i-1]:=0;
      L[f,2*i] :=0;
    end;
  end;
  for i:=1 to (N(2*R)-N(R)) do {----- Add fractures according to distribution ----}
  begin
    repeat
      s:=random(R)+1;
      until (L[e,s]=1) and not ((L[f,2*s-1]=1) and (L[f,2*s]=1));
      if L[f,2*s-1]=1 then L[f,2*s]:=1 else L[f,2*s-1]:=1;
    end;
  END;
PROCEDURE generate_1d_fractures2;
BEGIN

```

```

for i:=1 to R2 do  {-- Divide Measuring Scale and Bring down Fractures --}
begin
  if L[e,i]=1 then {-- If fractures are present, add fractures below -}
    begin
      for j:=1 to 1 do
        begin
          s:=random(2)+1;
          if s=1 then
            begin
              L[f,2*i-1]:=1;
              L[f,2*i] :=0;
            end;
          if s=2 then
            begin
              L[f,2*i-1]:=0;
              L[f,2*i] :=1;
            end;
          end;
        end;
      end
    else      {-- If no fractures are present, add spaces ----}
      begin
        L[f,2*i-1]:=0;
        L[f,2*i] :=0;
      end;
    end;

for i:=1 to (N2(2*R2)-N2(R2)) do  {----- Add fractures ----}
begin
  repeat
    s:=random(R2)+1;
    until (L[e,s]=1) and not ((L[f,2*s-1]=1) and (L[f,2*s]=1));
    if L[f,2*s-1]=1 then L[f,2*s]:=1 else L[f,2*s-1]:=1;
  end;
END;

PROCEDURE Assign_Fracture_Lengths;  {--- Assign lengths to horizontal fractures---}
BEGIN
  if G2=1 then  {---- Assign initial lengths to fractures ---}
    begin
      for i:=1 to R do
        begin
          if L[f,i]=1 then
            begin
              s3:=(random(100)+1)/100;
              Lf:=Exp((2/3)*Ln( (Exp(1.5*Ln(100))-Exp(1.5*Ln(2))))*s3 ));
              s4:=(random(100)+1)/100;
              Lfx1[i]:=0;
              Lfx2[i]:=s4*Lf;
            end;
          end;
        end;
      end;
    end;
  END;

```

```

PROCEDURE Count_Ended_Fractures; {--- count horizontal fractures that have ended---}
BEGIN
  for i:=1 to R do {--- If a fracture has ended, count it ---}
    begin
      if (L[f,i]=1) and (Lfx2[i]<(G2-1)*Step) then
        begin
          needed:=needed+1;
          L[f,i]:=0;
          Lfx1[i]:=(G2-1)*Step;
          Lfx2[i]:=0;
        end;
      end;
    END;

PROCEDURE Display_Grid_info; {--- display grid values on screen---}
BEGIN
  str(needed,Gnumber);outtextxy(trunc(sx+(G2-2)*Step*Magnif+7),trunc(cy-40),Gnumber);
  setcolor(15);outtextxy(trunc(22),trunc(cy-40),'Need:');
  outtextxy(trunc(22),trunc(cy-20),'Grid:');
END;

PROCEDURE Add_needed_fractures; {---add horizontal fractures that have ended ---}
BEGIN
  for i:=1 to needed do
    begin
      repeat
        Bar(trunc(sx-45),trunc(cy-10),
          trunc(sx-20),trunc(cy+335));
        s:=random(round(R/2))+1;
        str(s,resolution);setcolor(14);
        outtextxy(trunc(sx-30),Trunc(cy-17+ly*2*s),resolution);
      until (L[e,s]=1) and not ((L[f,2*s-1]=1) and (L[f,2*s]=1));
      if L[f,2*s-1]=1 then
        begin
          L[f,2*s]:=1;
          Lfx1[2*s]:=(G2-1)*Step;
        end
      else
        begin
          L[f,2*s-1]:=1;
          Lfx1[2*s-1]:=(G2-1)*Step;
        end;
      display_1d_fractures4; {--- show the new fracture positions as they are added ---}
    end;
  END;

```

```
PROCEDURE Assign_New_Fracture_Lengths; {--- Assign lengths to new fractures ---}
```

```
BEGIN
```

```
  for i:=1 to R do
```

```
    begin
```

```
      if (L[f,i]=1) and (Lfx1[i]=(G2-1)*Step) then
```

```
        begin
```

```
          s3:=(random(100)+1)/100;
```

```
          Lf:=Exp((2/3)*Ln( (Exp(1.5*Ln(100))-Exp(1.5*Ln(2))) *s3 ));
```

```
          s4:=(random(100)+1)/100;
```

```
          Lfx2[i]:=(G2-1)*Step + s4*Lf;
```

```
        end;
```

```
    end;
```

```
END;
```

```
PROCEDURE generate_horizontal_fractures; {--- procedure for generating horiz. fracture set ---}
```

```
BEGIN
```

```
  needed:=0; {--- Initialize this variable for the next generation ---}
```

```
  if G2=1 then {---Assign fracture lengths for the initial generation ---}
```

```
    Assign_Fracture_Lengths;
```

```
  display_frac_extensions;
```

```
  display_1d_fractures4;
```

```
  if G2>1 then
```

```
    begin
```

```
      Count_Ended_Fractures;
```

```
      Display_Grid_info;
```

```
      display_1d_fractures3;
```

```
      display_1d_fractures4;
```

```
      if needed>0 then
```

```
        begin
```

```
          Add_needed_fractures;
```

```
          Assign_New_Fracture_Lengths;
```

```
        end;
```

```
      end;
```

```
END;
```

```
PROCEDURE generate_initial_fracture_set;;
```

```
BEGIN
```

```
  for i:=1 to Ri do {--- Set the first Level Fractures --}
```

```
    begin
```

```
      L[1,i]:=1;
```

```
    end;
```

```
  G:=0;
```

```
  repeat {--- start 1-d fracture generation -----}
```

```
    G:=G+1; {---- G Counts the Generations -----}
```

```
    generation_parity; {--- is G odd or even ? -----}
```

```
    generate_1d_fractures; {---- Algorithm -----}
```

```
    display_1d_fractures;
```

```
    display_1d_fractures2;
```

```
    R:=2*R; {-- double the scale resolution --}
```

```
  until G=One_d_Generations; {--- end of 1-d loop --}
```

```
END;
```

```

PROCEDURE Eliminate_redundant_fractures;
BEGIN
  reset(datafile);
  m:=0;
  while not Eof(datafile) do {--- Read in values and count how many from fract_1.dat---}
  begin
    m:=m+1;
    readln(datafile,Lfx1[m],Lfx2[m],Lfy1[m]);
  end;
  close(datafile);
  for i:=1 to m do
  begin
    if Lfy1[i]<>(-1) then
    begin
      for j:=1 to m do
      begin
        if (i<>j) and ((Lfx1[i]=Lfx1[j]) and (Lfx2[i]=Lfx2[j]) and (Lfy1[i]=Lfy1[j])) then
          Lfy1[j]:=-1;
        end;
      end;
    end;
  end;
  Assign(datafile,c:\np\files\FRACT_2.DAT'); {---Create file of unique fractures---}
  Rewrite(datafile);
  for i:=1 to m do
  begin
    if Lfy1[i]<>(-1) then
    begin
      writeln(datafile,Lfx1[i]:3:2,',',Lfy1[i]:3:2,',',
              Lfx2[i]:3:2,',',Lfy1[i]:3:2);
    end;
  end;
  close(datafile);
END;

PROCEDURE Generate_Fractures_Along_Fractures; {-- assign fractures along horiz. fractures--}
BEGIN
  R2:=Ri2;
  for i:=1 to R do    {--- Set the first Level Fractures ---}
  L[1,i]:=1;
  G:=0;
  repeat
  G:=G+1;
  generation_parity;
  generate_1d_fractures2;
  display_fractures_y1;
  display_fractures_y2;
  R2:=2*R2;
  until G=One_d_Generations2;
END;

```

```

PROCEDURE Switch(Var a,b:Real); {--- This is used in the sorting procedure ---}
  Var
    c:real;
  BEGIN
    c:=a;
    a:=b;
    b:=c;
  END;

PROCEDURE Sort_fractures; {--- sort the fractures to assign vertical fractures to next one below ---}
  Var
    i3,i4:integer;
  BEGIN
    for i3:=2 to u do
      begin
        for i4:=u DownTo i3 do
          begin
            if (Lfy1[i4-1]>Lfy1[i4]) then
              begin
                Switch( Lfy1[i4] , Lfy1[i4-1] );
                Switch( Lfx1[i4] , Lfx1[i4-1] );
                Switch( Lfx2[i4] , Lfx2[i4-1] );
              end;
            end;
          end;
        end;
      END;

PROCEDURE Choose_The_Next_Fracture_Below; {--- Go through sorted list --}
  VAR
    i5:integer;
  BEGIN
    i5:=0;
    repeat
      i5:=i5+1;
    until (Lfy1[i5]>y1) and (xf1>=Lfx1[i5]) and (xf1<=Lfx2[i5]);
    y2:=Lfy1[i5];
  END;

PROCEDURE Read_Fract_2_Data;
  VAR
    i5:integer;
  BEGIN
    i5:=0;
    reset(datafile);      {---- datafile is Frac_2.dat ----}
    while not Eof(datafile) do
      begin
        i5:=i5+1;
        readln(datafile,Lfx1[i5],Lfy1[i5],Lfx2[i5],Lfy1[i5]);
      end;
    u:=i5;
    close(datafile);
  END;

```

```
PROCEDURE Assign_Lengths_to_Fracts_Along_Fracts; {-- generate and display vertical fractures --}
```

```
BEGIN
```

```
  for i:=0 to R2-1 do {---- Assign lengths to fractures ----}
```

```
    begin
```

```
      if L[f,i+1]=1 then
```

```
        begin
```

```
          y1:=Lfy1[i2];
```

```
          lx:=Abs(Lfx2[i2]-Lfx1[i2])/(R2);
```

```
          xf1:=Lfx1[i2]+lx*i+lx/2;
```

```
          xf2:=Lfx2[i2]+lx*i+lx/2;
```

```
          Sort_Fractures;
```

```
          Choose_The_Next_Fracture_Below;
```

```
          setcolor(1);
```

```
          Read_Fract_2_Data;
```

```
          if (y2>=0) and (y2<=500) then
```

```
            begin
```

```
              Line(Trunc(sx+(xf1)*Magnif+3),Trunc(y1),
```

```
                Trunc(sx+(xf1)*Magnif+3),Trunc(y2));
```

```
              Append(datafile3);
```

```
              writeln(datafile3,xf1:3:2,' ',y1:3:2,' ',
```

```
                xf1:3:2,' ',y2:3:2);
```

```
              close(datafile3);
```

```
            end
```

```
          else y2:=y1;
```

```
        end;
```

```
    end;
```

```
END;
```

```
PROCEDURE write_program_parameters_to_file; {--- make a datafile of the parameters used ---}
```

```
BEGIN
```

```
  Assign(datafile3,'c:\vp\files\fr_text.dat');
```

```
  Rewrite(datafile3);
```

```
  writeln(datafile3,Lacunarity:3:2,' ',Lacunarity2:3:2,' ',
```

```
    Fractal_Dim:3:2,' ',Fractal_Dim2:3:2,' ',
```

```
    Step,' ',One_d_Generations,' ',One_d_Generations2);
```

```
  close(datafile3);
```

```
END;
```

```

BEGIN {----- The Program starts here and executes the procedures-----}

initialize;

write_program_parameters_to_file;

generate_initial_fracture_set;

display_initial_frac_data;      {---- displays initial program info -----}

G2:=0;          {--- initialize G2 ---}
REPEAT          {----- Start 2-d fracture generation -----}
  G2:=G2+1;
  generate_horizontal_fractures;
UNTIL G2=Two_d_Generations;

close(datafile); {--- Datafile is Frac_1.dat, containing the endpoints of the horizontal fractures ---}

Eliminate_redundant_fractures; {--Remove duplicate fractures from frac_1.dat and save as frac_2.dat --}

Read_Fract_2_Data; {---Reinitialize variables with unique fracture values ---}

Assign(datafile3,'c:\tp\files\frac_3.dat'); {--- Create the datafile: frac_3.dat to store vertical fractures ---}
Rewrite(datafile3);
close(datafile3);

for i2:=1 to u do {--- u is the total number of horizontal fractures ---}
  begin
    if Abs(Lfx2[i2]-Lfx1[i2])>=20 then { - start vertical fractures only along horizon. fractures with length ≥ 20 units -}
      begin
        Generate_Fractures_Along_Fractures;
        Assign_Lengths_to_Fracts_Along_Fracts;
      end;
    end;

setcolor(14);
outtextxy(trunc(0.8*getmaxx),20,'Done. '); {--- Print a message that the program is finished ---}
readln; {--- wait until a key is pressed ---}
closegraph; {--- exit from graphics mode ---}

END. {----- Program ends here -----}

```

Appendix B: Program Listing: *Box-Counting Procedure*

```
$ type boxtest3.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 80

/* Function prototypes */
void box_count( float a[], int, float );
void read_me( int, FILE*, float );
float least_squares( double a[], double b[], int );

main()
{
char read_file[20];
int fracture_number=0;
float initial_ruler_length;
FILE *ifp;

printf("\n\n\nEnter the file to be read: ");
scanf("%s", read_file);
printf("\n\n Enter the initial ruler length: ");
scanf( "%f", &initial_ruler_length);

ifp=fopen(read_file,"r");
if( ifp == NULL )
{
fprintf(stderr,"%s","\n\n\t\tSorry, File cannot be located.....Try again!");
exit( 1 );
}
/*call function to read in data*/

read_me( fracture_number,ifp, initial_ruler_length);
close(ifp);

/*end main*/
}

void read_me( int fracture_number, FILE *ifp, float initial_ruler_length )
{
int i=1, k;
float x;
float fract[ SIZE ];

fract[ 0 ] = 0;
```

} Section
I

Session Name: Vax A

```
while( ! feof( ifp ))
{
    fscanf( ifp, "%f", &x );
    fract[ i ] = x;
    i++;
    fracture_number++;
}
```

I

```
/* end read_me*/
```

```
box_count( fract, fracture_number, initial_ruler_length);
}
```

```
void box_count( float fract[], int fn, float initial_ruler_length )
{
    /*fn=fracture_number*/
    int delta=1,i=0,k;
    int j=0;
    int a_fracture_was_covered=FALSE;
    int rulers_covered=0;
    float ruler_increment;
    float ruler_length;
    float initial_length;
    int rulers[ SIZE ];
    int del[ SIZE ];
    double fractal_regime[ SIZE ];
    int size_of_rulers_array;
    double delta_fractal[ SIZE ];
    initial_length=fract[--fn]; /* printf("\n\n%f",initial_length);return;*/
```

Section II

```
printf("    Delta    Rulers_covered\n");
```

```
while( rulers_covered < fn )
```

```
{
    rulers_covered = 0;
    i = 0;
    ruler_increment = initial_ruler_length/delta;
    ruler_length=ruler_increment;
```

```
while( ruler_length <= initial_ruler_length )
```

```
{
    a_fracture_was_covered=FALSE;
    while( fract[i] <= ruler_length && i < fn )
    {
        a_fracture_was_covered=TRUE;
        i++;
    }
```

```
if( a_fracture_was_covered ) rulers_covered++;
    ruler_length+=ruler_increment;
```

```
printf("    %d            %d\n",delta,rulers_covered);
```

```
/**** eliminate first linear regime *****/
```

```
if( rulers_covered != delta && rulers_covered != fn)
```

section III

Session Name: Vax A

```
    {
      rulers[ j ] = rulers_covered;          /**temporary arrays**/
      del[ j ] = delta;                      /**to isolate fractal region**/
      j++;
    }
  delta*=2;
}

size_of_rulers_array = j;
/** for(k=0;k<j;k++)printf("\n\rulers: %d del: %d",rulers[k],del[k]);
printf("\n size of= %d",size_of_rulers_array); **/ /*check*/

    /**Print Headers***/

printf("\n\n\t\t\t\t\t-----FRACTAL REGION-----\n");
printf("\n\t\t\t\t\tDelta\t\t\t\t\tRulers Covered");

    /**eliminate linear saturation***/
k=0;
while( k < size_of_rulers_array )
  {
    if( rulers[ k ] != rulers[ k + 1 ] )
      {
        fractal_regime[ k ] = rulers[ k ];
        delta_fractal[ k ] = del[ k ];
        printf("\n\t\t\t\t\t%f\t\t\t\t\t%f",delta_fractal[k],fractal_regime[k]);
        k++;
      }
    else break;
  }
size_of_rulers_array = k;
for( k = 0; k < size_of_rulers_array; k++ )
  {
    fractal_regime[ k ] = log10( fractal_regime[ k ] );
    delta_fractal[ k ] = log10( delta_fractal[ k ] );
    /** printf("\n%f\t\t\t\t\t%f",delta_fractal[k],fractal_regime[k]); **/
  }

least_squares(fractal_regime, delta_fractal, size_of_rulers_array);

}

$
```

IV

Session Name: Vax A

```
$ type least_squares.c
#include <math.h>

float least_squares( double y[], double x[], int n )
{
    float m;    /** m = slope **/
    float b;    /** b = y-intercept **/
    float m_sigma; /** standard deviation in slope **/
    float b_sigma; /** standard deviation in y-intercept **/
    float upstairs = 0;
    float downstairs;
    float xy_sum = 0.0;
    float x_sum = 0.0;
    float y_sum = 0.0;
    float sum_square = 0.0;
    float product_sum;
    float square_sum;
    int i = 0;

    while( i < n )
    {
        xy_sum += x[i] * y[i];
        x_sum += x[i];
        y_sum += y[i];
        sum_square += x[i] * x[i];

        /** printf("\n%f %f n= %d",x[i],y[i],n); **/ /*check*/
        /* printf("\n%f = xysum",xy_sum); **/
        i++;
    }
    product_sum = x_sum * y_sum;
    square_sum = x_sum * x_sum;

    m = (n * xy_sum - product_sum) / ( n * sum_square - square_sum );
    printf("\n\n\tThe slope of the line = %f.",m);
    /** printf("\nsumsquare=%f\nprodsum=%f\n",sum_square,product_sum); **/
    /** printf("\nx_sum=%f ysum=%f",x_sum,y_sum); **/

    /** y-intercept calculation.....y-intercept = b **/

    b = (sum_square * y_sum - x_sum * xy_sum)/( n*sum_square - square_sum);
    b = pow( 10.0, b);
    printf("\n\tb = %f", b);

    /** least square error analysis **/

    /* Standard deviation of slope */

    for( i = 0; i < n; i++)
    {
        upstairs += pow(( y[ i ] - ( m * x[ i ] +log10( b))), 2.0);
    }
    upstairs *= n;

    if( n ==2 )
```

section IV

Session Name: Vax A

```
{
printf("\n\n\tThere must be more than two data values for a complete");
printf("\n\terror analysis.");
exit( 1 );
}

downstairs = sqrt((double)(n-2) * ( n*sum_square - square_sum));
m_sigma = upstairs / downstairs;

/** Standard deviation of y-intercept **/

b_sigma = sum_square * upstairs / downstairs;

printf("\n\t\tError Analysis::\n");
printf("\t m = %f +- %f.",m, m_sigma);
printf("\n\t b = %f +- %f.",b, b_sigma);

/* End function---least_squares */
}
$
```