MORPH-II,
a software package for the analysis of scanning-electron-micrograph images for the assessment of the fractal dimension of exposed stone surfaces

by Victor G. Mossotti[1] and A. Raouf Eldeeb[2]

Open-File Report 00-013

2000

U.S. DEPARTMENT OF THE INTERIOR
U.S. GEOLOGICAL SURVEY

[1]345 Middlefield Road, MS 901, Menlo Park, CA 94025
[2]508 Cheyenne Drive, Sunnyvale, CA 94087

This report is available online at http://geopubs.wr.usgs.gov/open-file/of00-013/.

CONTENTS

Page

Introduction ---- 1

Overview of software functions ---- 1

Overview of image analysis programs ---- 1

Image generation ---- 1

Sample preparation ---- 2

SEM signal considerations ---- 2

Use of EDGE.EXE ---- 3

Hardware requirements ---- 3

Format requirements ---- 3

Image display requirements ---- 3

Image calibration ---- 4

Image analysis ---- 4

EDGE.C algorithm design & operation ---- 10

Computation of fractal dimension ---- 10

Finite-state approach to image analysis ---- 11

Isolation and definition of pore perimeters ---- 11

Acknowledgements ---- 13

References ---- 14

Appendix 1: Source code ---- 15

Appendix 2: Users manual ---- 27

Introduction

Turcotte, 1997, and Barton and La Pointe, 1995, have identified many potential uses for the fractal dimension in physicochemical models of surface properties. The image-analysis program described in this report is an extension of the program set MORPH-I (Mossotti and others, 1998), which provided the fractal analysis of electron-microscope images of pore profiles (Mossotti and Eldeeb, 1992). MORPH-II, an integration of the modified kernel of the program MORPH-I with image calibration and editing facilities, was designed to measure the fractal dimension of the exposed surfaces of stone specimens as imaged in cross section in an electron microscope.

Overview of software functions

This report documents the computer code for the software package MORPH-II. This package was created for the analysis of scanning-electron microscope (SEM) and electron-microprobe images for the following materials properties: fractal dimension of exterior surface area, fracture density of interior structure, length of structural displacements (*e.g.*, recession), and standard statistics on edge profiles.

Overview of image analysis programs

Two versions of MORPH were developed. MORPH-I was designed to measure the fractal dimension, *D* (Barton and La Pointe, 1995), of the folded rock surface at the interface between the rock mass and pore-space volumes. MORPH-II was written to measure the fractal dimension of the exposed surface of a rock specimen cut in cross section. Although, in principle, MORPH-I and MORPH-II can be used to define an imbedded or an exposed contact between unconforming physical structures over any range of scale for which imagery is available, the programs were specifically designed for the analysis of secondary-electron micrograph. MORPH-II requires a format *N* rows x 512 columns x 8 binary files (images with 8 bits/pixel on a 256-shade gray-scale palette). Many of the functions (image display, calibration, analysis, and so on) provided by the collection of programs in MORPH-I, as well as additional features, are supported by the single program EDGE.EXE in MORPH-II; this report addresses the details of MORPH-II.

Image generation

The method for surface profile analysis used here is based on direct observation of exposed-surface cross sections. The vertical and lateral variegations along the exposed surface of a rock specimen provide a sample of the irregularities in the interface surface between the solid rock mass and open-space volumes. Much as the holes in a slice of Swiss cheese appear in cross section, the edge profiles of the depressions and holes in a polished cross section of stone show the pore geometry of the specimen. In the work described here, the program package is used extensively for the analysis of the microstructure of Salem limestone and Berkshire Lee marble. In dealing with these materials, we assume that mineral grains in the mass of the stone are cut cleanly at the level of the polished surface and that cutting and polishing does not pluck grains from the surface. We assume that local relief below the polished stone surface represents exposed pore space. We also assume, at least in this initial report, that the stone pore structure is isotropic.

The image-analysis software was designed to analyze secondary-electron images. Such images are collected by raster scanning over selected areas of interest; the magnification (15 to 100,000X) was controlled by varying the size of the scanned area. The electron-beam current, nominally 350 μAmp, provided a depth of focus of about 5 μm. At the highest magnification used in the study, a 36-μm^2 area of the surface is represented by a 512 x 512-pixel electron-micrograph image. Thus, the lateral extent represented by the width of a single pixel is approximately 0.01 μm; because of edge effects and excitation volume effects along the rim of a pore cross-section, the actual lateral resolution is in the range 0.03-0.05 μm. Thus, in order to make full use of the spatial resolution capabilities of the instrument, it is necessary for the sample to be polished to a flatness of 0.05 μm.

Sample preparation

The chemical makeup of the Salem limestone and Berkshire Lee marble selected for study has been described in detail by Mossotti and others (1986, 2000). The laboratory samples of the limestone were obtained in the form of briquettes measuring 2.4 x 3.5 x 2.5 cm. They were obtained from the collection of freshly quarried stone used in connection with field studies of pollutant effects on carbonate stone (Sherwood and Reddy, 1988); the Lee marble samples were cut from cores taken from Philadelphia City Hall in connection with a masonry-cleaning study on that structure (Mossotti and others, 2000).

The limestone and marble field cores (2.54-cm diameter) used for exposed-surface analysis in this study were sliced perpendicular to the weathered surface with a diamond wafering-saw lubricated with water. The specimens were potted with YALE CIBA 6010 resin with YALE CIBA 840 hardener, with the flat-cut surface pressed to the bottom of the mold. The flat sides of the potted specimens were then smoothed on a 600-grit lap wheel until the exterior layer of epoxy was removed and until evidence of grain disaggregation and grain plucking disappeared. The exposed stone surfaces were subsequently polished to a flatness of 0.05 μm with alumina polishing compound. The specimens were cleaned ultrasonically to remove the rock fragments and polishing snow from the pore spaces and were coated with a 200-nm layer of gold.

SEM signal considerations

The secondary-electron signal originates at the point on the surface where the primary electrons enter the specimen. The conventional technique for image production uses the secondary-electron signal to monotonically modulate the brightness of the display such that the brighter parts of the image correspond to those parts of the specimen from which the signal is greatest. The intensity of the secondary-electron signal depends, in a complex way, on the shape, chemical composition, and orientation of the volume of material excited by the primary-electron beam.

The contrast in the displayed image is caused primarily by variations in the surface topography. For example, when the surface contains a cavity or is delimited by an edge, the surface discontinuity appears to be darker than the surrounding surface. This is because secondary electrons are either altogether missing or are extracted with less efficiency from the recessed area than from the more elevated area; fewer secondary electrons reach the collector from such surface depressions. When SEM images are viewed by the human eye, the human brain provides topological interpretation based on the rendering of primary-beam shadowing in the displayed image. In this work we simplify the interpretational problem

by associating the three phases of interest—the depression space, the solid phase, and the surface—with the corresponding states of a finite state machine; we will denote these machine states as PORE, MASS and EDGE, respectively.

In effect, by polishing the specimen to a flatness of 0.05 μm, we add information to the image. Polishing the specimen favors the uniform excitation and collection of secondary electrons along a given contour by removing surrounding topological features that can cast a shadow inside surface depressions and over edges. In particular, we know that the highly-polished surface shows an abrupt drop-off where the edge is exposed; any local point on the specimen that is not in a recessed area is at a fixed topological elevation. Based on this information, we can partition the set of image pixels into two subsets: the PORE pixels corresponding to recessed regions, and the MASS pixels corresponding to the polished solid phase.

Use of EDGE.EXE

Hardware requirements

The programs described in this report are executable on computers providing an MSDOS-compatible environment. Suitable command lines are provided by native MSDOS, OS/2 virtual DOS, Windows 3.1, Windows 95 DOS, Windows 98 DOS, and Windows NT DOS emulation. The main image-analysis program requires Super VGA (SVGA) mode-101 display. Accordingly, an SVGA adaptor board with suitable drivers is required to run the program. A 60-MHz, 486-DX (math coprocessor) PC with at least 4 MB of RAM is recommended as the minimal system.

Format requirements

The program EDGE.EXE, the main component of MORPH-II, requires an image file with a similar format specification required for MORPH-I (Mossotti and others, 1998). The 262,144-byte linear binary used by MORPH-I represents an array of 512 rows with 512 pixels per row. The essential difference between the data file formats in MORPH-I and MORPH-II is that the number of rows in the MORPH-II data files is dynamically established by the user. This feature allows the user to crop an image as needed, thereby reducing the file size. When the requisite file format is provided by the image source, the images can be displayed, calibrated, and analyzed without any further concern for format.

Image display requirements

EDGE.EXE in the only program providing image display in MORPH-II. The computer display can be configured in a variety of standardized display formats. These formats control the configuration of the display memory, the resolution of the display, the number of bits per pixel, the default character font, and the starting address of display memory.

EDGE.EXE runs in SVGA mode-101 (800 x 600 x 8). Unlike the EGA and VGA boards, which include their own on-board basic input/output system (BIOS), the SVGA adaptor boards generally

require resident software drivers to control the SVGA display modes. These drivers, which are generally compliant with the VESA standard for BIOS extension, must be installed before the main program in MORPH-II, EDGE.EXE, is started. Unless provision has been made for the operation of SVGA mode-101, EDGE.EXE will return an error message when it is started from the command line. When the program has finished its work, it returns the display mode to the standard VGA 640 x 480 x 4 before closing.

Image calibration

The central task of the program EDGE.EXE, and certainly the most challenging job, is the identification of unconforming contacts representing physical structures in the electron-micrograph image. In the design of the central algorithm for MORPH-II (EDGE.EXE) we considered the electron-micrograph image in terms of a population of cellular automata that was made to evolve through three sequential generations. In this algorithm, the fate of a given cell is determined by the relationship between the signal intensity of the cell and that of its first- and second-nearest neighbors. As the state structure of the image evolves, contention issues related to the state of each pixel are resolved by reference to unambiguous definitions of PORE and MASS pixels. Thus, two numbers between 0 and 255 delimit the range of gray values defining the states PORE and MASS. For example, pixels with gray values less than or equal to the lower threshold (T_L) are defined as PORE. Similarly, pixels with gray values equal to or greater than the upper threshold (T_U) are defined as MASS. Contention issues arise along the edges (EDGE pixels) delimiting pore space and the solid phase. These are resolved by associating a range of gray values falling between the threshold parameters with the state EDGE. All EDGE pixels eventually evolve into either PORE or MASS pixels, depending on the spatial environment of each pixel in the image.

A fundamental technical problem in automated SEM image analysis concerns the interpretation of contrast in a photomicrograph. The purpose of image calibration is to identify the gray-level ranges for the identification and partitioning of PORE, EDGE, and MASS pixels in the image. Thus, image calibration is the first step in image analysis.

Image calibration is conducted by visual inspection of the SEM image with the pixels tentatively identified as EDGE highlighted in red in the display. When an image is opened with the program EDGE.EXE, the calibration function is menu-selectable. When the calibration function is invoked, the image appears with the highlighted EDGE pixels; on start-up of the program, the EDGE state is based on default values for the upper and lower thresholds (T_U, T_L). During the calibration procedure, the user can actively adjust the upper and lower thresholds to alter the population of pixels identified as EDGE. The user may elect to increase the sensitivity of the analysis to particular variegations along a contour line on the specimen by altering the threshold parameters to include pixels falling within a given range of gray values. The adjustments of T_U and T_L are based on the user's experience with electron micrographs and on the user's knowledge of the physical structure of the material under examination.

Image analysis

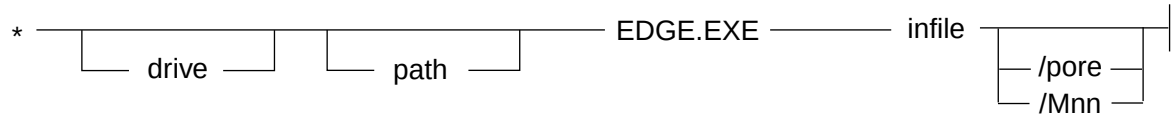
Program name: EDGE.EXE.

Program use: The program EDGE.EXE provides fractal analysis of the exposed surface of the specimen

based on the cross-sectional profile of the exposed surface. The kernel (the image-processing engine) of the program is described below and in Mossotti and others, 1998 (MORPH-I). EDGE.EXE provides for integrated image calibration, image editing, fracture density measurement, and the measurement of linear distances between any two planes through the image. In its default configuration, the program generates a fractal analysis of the exposed surface of the specimen; a tacit assumption in this default configuration is that the specimen has been prepared for SEM analysis of the cross-sectional profile of the exposed surface. The program also can be switched on startup to compute the statistics of enclosed pore surface as described below in the command line synopsis.

Program environment: SEM2BIN.EXE is invoked from an MSDOS-compatible command line.

Synopsis of the command line:



In the command line synopsis above, "infile" is the name of the image file to be processed; the file name should be preceded by the full path (drive and directory containing the file) if necessary. It is assumed that the file format conforms to the binary image file format described above. This software does not read any other format.

As a default, EDGE.EXE initially isolates and displays the trace of the longest contact or edge in the image using default calibration parameters T_U and T_L; these parameters can be changed by selection of the calibration tool from the main menu or by hitting F9. The longest edge generally corresponds to the exposed surface of the specimen. The optional switch "/pore" cues the program to compute and report on the fractal dimension of every exposed pore in the cross-sectional image; implicit in the use of this switch is the assumption that the specimen has been prepared for enclosed-pore-surface analysis (image of cross-sectional cut through the rock).

The switch "/Mnn" advises the program on the image magnification. Because the definition of magnification is somewhat ambiguous, the symbol "nn" in the switch "/Mnn" represents the actual linear raster-scan resolution in units of μm/pixel. At the highest resolution of the TN/SEM at the USGS in Menlo Park, a lateral resolution of 0.05 μm roughly corresponds to a linear magnification factor of 2000X if the 512 x 512-pixel image is rendered over a 10 x 10-cm display. Most of the work described in this report was conducted at a linear magnification factor of either 10X or 100X. A default value of 17.4 μm/pixel, a raster-scan resolution corresponding to 10X magnification, is loaded into the analysis program EDGE.EXE (see check_args() in EDGE.C); use of the "/Mnn" switch is optional. The linear resolution is used by EDGE.EXE to calculate the distance between two planes through the images as controlled by the loss function (F9 from the main menu).

Program I/O: The input file is a generic, tab-delimited binary file representing N rows of 512 pixels; N is equal to or less than 512 (Mossotti and others, 1998). EDGE.EXE provides a menu-selectable tool, "clip," that allows the user to visually crop the image as a rectangular frame.

Use of program: EDGE.EXE provides a continuously-updated graphical display of the image as various menu functions are exercised. In the main menu, the user is given the following options:

Command	F-key	Letter	Function
Clip	F2	C	Clips a section of the image
Edit	F3	E	Edits individual pixels of the image
Erase	F4	R	Resets a region of the image
Calibr8	F5	B	Opens image-calibration tool
Trace	F6	T	Traces edge and finds fractal dimension
Save	F7	S	Saves the current image
Density	F8	D	Computes fracture density
Loss	F9	L	Opens caliper tool
Exit	F10	X	Exits the program
Print	—	P	Prints image to a Paintjet printer

Except for the print function, each of these options can be invoked, either by the appropriate function key, by typing the designated letter, or by highlighting the command with the cursor (left or right) and pressing ENTER. The following is a more detailed description of each of these features:

CLIP Function key: F2 Fast key: C Return to main menu: ESC
Summary: clips and opens section of the image for analysis

When the exposed surface of the specimen is selected for analysis, most of the image information above and below the exposed surface is not used. In these situations, the file size can be reduced and the computation efficiency increased without compromise by discarding the portion of the image that represents the bulk properties of the specimen and the void region above the surface. The clip tool allows the user to crop the useful part of the image before quantitative analysis is performed on the file. By selecting "clip" from the EDGE.EXE menu, the user can manipulate a set of graphical delimiters to isolate the region of the original image to be retained in the reduced file. After the section of the image to be retained has been cropped with the delimiters on the displayed image, execution of the clip function (by pressing the ENTER key) will invoke a prompt for the name of the clipped file to be saved; a binary file will be generated with the stem name of the input file and with the default extension CUT. After entering the stem name for the newly-clipped file, the user presses the ESC key to return to the main menu. Before displaying the main menu, EDGE.EXE closes the original input file and loads the clipped file on the fly without the user having to reschedule EDGE.EXE.

EDIT Function key: F3 Fast key: E Return to main menu: ESC
Summary: edits individual pixels of the image

This function is invoked to edit specific pixels of the image or to adjust the overall brightness of the image. The edit screen has the following sub-options: F3, increases the brightness of every pixel in the image by a magnitude of 15 (on each activation of F3) such that the brightest pixel will not exceed maximum brightness (255); F4, decreases the overall brightness of the image by a magnitude of 15 (on each activation of F4) such that the dimmest pixel will not fall below 0.

Additional sub-commands are linked to a specific targeted pixel in the image that is highlighted by a cross-hair pattern. These sub-commands include:

Movement of cross-hair target for pixel selection:

NW (Home key) **NE** (PgUp key)

NORTH (Up arrow)

WEST (Left arrow) **EAST** (Right arrow)

SOUTH (Down arrow)

SW (End key) **SE** (PgDn key)

In addition, a display on the image shows the numerical values of the targeted pixel and of the first and second neighboring pixels in every direction. Pixel values less than that of the targeted pixel are displayed in green, those agreeing with it in red, and those exceeding it are in yellow.

\+ increments the value of the targeted pixel
– decrements the value of the targeted pixel

U sets the upper limit of the boundary region to the value of the targeted pixel
L sets the lower limit of the boundary region to the value of the targeted pixel

ERASE Function key: F4 Fast key: R Return to main menu: ESC
Summary: rewrites selected region of the image

This function is used to edit selected groups of pixels in the image by rewriting the intensity values of all of the pixels in the group. The erase function is useful to remove image noise, errors, or structures that do not contribute to the structural information of interest. The size and location of the region to be modified can be selected with the following sub-options:

Movement of region to be modified:

NORTH
(Up arrow)

WEST (Left arrow) **EAST** (Right arrow)

SOUTH
(Down arrow)

Size of the region to be modified:
F5 or **PgUp** expands the edit region
F6 or **PgDn** shrinks the edit region

Activation of EDIT mode:
F3 toggles edit mode ON/OFF, where
\+ increments the intensity value of all pixels in the selected region, and
– decrements the intensity value of all pixels in the selected region.

CALIBR8 Function key: F5 Fast key: B Return to main menu: ESC
Summary: provides for image calibration

This function is used to establish calibration parameters for the image. Image calibration is conducted by visual inspection of the SEM image with the pixels tentatively identified as EDGE highlighted in red in the display. The calibration mode allows the user to change the upper and lower threshold parameters (T_U and T_L, respectively) in accordance with the calibration procedure described above in the section entitled "Image calibration." During modification, T_U and T_L are displayed below the image; the threshold parameters T_U and T_L can be modified by activation of the following sub-options:

Modification of upper threshold (T_U):
F3 increments upper threshold (T_U)
F4 decrements upper threshold (T_U)

Modification of lower threshold (T_L):
F5 increments upper threshold (T_L)
F6 decrements upper threshold (T_L)

TRACE Function key: F6 Fast key: T Return to main menu: ESC
Summary: traces edge and reports fractal dimension

This function graphically highlights the profile over which the fractal and conventional statistics are computed. The linear regression line through the profile is also displayed along with a summary of the statistical computations over the profile. The report displayed on the screen includes:

Dimension: fractal dimension of profile
Crossings: number of profile crossings of regression line
Deviation: standard deviation around regression line
PixlCount: number of pixels in profile

SAVE Function key: F7 Fast key: S Return to main menu: ESC
Summary: saves current image with all changes

This operation will overwrite any previous files with the name of the current file without prompting the user. A message indicates that the image has been saved and gives the size of the image.

DENSITY Function key: F8 Fast key: D Return to main menu: ESC
Summary: computes fracture density

The density function displays a continuously-updated report on the percentage of PORE, MASS, and BOUNDARY pixels in an isolated area of the image and reports the percent fracturing in the selected area as the sum of the PORE and BOUNDARY percentages. The size and location of the isolated area on the image can be selected with the following sub-options:

Movement of selected area:

NORTH
(Up arrow)

WEST **EAST**
(Left arrow) (Right arrow)

SOUTH
(Down arrow)

Size of selected area:
F5 or **PgUp** expands the selected area
F6 or **PgDn** shrinks the selected area

LOSS Function key: F9 Fast key: L Return to main menu: ESC
Summary: opens caliper tool

This operation allows the user to measure the distance between two planes through the image by manipulation of a set of calipers superimposed on the image. The separation of the calipers is continuously updated and numerically reported on the display. The report, in units of μm, is based on linear pixel resolution of the image. The user can override the default resolution of 17.4 μm/pixel, a raster-scan resolution corresponding to 10X, by invoking the switch "/Mnn" on the command line when EDGE.EXE is scheduled. In the switch, "nn" represents the actual linear raster-scan resolution in units of μm/pixel. Use of the "/Mnn" switch is discussed in more detail above under "Image analysis: synopsis

of the command line." The location, slope, and separation of the calipers can be controlled with the following sub-options:

Location of caliper set:
F3 moves set UP
F4 moves set DOWN

Slope of caliper set:
F5 INCREASES slope
F6 DECREASES slope

Separation of caliper set:
F7 OPENS calipers
F8 CLOSES calipers

EXIT Function key: F10 Fast key: X Return to main menu: ESC
Summary: exits the program

Before activating EXIT, the user should first save the current image, if desired, with the menu selection SAVE.

PRINT Fast key: P Return to main menu: ESC
Summary: generates ESC sequence and image file for Paintjet printer

Activation of the print function generates an ESC sequence and image file for Paintjet printer with the PRN extension and prints an image on a reduced gray-scale palette to a Paintjet printer. Caution: this function will produce unpredictable results if the printer attached to printer port LPT1 is not a Paintjet printer. On completion of the print function, control is returned to the main menu.

EDGE.C algorithm design & operation

Computation of fractal dimension

The method used for the determination of the fractal dimension in this work is based on a Richardson structured walk at a fixed contour level along the trace of the stone pore surface or along the profile of the exposed surface of the specimen (Turcotte, 1997, p. 55-66.) The Richardson effect, in its elegant simplicity, asserts that an irregular contour line can be approximated with a broken line made up of N intervals of length r, where

$$N = \mu_D r^{-D}.$$

If the value of the exponent D is constant within a range of values for the yardstick r, the line is said to be fractal and the exponent D is called the fractal dimension. In keeping with common practice, our algorithm uses the slope of the log (N) vs. log (r) curve to estimate the fractal dimension for a given

contour. Because we assume the stone pore structure is isotropic, we can obtain the fractal dimension of the interfacial surface simply by incrementing the fractal dimension determined for the contour line by unity.

The most difficult aspect of automated pore-structure analysis, and perhaps the most exacting part of automating measurements on any fractal structure, is training the computer to recognize a fractal boundary. If the interfacial surface between the mass and pore space is truly fractal, the boundary trace will be self-similar at all scales. If the user directly could view the boundary itself instead of an image of the boundary, the user would be unable to unambiguously define the interfacial trace in the lateral plane at any scale. The fundamental uncertainty in the location of an interfacial boundary is given by the length of the smallest yardstick available for the fractal measurement. In SEM photomicrographs, this yardstick length is the lateral resolution represented by one image pixel. The first operation in the implementation of the Richardson algorithm is the isolation and definition of each exposed pore perimeter in the cross-sectional view of the specimen.

Finite-state approach to image analysis

At the start of image processing, the only way the program can regard the image is as an amorphous collection of pixels. Before the program has recognized or processed any information, the granularity of the information is low and the information is disseminated throughout the entire image. Although the first stage of image processing responds to finite-state information from within the immediate neighborhood of a given pixel, the early operations follow a locus that extends to the full horizon of the image. Thus, 262,144 bytes (four 64K segments) of data need to be accessible, in memory, by the program during the primitive stages of image processing. As the program proceeds with the image processing, its task is to localize the information into well-defined structures. As the program isolates such structures in the image, the granularity of the information increases and the horizon over which the program needs to operate becomes increasingly circumscribed.

In the implementation of the kernel, a pixel is a pointer to a location in memory that contains the intensity of the pixel; the intensity of the pixel on a 256-shade gray-scale palette is simply obtained by dereferencing the pointer. When the image initially is read into memory, the intensity of each pixel is identical to that generated by the SEM. As the image experiences each generation of iteration, the intensity of each pixel is reassigned to a value corresponding to MASS, PORE, or to the intermediate states EDGE, BOUNDARY, BORDER (identified by its address), MARKED_EDGE, and LOOSE_END; specific numbers for these states are defined in edge.h. Eventually, all pixels evolve into either MASS or PORE pixels.

Isolation and definition of pore perimeters

The image-analysis kernel first operates by evolving the image through three generations of cellular automata where the state of each pixel in each generation is defined by the state of its first- and second-nearest neighbors. Because each pixel is in contact with a set of neighboring pixels, and because each of the neighboring pixels have unique sets of neighbors, the state of a given pixel in each generation is influenced by the entire soup of pixels which make up the image.

The various stages of image evolution can be conceptually viewed as follows:

1. After program initialization:
   a. the image is read in the requisite format into memory (EDGE.C);
   b. the display is set to the graphics mode VESA 101 and the palette is set to a gray scale of 128 shades of gray in addition to eleven custom colors for highlighting purposes; and
   c. the image is then displayed, the name of the file shown, and the main menu is activated.

2. The user may manipulate the size or attributes of the image as a whole vis-à-vis menu-selectable functions such as clip, calibration, and so on (process.c).

3. Each pixel is tentatively classified into one of three states: PORE, BOUNDARY, and MASS.

4. A horizontal sweep thins all sets of adjacent BOUNDARY states to one pixel wide.

5. A clean-up horizontal sweep annihilates orphan sets of pixels that fail to qualify as recognizable structures.

6. In order to facilitate subsequent processing, a vertical sweep restores a one-pixel wide BOUNDARY between PORE and MASS regions; the BOUNDARY state subsequently evolves into the EDGE state. Each pixel is either PORE, MASS or EDGE at this point.

7. A horizontal sweep looks for EDGE pixels to pick up starting points for tracing the perimeter of individual pores.
   a. The first EDGE pixels discovered in the sweep are tagged MARKED_EDGE pixels; these pixels are the first elements in a set of pixel chains that define the pore perimeters; the chains are implemented as a linked list.
   b. As the program sweeps the image, it polls the neighbors of a given pixel for their state. The program starts polling from the left-most neighbor and proceeds counterclockwise; this is an arbitrary choice made by the programmer. Consequently, the pores are traced counterclockwise. When a neighboring EDGE pixel is discovered, it is attached to the chain as the next link. It follows that, for a given pore, the first EDGE pixel to be identified is the one that is the furthest north.
   c. Any subsequent EDGE pixels that join the linked list are also tagged MARKED_EDGE; each pixel in the chain is a node in that list and points to the next pixel in the chain.
   d. The chain terminates if it (a) closes in on itself (finds a MARKED_EDGE), (b) reaches the border of the image (finds a BORDER pore), or (c) finds only regular PORE neighbors (finds no neighboring pixels to continue the chain); a LOOSE_END is considered an error condition. Before reaching the LOOSE_END condition, the program polls both first- and second-nearest neighbors; polling of the second coordination shell of neighbors stabilizes the operation of the program.
   e. Once a pore perimeter has been specified, it will have a starting point, a pixel count, and a type (border, loose-end, or regular). Also, if the pore perimeter has a sufficiently high pixel count, it will eventually have a fractal dimension. These attributes are assigned to a field in the structure called "pore."

8. After all of the pores have been isolated and specified, the program checks if the fractal dimensions of all of the pores are required or if only the fractal dimension of the exposed surface is required. (PORE.EXE provides analysis of all pores; EDGE.EXE will analyze all pores if the switch "/pore" is specified when EDGE.EXE is scheduled from the command line.) If only the fractal dimension of the exposed surface is required, EDGE.EXE will choose the linked list with the largest pixel count as the profile for fractal analysis. The list of linked lists for fractal analysis is then passed to the Richardson algorithm.

If any of the above operations fail, the program exits with an appropriate error message.

Special libraries: Standard ANSI C: the display memory, consisting of up to 256 KB of 16-bit memory (note: 1 K = 1,024 bytes), is mapped directly into the host address space at locations reserved for graphics memory. This allows the programmer to utilize the PC assembly language "move" or "move string" instructions for rapid data transfers of data between host memory and display memory or within the display memory. In addition, a variety of read-and-write operations can be implemented in the display processor. Such operations accelerate graphics processing significantly compared to operations that are solely dependent on the host processor. For example, the programmer can read or write to display memory without having to wait for the horizontal or vertical retrace periods. Other write modes allow logical operations to be performed without the host processor assisting in the calculations. MORPH-II makes use of standard ANSI C function libraries that capitalize on the efficiencies accruing to such operations.

Source code: The full source code for EDGE.C is provided in appendix 1.

Acknowledgments

The authors wish to express their appreciation to the National Park Service National Center for Preservation Technology and Training, Natchitoches, La., for their support of this project, and to Mary Jane Coombs for editing the final manuscript.

References

Barton, C.C. and La Pointe, P.R., 1995, Fractals in petroleum geology and earth processes: New York, Plenum Press, 314 p.

Mossotti, V.G. and Eldeeb, A.R., 1992, The fractal nature of limestone *in* International Congress on Deterioration and Conservation of Stone, 7th, Lisbon, Portugal, 1992 [Proceedings], p. 621-630.

Mossotti, V.G., Eldeeb, A.R., Fries, T.L., Coombs, Mary Jane, Naudé, V.N., Soderberg, Lisa, and Wheeler, G.S., 2001, The effect of selected cleaning techniques on Berkshire Lee marble—a scientific study at Philadelphia City Hall: U.S. Geological Survey Professional Paper 1635, (CD-ROM), (URL http://geopubs.wr.usgs.gov/prof-paper/pp1635).

Mossotti, V.G., Eldeeb, A.R., and Oscarson, Robert, 1998, MORPH-I (Ver. 1.0), a software package for the analysis of scanning electron micrograph (binary formatted) images for the assessment of the fractal dimension of enclosed pore surfaces: U.S. Geological Survey Open-File Report 98-248, (URL http://geopubs.wr.usgs.gov/open-file/of98-248).

Mossotti, V.G., Lindsay, J.R., and Hochella, M.F., 1986, Acid rain weathering of Salem limestone—surface characterization of control material: U.S. Geological Survey Open-File Report 86-366, 36 p.

Sherwood, S.I. and Reddy, M.M., 1988, A field study of pollutant effects on carbonate stone dissolution *in* The engineering geology of ancient works, monuments and historical sites—preservation and protection: Rotterdam, Netherlands, A.A. Balkema Publishers, p. 917-923.

Turcotte, D.L., 1997, Fractals and chaos in geology and geophysics, 2d: Cambridge, Cambridge University Press, 398 p.

Appendix 1

Source code

CONTENTS

Page

EDGE.H ---------- 16

Header file for EDGE.C

EDGE.C ---------- 17

Displays and processes 256-shade gray-scale images in binary format

ENGINE.C ---------- 19

Numerical/statistical routines for calculating the fractal dimension of pores

LINKS.C ---------- 21

Routines for pore tracing and edge recognition as a linked list of pore structures

```
/*************************************************************************
 *                                                                       *
 *      Utility:    edge.h                                               *
 *      Platform:   DOS                                                  *
 *      Author:     Raouf Eldeeb                                         *
 *      Date:       Feb 2, 1995                                          *
 *      Purpose:    Header file for edge.c                               *
 *                                                                       *
 *************************************************************************/

# define    ENDFLAG                 20
# define    ERROR                   -1
# define    COLUMNS                 512
# define    SCREENWIDTH             640
# define    SCREENDEPTH             480
# define    TEXTDISPLAY             170
# define    TEXTHIGHLIGHT           171
# define    TEXTWARN                172
# define    WARNCOLOR               172
# define    HIGHLIGHTCOLOR          180
# define    MASSCOLOR               181
# define    PORECOLOR               183
# define    EDGECOLOR               184
# define    MARKEDEDGECOLOR         186
# define    PROBLEMEDGECOLOR        187
# define    REGULARPORECOLOR        188
# define    BORDERCOLOR             189
# define    LOOSEENDPORECOLOR       190
# define    BOUNDARYCOLOR           191
# define    MAX_REGRESSION_PTS      13
# define    LARGE_PORE              95
# define    PRINTMARGIN             1000


/************************** ENCAPSULATION MACROS ************************/
# define p2long(p)      ((((long)(unsigned)(void _seg*)(p))<<4)+(long)(unsigned)(
p))
# define left(x)        ((x)-1)
# define right(x)       ((x)+1)
# define upper_left(x)  ((x)-1-COLUMNS)
# define upper_right(x) ((x)+1-COLUMNS)
# define upper(x)       ((x)-COLUMNS)
# define lower(x)       ((x)+COLUMNS)
# define lower_left(x)  ((x)-1+COLUMNS)
# define lower_right(x) ((x)+1+COLUMNS)
# define col(p)         ((int)((p2long(p)-origin_long)%COLUMNS+1))
# define row(p)         ((int)((p2long(p)-origin_long)/COLUMNS+1))
# define xcoord(p)      ((int)((p2long(p)-origin_long)%COLUMNS)+columnStart)
# define ycoord(p)      (SCREENDEPTH-(int)((p2long(p)-origin_long)/COLUMNS))


enum states     { NONE , BOUNDARY , PORE , MASS , EDGE , MARKED_EDGE };
enum pore_type  { LOOSE_END , REGULAR_PORE , BORDER_PORE   };

struct node
{
     char   huge *p;
     struct node *next;
};

struct pore
{
    struct node *start;
    struct pore *next;
    int         type;
    unsigned    pix_count;
```



```
    double      dimension;
};


typedef struct
{
    char          low_bound;  /* The lower and higher cutoffs             */
    char          high_bound; /*      for the boundary                    */
    char huge *   origin;     /* The first pixel of the image             */
    char huge *   last_pix;   /* The last pixel of the image              */
    struct pore * head;
    char          report;     /* A flag to generate a report file         */
    char          debug;      /* A flag for debugging mode                */
    double        roughness;  /* the coefficient of roughness of the edge */
} IMAGE;


/************************** Prototypes **********************************/
int  get_input          ( void );
void process_image      ( void );
void warn               ( void );
void nudge              ( void );
void saveTempfile       ( void );
void readTempfile       ( void );
void terminate          ( char *);
void trace_edge         ( IMAGE * );
void calculate_dimension( IMAGE * );
void threshold_image    ( IMAGE * );
void thin_boundary      ( IMAGE * );
void fill_gaps          ( IMAGE * );
void clean_up           ( IMAGE * );
void print_BW_image     ( void );
void print_color_image  ( IMAGE * , int , int );
void color_pore         ( struct pore * , int  );
FILE *open_with_ext     ( char * , char * , char * );


extern char     filename[];     /* The name if the data file                */
extern char     message[];      /* A buffer for general purpose messages    */
extern char huge *origin;       /* The first pixel of the image             */
extern int      fg_mode;        /* the graphics mode                        */
extern long     imageSize;      /* The size of the image                    */
extern long     origin_long;    /* The first pixel of the image             */
extern float    micronsPerPixel;/* the number of microns per image pixel    */
extern unsigned imageRows;      /* The number of rows in an image           */
extern unsigned columnStart;
extern unsigned columnEnd;
extern unsigned rowStart;
extern unsigned rowEnd;
extern unsigned debugMode;
```

```
/*****************************************************************************
 *                                                                           *
 *      Utility:    edge.exe                                                 *
 *      Platform:   DOS                                                      *
 *      Author:     Raouf Eldeeb                                             *
 *      Date:       Feb 2, 1995                                              *
 *      Purpose:    Display and process 256 grayscale images as SEM data     *
 *                  data file                                                *
 *                                                                           *
 *****************************************************************************/


# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <conio.h>
# include <alloc.h>
# include <fcntl.h>
# include <ctype.h>
# include <dos.h>
# include <io.h>
# include "edge.h"
# include "key.h"
# include "fg.h"


/*************************  Global Variables  *************************/
unsigned    _stklen = 15000;
char        filename[50];     /* The name of the data file             */
char        message[100];
char huge * origin;           /* The first pixel of the image          */
int         fg_mode;          /* the graphics mode                     */
long        imageSize;        /* The size of the image                 */
long        origin_long;      /* the origin in (long) format           */
float       micronsPerPixel;  /* the number of microns per image pixel */
unsigned    imageRows;        /* The number of rows in an image        */
unsigned    rowStart;         /* start row of the image display        */
unsigned    rowEnd;           /* end  row of the image display         */
unsigned    columnStart;      /* start column of the image display     */
unsigned    columnEnd;        /* end column of the image display       */
unsigned    debugMode;
static void check_args        ( int , char *[] );
static void initiate_graphmode ( void );
static void read_image        ( char * );
static void set_grayscale     ( void );


/*****************************************************************************/
    int main( int argc , char **argv )
{
    check_args ( argc , argv );
    initiate_graphmode ();
    read_image ( filename );
    process_image ();
    terminate ( NULL );
}


/*****************************************************************************/
    void check_args ( int narg, char *argv[]  )
/*****************************************************************************
 * Checks the command line arguments
 * Accepts only the m flag or the d flag
 *****************************************************************************/
{
```



```
    int temp;

    if ( narg != 2 && narg != 3 )
    {
         puts ( "Usage: edge filename [/mxx]" );
         exit(1);
    }
    strcpy ( filename , argv[1] );
    micronsPerPixel = 17.4;
    if ( narg == 3 )
    {
        if ( argv[2][0] == '/' && argv[2][1] == 'd' )
            debugMode=1;
        else
            debugMode=0;
        if ( argv[2][0] == '/' && argv[2][1] == 'm' )
            if ( sscanf ( (argv[2]+2) , "%d" , &temp ) )
                micronsPerPixel *= temp/10.0;
    }
}


/*****************************************************************************/
    void initiate_graphmode ( void )
/*****************************************************************************
 * Invokes the SVGA graphics initiation for VESA mode 103 = 800x600x256
 * Exits the program if unable to initiate this graph mode
 *****************************************************************************/
{
    fg_mode = fg_init_vesa1();
    if ( fg_mode == FG_NULL )
    {
        puts ( "Graphics error : Unable to initiate SVGA mode 101" );
        exit(-1);
    }
    set_grayscale();
}


/*****************************************************************************/
    void set_grayscale ( void )
/*****************************************************************************
 * Sets a scale of 64 shades of gray for the first 64 colors
 * and two custom colors
 *****************************************************************************/
{
    unsigned color;

    fg_setpalette ( TEXTDISPLAY , 127 , 255 , 63 );
    fg_setpalette ( TEXTHIGHLIGHT , 255 , 127 , 31 );
    fg_setpalette ( TEXTWARN , 31 , 127 , 255 );
    fg_setpalette ( PORECOLOR , 64 , 10 , 32 );
    fg_setpalette ( MASSCOLOR , 0 , 255 , 0 );
    fg_setpalette ( EDGECOLOR , 255 , 255 , 0 );
    fg_setpalette ( PROBLEMEDGECOLOR , 255 , 0 , 255 );
    fg_setpalette ( LOOSEENDPORECOLOR , 255 , 0 , 127 );
    fg_setpalette ( BORDERCOLOR , 0 , 225 , 225 );
    fg_setpalette ( REGULARPORECOLOR , 255 , 0 , 0 );
    fg_setpalette ( HIGHLIGHTCOLOR , 255 , 31 , 31 );
    for ( color = 0 ; color < 128 ; color++ )
        fg_setpalette ( color , color << 1 , color << 1 , color << 1 );
}


/*****************************************************************************/
    void read_image   ( char *source )
```

```
/*************************************************************************
 * Read the data image                                                   *
 *************************************************************************/

{
 long block = 32768L , size;
 unsigned bytes_read;
 int filehandle;
 char huge *p;

 filehandle = open ( source , O_RDONLY | O_BINARY );
 if ( filehandle == -1 )
 {
    sprintf ( message , "Unable to open the file : %s" , source );
    terminate ( message );
 }
 imageSize = filelength(filehandle);
 imageRows = (unsigned)(imageSize/COLUMNS);
 origin = (char huge *) farmalloc ( imageSize );
 if (  origin == NULL )
   terminate ( "Unable to allocate memory for the image buffer\n");
 size=imageSize;
 p = origin;
 while ( size > 0 )
 {
     if ( block > size )
         block = size;
     bytes_read = read ( filehandle ,(void*) p  , (unsigned)block ) ;
     if ( bytes_read == 0 )
        terminate ( "Unable to read data from the data file" );
     p = p + block;
     size -= block;
 }
 close  ( filehandle );
}



/*************************************************************************/
    FILE *open_with_ext ( char *name , char *extension , char *mode )
/*************************************************************************
 * Opens a file with same name different extension                       *
 *************************************************************************/
{
    char *s=name, newname[50];

    s = strrchr ( name , '\\' );
    if ( s )
        strcpy ( newname , s+1 );
    else
        strcpy ( newname , name );
    s = strrchr ( newname , '.' );
    if ( !s )
        s = newname + strlen ( newname );
    else
        s += 1;
    strcpy ( s , extension );
    strcpy ( name , newname );
    return fopen ( newname , mode );
}


/*************************************************************************/
    int get_input ( void )
/*************************************************************************
 * Gets user input
```



```
 *************************************************************************/
{
    int c;

    c = getch();
    if ( !c )
       c = getch() << 8;
    else
       c = toupper ( c );
    return c;
}



/*************************************************************************/
    void  warn ( void )
/*************************************************************************
 * gives an audible warning                                              *
 *************************************************************************/
{
    sound ( 400 );
    delay ( 10  );
    sound ( 550 );
    delay ( 30  );
    nosound();
    delay ( 80 );
    sound ( 400 );
    delay ( 10 );
    sound ( 30 );
    delay ( 10 );
    nosound();
}


/*************************************************************************/
    void  nudge ( void )
/*************************************************************************
 * gives an small audible warning                                        *
 *************************************************************************/
{
    sound ( 1100 );
    delay ( 3  );
    nosound();
}


/*************************************************************************/
    void terminate ( char *s )
/*************************************************************************
 * reverts back to text mode and displays any exit messages              *
 *************************************************************************/
{
    fg_term();
    if ( origin )
        free ( origin );
    if ( s )
    {
        puts ( s );
        exit ( ERROR );
    }
    exit (1);
}
```

```
/*************************************************************************
 *                                                                       *
 *      Module:     Engine.c                                             *
 *      Platform:   DOS                                                  *
 *      Author:     Raouf Eldeeb                                         *
 *      Date:       Feb 2, 1995                                          *
 *      Purpose:    The Numerical/Statistical routines for calculating   *
 *                  the fractal dimension of pores                       *
 *                                                                       *
 *************************************************************************/

# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <conio.h>
# include <ctype.h>
# include <math.h>
# include "fg.h"
# include "key.h"
# include "paintjet.h"
# include "edge.h"

# define   FRACTAL_MULTIPLIER      1.5

static double   regress             ( double * , double * , int );
static double   dist2               ( char huge * , char huge * );
static double   roughness_factor    ( double * , double * , int );
static int      pore_perimeter      ( struct node * , double );
static int      calculate_roughness( IMAGE * );
extern int      m2printf            ( char * , ... );
extern int      m2puts              ( char * );


/*************************************************************************
   void calculate_dimension ( IMAGE *k )
/*************************************************************************
 *  Calculates the fractal dimension of the given edge
 *************************************************************************/
{
    unsigned n, perim;
    double pore_stats[MAX_REGRESSION_PTS];
    double xaxis[MAX_REGRESSION_PTS],yardstick;
    fg_box_t box;
    int c;

    for ( n = 0 , xaxis[0] = 0 ; n < MAX_REGRESSION_PTS ; n++ )
        xaxis[n+1] = log(2) + n*log(FRACTAL_MULTIPLIER);
    pore_stats[0] =  log( k->head->pix_count );
    yardstick = 2;
    perim = 10;
    for ( n = 0 ; perim > 3 && n < MAX_REGRESSION_PTS ;  n++ )
    {
        perim = pore_perimeter ( k->head->start , yardstick );
        if ( perim > 3 )
            pore_stats[n+1] = log ( perim );
        yardstick *= FRACTAL_MULTIPLIER;
    }
    k->head->dimension = regress ( xaxis , pore_stats , n );
    sprintf ( message , "Dimension = %4.3f" , k->head->dimension );
    fg_puts ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 , FG_ROT0 ,
              270 , 190 , message , fg.displaybox );
    sprintf ( message , "PixelCount = %.0f " , (float)k->head->pix_count );
    fg_puts ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 , FG_ROT0 ,
              270 , 145 , message , fg.displaybox );
    calculate_roughness ( k );
    c = get_input();
```



```
    if ( c == 'P' )
        if ( k->debug )
            print_color_image ( k , PJETPAPER , 0 );
        else
            print_color_image ( k , PJETPAPER , 4 );
    if ( c == 'T' )
        if ( k->debug )
            print_color_image ( k , PJETTRANSPARENCY , 0 );
        else
            print_color_image ( k , PJETTRANSPARENCY , 4 );
    fg_make_box ( box, 270, 120, SCREENWIDTH, 230);
    fg_fillbox ( 0, FG_MODE_SET, 0, box );
}


/*************************************************************************
   int pore_perimeter ( struct node *start , double scale )
/*************************************************************************
 * Returns the number of times a yardstick of length 'scale'
 * fits along the perimeter of the pore
 *************************************************************************/
{
    int count=0;
    double scale2=scale*scale, distance;
    struct node *current=start, *mark=start;

    while ( current->next != NULL && current->next != start )
    {
        distance = dist2 ( current->p , mark->p );
        if ( distance >= scale2 )
        {
            count++;
            mark = current;
        }
        current = current->next;
    }
    if ( scale2 < 4*dist2 ( current->p , mark->p ))
        count++;
    return count;
}


/*************************************************************************
  double dist2 ( char huge *pp , char huge *qq )
/*************************************************************************
 * Computes the square of the distance between two points
 *************************************************************************/
{
    int x1,x2,y1,y2;
    long p=pp-origin, q=qq-origin;

    x1 = (int)(p%COLUMNS);
    x2 = (int)(q%COLUMNS);
    y1 = (int)(p/COLUMNS);
    y2 = (int)(q/COLUMNS);
    return (double)(x1-x2)*(x1-x2) + (double)(y1-y2)*(y1-y2);
}


/*************************************************************************
  double regress ( double *xdata , double *ydata , int n_dat )
/*************************************************************************
 *  Performs linear regression on the data
 *  Returns the negative of the slope ( which is the fractal dimension )
                                    //  ---Used with demtil.c---
```

```
 ****************************************************************************/
{
    double sumxy=0, sumx=0, sumy=0, sumx2=0, slope;
    int n;

    for (n=0 ; n < n_dat ; n++ )
    {
        sumx  +=  xdata[n];
        sumy  +=  ydata[n];
        sumxy +=  xdata[n] * ydata[n];
        sumx2 +=  xdata[n] * xdata[n];
    }
    slope = ( n_dat * sumxy - sumx * sumy )/( n_dat * sumx2 -  sumx * sumx );
    return -1*slope;
}


/****************************************************************************/
 int calculate_roughness ( IMAGE *k  )
/****************************************************************************
 *  Calculates the roughness factor of the designated edge
 ****************************************************************************/
{
 struct node *node=NULL;
 double *xaxis, *yaxis;
 int n;

    xaxis = (double*)calloc ( k->head->pix_count , sizeof ( double ) );
    yaxis = (double*)calloc ( k->head->pix_count , sizeof ( double ) );
    node = k->head->start;
    for ( n=0 ; n < k->head->pix_count && node ; n++ , node = node->next )
    {
        xaxis[n] = col ( node->p );
        yaxis[n] = row ( node->p );
    }
    k->roughness = roughness_factor ( xaxis , yaxis ,  k->head->pix_count );
    sprintf ( message , "Deviation = %.2fμ" , k->roughness*micronsPerPixel );
    fg_puts ( HIGHLIGHTCOLOR , FG_MODE_SET , ~0 , FG_ROT0 ,
                    270 , 160 , message , fg.displaybox );
    free ( xaxis );
    free ( yaxis );
}


/****************************************************************************/
  double roughness_factor ( double *xdata , double *ydata , int n_dat )
/****************************************************************************
 *  Performs linear regression on the data
 *  Calculates the l_2 distance from the data points to its linear regression
 *  normalized by the number of data points
 *  Also calculates the number of times the data curve crosses the
 *  regression line
 ****************************************************************************/
{
    double sumxy=0, sumx=0, sumy=0, sumx2=0, slope, intercept, factor=0,temp;
    int n,crossings=0, sign;
    fg_line_t line;

    for ( n = 0 ; n < n_dat ; n++ )
    {
        sumx  +=  xdata[n];
        sumy  +=  ydata[n];
        sumxy +=  xdata[n] * ydata[n];
        sumx2 +=  xdata[n] * xdata[n];
    }
    slope = ( n_dat * sumxy - sumx * sumy )/( n_dat * sumx2 -  sumx * sumx );
```



```
    intercept = ( sumy - slope*sumx )/n_dat;
    line [ FG_X1 ] = columnStart;
    line [ FG_Y1 ] = SCREENDEPTH-intercept;
    line [ FG_X2 ] = columnEnd-1;
    line [ FG_Y2 ] = SCREENDEPTH-(intercept+slope*COLUMNS-1);
    fg_drawline ( HIGHLIGHTCOLOR , FG_MODE_SET , ~0 , FG_LINE_SOLID , line );
    sign = ( ydata[0] - slope*xdata[0] - intercept ) > 0 ? 1 : -1;
    for ( n= 0 ; n < n_dat ; n++ )
    {
        temp =   ydata[n] - slope*xdata[n] - intercept;
        factor += temp*temp;
        if ( temp * sign < 0 )
        {
           sign *= -1;
           crossings++;
        }
    }
    sprintf ( message , "Crossings = %d" , crossings );
    fg_puts ( HIGHLIGHTCOLOR , FG_MODE_SET , ~0 , FG_ROT0 ,
                        270 , 175 , message , fg.displaybox );
    return sqrt( factor/n_dat );
}
```

```
/*************************************************************************
 *                                                                       *
 *      Module:     Links.c                                              *
 *      Platform:   DOS                                                  *
 *      Author:     Raouf Eldeeb                                         *
 *      Date:       Feb 2, 1995                                          *
 *      Purpose:    The routines for pore tracing and edge recogintion   *
 *                  as a linked list of pore structures                  *
 *                                                                       *
 *************************************************************************/

# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <conio.h>
# include <alloc.h>
# include <fcntl.h>
# include <io.h>
# include "fg.h"
# include "edge.h"

# define is_leftedge(p)      ((long)( p - origin     ) % COLUMNS == 0 )
# define is_rightedge(p)     ((long)( p - origin + 1 ) % COLUMNS == 0 )
# define is_upperedge(p)     ((long)( p - origin     ) < (long)COLUMNS  )
# define is_loweredge(p,r)   ((long)( p - origin     ) > (long)COLUMNS*(r-1))
# define test_p(x)           ( *(x) == EDGE ? (x) : NULL )


static int      trace_pore  ( struct pore *, char huge *, IMAGE * );
static int      border_pore ( struct pore *, struct node *, char huge *, IMAGE *)
;
static void     reverse_list( struct node **, struct node ** );
static struct   node* backtrack ( struct node * , struct node * );
static struct   node* more_nodes( struct node * , int );
static struct   pore *more_heads( struct pore * , int );
static int      border_pixel    ( char   huge * , int );
static char huge* look_further  ( char   huge * , int );
static int      cleanup_pore    ( struct node * , int );
static int      count_pix       ( struct pore * );
static int      color_count     ( struct pore *  , int , int );

/************  Global variables specific to this module ******************/
static struct node *node_list;   /* the list of currently available nodes*/
static void   *list_mallocs[50]; /* the list of malloc allocations       */
static int    mallocs;

/*************************************************************************/
    void trace_edge ( IMAGE *k )
/*************************************************************************
 *  Creates the main linked list of all the lists  of pore edges
 *  'heads' is a single linked list of heads of the pore edge lists
 *************************************************************************/
{
    struct pore *largest_pore;
    struct pore *current_pore;
    char huge *current_pix= k->origin;
    int maxPixcount =0;

    mallocs = 0;
    threshold_image ( k );
    thin_boundary ( k );
    clean_up ( k );
    fill_gaps ( k );
    k->head = NULL;
    node_list = NULL;
    current_pore = more_heads ( NULL , 70 );
```



```
    node_list    = more_nodes ( NULL , 700 );
    while ( current_pix < k->last_pix )
    {
        while ( *current_pix != EDGE && current_pix < k->last_pix )
            current_pix++;
        current_pore->type = trace_pore ( current_pore , current_pix , k );
        if ( count_pix ( current_pore ) > maxPixcount )
        {
                largest_pore = current_pore;
                maxPixcount = current_pore->pix_count;
        }
        if ( k->debug && current_pore->pix_count > 20 )
        {
            color_pore ( current_pore , PROBLEMEDGECOLOR );
        }
        cleanup_pore ( current_pore->start , k->debug );
        if ( current_pore->next == NULL )
            more_heads ( current_pore , 30 );
        if ( current_pix < k->last_pix )
                current_pore = current_pore->next;
        else
                current_pore ->next = NULL;
    }
    color_pore ( largest_pore , EDGECOLOR );
    k->head = largest_pore;
    calculate_dimension ( k );
    while ( mallocs-- )
        farfree (list_mallocs[mallocs]);
}


/*************************************************************************/
    void threshold_image ( IMAGE *k )
/*************************************************************************
 *  Plots the initial display of the image
 *  Determines the threshold of the PORE, BOUNDARY, and MASS
 *  and changes the image data to the appropriate value
 *************************************************************************/
{
    int x,y;
    int color[5];
    char huge *p=k->origin;
    char *pp=(char*)p;

    color[PORE]=PORECOLOR;
    color[MASS]=MASSCOLOR;
    color[EDGE]=EDGECOLOR;
    for ( y = rowStart ; y > rowEnd ; y--, p += (long)COLUMNS, pp = (char*)p)
        for ( x = columnStart ; x < columnEnd ; x++ , pp++ )
        {
            *pp = (*pp > k->high_bound ? MASS :
                        (*pp < k->low_bound ? PORE : BOUNDARY));
            if ( k->debug )
                fg_drawdot ( color[*pp], FG_MODE_SET, 0, x, y);
        }
}

/*************************************************************************/
   void thin_boundary (   IMAGE *k )
/*************************************************************************
 * This function detects pore edges by scanning the data horizontally
 *************************************************************************/
{
    char last_state;    /* flag to indicate the last non-boundary state */
    int x, y;
    char huge *p=k->origin;
```

```
for ( y = rowStart ; y > rowEnd ; y-- )
{
    if ( *p == BOUNDARY )
    {
        *p = EDGE;
        if ( k->debug )
            fg_drawdot ( EDGECOLOR, FG_MODE_SET, 0, columnStart, y);
    }
    last_state = *p;
    p++;
    for ( x = columnStart+1 ; x < columnEnd ; x++ , p++ )
    {
        if ( *p == last_state )
            continue;
        switch( *p )
        {
        case BOUNDARY:
            *p = EDGE;
            if ( k->debug )
                fg_drawdot ( EDGECOLOR , FG_MODE_SET , 0 , x , y );
            if( *left(p)== EDGE )
            {
                if ( last_state == PORE )
                {
                    last_state = MASS;
                    while ( *p == BOUNDARY && x < columnEnd )
                    {
                        *p++ = MASS;
                        if ( k->debug )
                            fg_drawdot ( MASSCOLOR, FG_MODE_SET, 0, x, y);
                        x++;
                    }
                    if ( x == columnEnd )
                        p--;
                }
                else
                {
                    *(p-1) = MASS;
                    if ( k->debug )
                        fg_drawdot ( MASSCOLOR, FG_MODE_SET, 0, x-1, y);
                }
            }
            break;

        case PORE :
            if ( *left(p) == MASS )
            {
                *(p-1) = EDGE;
                if ( k->debug )
                    fg_drawdot ( EDGECOLOR, FG_MODE_SET, 0, x-1, y);
            }
            last_state = PORE;
            break;

        case MASS :
            if ( *left(p) == PORE )
            {
                *(p-1) = EDGE;
                if ( k->debug )
                    fg_drawdot ( EDGECOLOR, FG_MODE_SET, 0, x-1, y);
            }
            last_state = MASS;
            break;
        }
    }
```



```
    }
}

/*************************************************************************/
    void fill_gaps  ( IMAGE *k )
/*************************************************************************
 * This function detects pore edges by scanning the bit data vertically
 * and fills in any PORE to MASS or MASS to PORE transitions
 *************************************************************************/
{
    char last_state;      /* flag to indicate the last non-boundary state */
    int x, y;
    char huge *p, huge *q=k->origin;

    for ( x = columnStart ; x < columnEnd ; x++, q++  )
    {
        p = q;
        last_state = *q;
        for ( y = rowStart ; y > rowEnd ; y-- , p+=(long)COLUMNS )
        {
            if ( *p == last_state )
                continue;
            switch( *p )
            {
            case PORE :
                if ( *upper(p) == MASS )
                {
                    *upper(p) = EDGE;
                    if ( k->debug )
                        fg_drawdot ( EDGECOLOR, FG_MODE_SET, 0, x, y+1);
                }
                last_state = PORE;
                break;

            case MASS :
                if ( *upper(p) == PORE )
                {
                    *upper(p) = EDGE;
                    if ( k->debug )
                        fg_drawdot ( EDGECOLOR, FG_MODE_SET, 0, x, y+1);
                }
                last_state = MASS;
                break;

            default:
                last_state = *p;
            }
        }
    }
}

/*************************************************************************/
    void clean_up   ( IMAGE *k )
/*************************************************************************
 * Removes isolated EDGE pixels
 *************************************************************************/
{
    int x, y;
    char huge *p=k->origin;
    int s;

    for ( y = rowStart ; y > rowEnd ; y-- )
        for ( x = columnStart ; x < columnEnd  ; x++ , p++ )
            if ( *p == EDGE )
            {
```

```
            s = *right(p);
            if ( *left(p)==s && s != EDGE )
                switch ( s )
                {
                case MASS :
                    *p = MASS;
                    if ( k->debug )
                        fg_drawdot ( MASSCOLOR, FG_MODE_SET, ~0, x, y);
                    break;

                case PORE :
                    *p = PORE;
                    if ( k->debug )
                        fg_drawdot ( PORECOLOR, FG_MODE_SET, ~0, x, y);
                    break;
                }
        }
}

/*************************************************************************/
    int trace_pore ( struct pore *pore , char huge *start_pix , IMAGE *k )
/*************************************************************************
 *  Creates the circular linked lists for each individual pore edge
 *  'pore' is the node in the main linked list which will point to the
 *  current pore list
 *************************************************************************/
{
    struct node *current_node=node_list, *branch_node;
    char   huge *p=start_pix;       /* the pixel at which the pore starts */
    char   huge *q;

    if ( current_node == NULL || start_pix == NULL || pore == NULL)
    {
    /*      if ( k->debug )
     *          fprintf ( k->msg , "trace_pore: invalid parameter" ); */
        return (ERROR);
    }
    pore->start = current_node;
    do {
        current_node->p = p;
        if ( border_pixel ( p , imageRows ))
            return  border_pore ( pore , current_node , p , k );
        if ( !( q = test_p ( left(p))))
            if ( !( q =  test_p ( lower_left(p))))
                if ( !( q = test_p ( lower(p))))
                    if ( !( q = test_p ( lower_right(p))))
                        if ( !( q = test_p ( right(p))))
                            if ( !( q = test_p ( upper_left(p))))
                                if ( !( q = test_p ( upper(p))))
                                    if ( !( q = test_p ( upper_right(p))))
                                        if ( !( q= look_further(p, imageRows)))
                                        {
                                            *p = MARKED_EDGE;
                                            if ( k->debug )
                                                fg_drawdot ( MARKEDEDGECOLOR, FG_
MODE_SET, ~0 , xcoord(p) , ycoord(p) );
                                            branch_node = backtrack ( pore->start
, current_node);
                                            if ( branch_node )
                                            {
                                                current_node = branch_node;
                                                p = current_node->p;
                                                continue;
                                            }
                                            if ( k->debug )
```



```
                                                {
                                                    fg_drawdot ( PROBLEMEDGECOLOR , F
G_MODE_SET , ~0 , xcoord(p) , ycoord(p) );
                                                }
                                                if ( current_node->next == NULL )
                                                    more_nodes ( current_node , 200 )
;
                                                node_list = current_node->next;
                                                current_node->next=NULL; /* terminate
 the link */
                                                pore->type = LOOSE_END;
                                                return ( LOOSE_END );
                                            }
        if ( p != start_pix )
        {
            *p = MARKED_EDGE;
            if ( k->debug )
                fg_drawdot (MARKEDEDGECOLOR, FG_MODE_SET, ~0, xcoord(p), ycoord(p
));
        }
        p = q;                          /* defines the next pixel in the chain*/
        if ( current_node->next == NULL )
            more_nodes ( current_node , 200 );
        if ( p != start_pix )
            current_node = current_node->next;
    } while ( p != start_pix );
    *p = MARKED_EDGE;
    if ( k->debug )
        fg_drawdot (MARKEDEDGECOLOR, FG_MODE_SET, ~0, xcoord(p), ycoord(p));
    node_list = current_node->next;     /* the rest of the list of nodes   */
    current_node->next = pore->start;   /* closing the ring                */
    pore->type  = REGULAR_PORE;
    return ( REGULAR_PORE );
}

/*************************************************************************/
  int border_pixel ( char huge * p , int rows )
/*************************************************************************
 * Checks to see if p lies on the borders of the image
 *************************************************************************/
{
    long temp =  p2long (p) - origin_long;

    if (  temp % COLUMNS == 0 )
        return 1;
    if ( ( temp + 1 ) % COLUMNS == 0 )
        return 1;
    if ( temp < (long)COLUMNS  )
        return 1;
    if ( temp > (long)COLUMNS*(rows-1) )
        return 1;
    return 0;
}

/*************************************************************************/
int border_pore (struct pore *head, struct node *current_node, char huge *p, IMAG
E* k)
/*************************************************************************
 *  Treats the case of an incomplete pore at the edge of the picture
 *************************************************************************/
{
    char huge *q;
    int loose_ends=1;
    struct node*temp;
```

```
head->type = BORDER_PORE;
*p = MARKED_EDGE;
*(head->start->p) = MARKED_EDGE;
if ( k->debug )
{
     fg_drawdot ( MARKEDEDGECOLOR, FG_MODE_SET, 0,
                  xcoord(p) , ycoord(p) );
     fg_drawdot ( MARKEDEDGECOLOR, FG_MODE_SET, 0,
                  xcoord(head->start->p) , ycoord(head->start->p) );
}
if ( current_node->next == NULL )
     more_nodes ( current_node , 200 );
if ( is_upperedge(p) )    /* special case for the first row */
{
    q = p;
    loose_ends = 3;
    while ( is_upperedge(q+1) && *(q+1) == EDGE )
    {
        if ( current_node->next == NULL )
             more_nodes ( current_node , 200 );
        current_node = current_node->next;
        current_node->p = q++;
        *q = MARKED_EDGE;
        if ( k->debug )
            fg_drawdot ( MARKEDEDGECOLOR, FG_MODE_SET, 0,
                         xcoord(p) , ycoord(p) );
    }
}
if ( head->start != current_node )
{    /* did we start in the middle of the edge ?  */
     reverse_list ( &(head->start) , &current_node );
     loose_ends -= 1;
     p = current_node->p;
     temp = current_node->next;
     current_node->next=NULL;
     if ( k->debug )
         color_pore ( head , PROBLEMEDGECOLOR );
     current_node->next=temp;
}
while ( loose_ends )
{
    current_node->p = p;
    *p = MARKED_EDGE;
    if ( k->debug )
        fg_drawdot ( MARKEDEDGECOLOR, FG_MODE_SET, 0, xcoord(p), ycoord(p));

    if ( is_leftedge(p) || is_rightedge(p) || is_upperedge(p) )
        if ( !--loose_ends )
        {
            if ( current_node->next == NULL )
                more_nodes ( current_node , 200 );
            node_list = current_node->next;
            current_node->next= NULL;
            return BORDER_PORE ;
        }
    q = NULL;
    if ( !is_leftedge(p) )
        q = test_p ( left(p));
    if ( !q )
        if ( !is_leftedge(p) )
            q = test_p ( lower_left(p));
    if ( !q )
        q =  test_p ( lower(p));
    if ( !q )
        if ( !is_rightedge(p) )
```



```
            q = test_p ( lower_right(p));
     if ( !q )
        if ( !is_rightedge(p) )
            q = test_p ( right(p));
     if ( !q )
        if ( !is_leftedge(p) )
            q = test_p ( upper_left(p));
     if ( !q )
        q = test_p ( upper(p));
     if ( !q )
        if ( !is_rightedge(p) )
            q = test_p ( upper_right(p));
     if ( !q )
        q=look_further( p , imageRows );
     if ( !q )
     {
        node_list = current_node->next;
        current_node->next = NULL;
        *p = MARKED_EDGE;
        if ( k->debug )
            fg_drawdot ( PROBLEMEDGECOLOR, FG_MODE_SET, 0,
                         xcoord(p), ycoord(p));
        return ( LOOSE_END );
     }
     p = q;                      /* defines the next pixel in the chain */
     if ( current_node->next == NULL )
        more_nodes ( current_node , 200 );
     current_node = current_node->next;
  }
  return ERROR;
}


/*****************************************************************************/
  char huge *look_further ( char huge * p , int rows )
/*****************************************************************************
 * Expands the search for a neighbouring unmarked EDGE
 *****************************************************************************/
{
    char huge *q=NULL;
    int i;
    char huge *pts[16];          /* the array of second-order neighbors of p*/

    pts[0]=p-2+2*(long)COLUMNS;        /* lower lower left left pixel       */
    pts[1]=p-1+2*(long)COLUMNS;        /* lower lower left pixel            */
    pts[2]=p+2*(long)COLUMNS;          /* lower lower pixel                 */
    pts[3]=p+1+2*(long)COLUMNS;        /* lower lower right pixel           */
    pts[4]=p+2+2*(long)COLUMNS;        /* lower lower right right pixel     */
    pts[5]=p-2+(long)COLUMNS;          /* lower left left pixel             */
    pts[6]=p+2+(long)COLUMNS;          /* lower right right pixel           */
    pts[7]=p-2;                        /* left left pixel                   */
    pts[8]=p+2;                        /* right right pixel                 */
    pts[9]=p-2-(long)COLUMNS;          /* upper left left pixel             */
    pts[10]=p+2-(long)COLUMNS;         /* upper right right pixel           */
    pts[11]=p-2-2*(long)COLUMNS;       /* upper upper left left pixel       */
    pts[12]=p-1-2*(long)COLUMNS;       /* upper upper left pixel       */
    pts[13]=p-2*(long)COLUMNS;         /* upper upper pixel                 */
    pts[14]=p+1-2*(long)COLUMNS;       /* upper upper right pixel           */
    pts[15]=p+2-2*(long)COLUMNS;       /* upper upper right right pixel     */
    if ( is_rightedge(p))
       pts[3]=pts[4]=pts[6]=pts[8]=pts[10]=pts[14]=pts[15]=NULL;
    else if ( is_rightedge(right(p)))
       pts[4]=pts[6]=pts[8]=pts[10]=pts[15]=NULL;
    else if ( is_leftedge(p))
       pts[0]=pts[1]=pts[5]=pts[7]=pts[9]=pts[11]=pts[12]=NULL;
    else if ( is_leftedge(left(p)))
```

```
        pts[9]=pts[5]=pts[7]=pts[9]=pts[11]=NULL;
    if ( is_upperedge(p))
        pts[9]=pts[10]=pts[11]=pts[12]=pts[13]=pts[14]=pts[15]=NULL;
    else if ( is_upperedge(upper(p)))
        pts[11]=pts[12]=pts[13]=pts[14]=pts[15]=NULL;
    else if ( is_loweredge(p, rows))
        pts[0]=pts[1]=pts[2]=pts[3]=pts[4]=pts[5]=pts[6]=NULL;
    else if ( is_loweredge(lower(p), rows) )
        pts[6]=pts[1]=pts[2]=pts[3]=pts[4]=NULL;
    for ( i=0 ; i<16 && q == NULL ; i++)
        if ( pts[i] )
            q = test_p ( pts[i] );
    return q;
}


/*************************************************************************/
struct node * backtrack ( struct node * start_node, struct node * end_node )
/*************************************************************************
 * Returns the last node before 'end_node' which has a branching point
 *************************************************************************/
{
    struct node *current_node=start_node, *last_branch=NULL;
    char huge * p , huge *q;

    while ( current_node != end_node )
    {
        p = current_node->p;
        q = NULL;
        if ( !( q = test_p ( left(p))))
            if ( !( q = test_p ( lower_left(p))))
                if ( !( q = test_p ( lower(p))))
                    if ( !( q = test_p ( lower_right(p))))
                        if ( !( q = test_p ( right(p))))
                            if ( !( q = test_p ( upper_left(p))))
                                if ( !( q = test_p ( upper(p))))
                                    q = test_p ( upper_right(p));
        if ( q )
            last_branch = current_node;
        current_node = current_node->next;
    }
    if ( last_branch== start_node || last_branch == NULL )
        return NULL;
    return last_branch;
}


/*************************************************************************/
   int cleanup_pore ( struct node* start_node , int debug_mode )
/*************************************************************************
 * Turns any EDGE pixels which are adjacent to any MARKED_EDGES of the
 * current pore to MARKED_EDGE pixels which will later be turned
 *************************************************************************/
{
    struct node *current_node=start_node->next;
    char huge *p;

    while ( current_node != start_node && current_node != NULL )
    {
        p = current_node->p;
        if ( *left(p) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(left(p)) , ycoord(left(p)) );
            *left(p) = MARKED_EDGE;
```



```
        }
        if ( *lower(p) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(lower(p)) , ycoord(lower(p)) );
            *lower(p) = MARKED_EDGE;
        }
        if ( *right(p) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(right(p)) , ycoord(right(p)) );
            *right(p) = MARKED_EDGE;
        }
        if ( *upper(p) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(upper(p)) , ycoord(upper(p)) );
            *upper(p) = MARKED_EDGE;
        }
        if ( *lower(left(p)) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(left(p)) , ycoord(left(p)) );
            *lower(left(p)) = MARKED_EDGE;
        }
        if ( *lower(right(p)) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(lower(right(p))), ycoord(lower(right(p))));
            *lower(left(p)) = MARKED_EDGE;
        }
        if ( *upper(right(p)) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(upper(right(p))), ycoord(upper(right(p))));
            *upper(right(p)) = MARKED_EDGE;
        }
        if ( *upper(left(p)) == EDGE )
        {
            if ( debug_mode )
                fg_drawdot ( HIGHLIGHTCOLOR , FG_MODE_SET , 0 ,
                             xcoord(upper(left(p))) , ycoord(upper(left(p))));
            *upper(left(p)) = MARKED_EDGE;
        }
        current_node = current_node->next;
    }
    return 1;
}


/*************************************************************************/
    void reverse_list ( struct node **head , struct node **tail )
/*************************************************************************
 * Reverses the order of the linked list  ( non-recursive version )
 *************************************************************************/
{
    struct node *mark1 , *mark2 , *temp, *end;

    if ( *head == NULL || *tail == NULL || *head == *tail )
        return;
```

```
    end = (*tail)->next;          /* mark the next free node */
    mark1=*head;
    mark2=mark1->next;
    mark1->next = (*tail)->next;
    while ( mark1 != *tail )
    {
        temp = mark2;
        mark2 = mark2->next;
        temp->next = mark1;
        mark1 = temp;
    }
    temp = *head;
    *head = *tail;
    *tail = temp;
    (*tail)->next = end;
}


/*************************************************************************/
    struct pore * more_heads ( struct pore *last , int count )
/*************************************************************************
 * Allocates more links to the end of the heads 'last'
 *************************************************************************/
{
    struct pore *link, *nnew;

    if ( farcoreleft ()  < count*sizeof( struct pore ) )
        return (NULL);
    nnew = (struct pore *) farmalloc ( count*sizeof ( struct pore ));
    link = nnew;
    if ( link == NULL )
        terminate ( "Out of memory for the linked list");
    list_mallocs[mallocs++] = (void*)nnew;
    if ( last != NULL )
    {
        while (last->next != NULL )
            last =last->next;
        last->next=nnew;          /* connect to the existing list */
    }
    else
        last = nnew;              /*  creates a new list          */
    while ( count-- )
    {
        link->start = NULL;
        link->next = link + 1;
        link++;
    }
    (link-1)->next=NULL;
    return nnew;
}


/*************************************************************************/
    struct node * more_nodes ( struct node *last , int count )
/*************************************************************************
 * Allocates more nodes to the end of the node 'last'
 *************************************************************************/
{
    struct node *link, *nnew;

    if ( farcoreleft ()  < count*sizeof( struct node ) )
        return (NULL);
    nnew = (struct node *) farmalloc ( count*sizeof ( struct node ));
    link = nnew;
    if ( link == NULL )
        terminate ( "Out of memory for the linked list");
```



```
    list_mallocs[mallocs++]= (void*)nnew;
    if ( last != NULL )
    {
        while ( last->next != NULL)
            last = last->next;
        last->next=nnew;          /* connect to the end of the existing list */
    }
    while ( count-- )
    {
        link->p = NULL;
        link->next = link + 1;
        link++;
    }
    (link-1)->next=NULL;
    return nnew;
}

/*************************************************************************/
    void color_pore (  struct pore *h , int color )
/*************************************************************************
 * Draws the pixels of the pore in the chosen color
 *************************************************************************/
{
    struct node *current;

    if ( h == NULL )
        return;
    if (  h->start == NULL )
        return;
    current= h->start->next;
    while ( current != NULL && current != h->start )
    {
        fg_drawdot ( color, FG_MODE_SET, 0,
                     xcoord(current->p), ycoord(current->p) );
        current = current->next;
    }
}

/*************************************************************************/
    int color_count (  struct pore *h , int color , int count )
/*************************************************************************
 * Draws the pixels of the pore in the chosen color
 *************************************************************************/
{
    struct node *current;

    if ( h == NULL )
        return;
    if (  h->start == NULL )
        return;
    current= h->start->next;
    while ( count-- )
    {
        fg_drawdot ( color, FG_MODE_SET, 0,
                     xcoord(current->p), ycoord(current->p) );
        current = current->next;
    }
}


/*************************************************************************/
    int count_pix (  struct pore *h )
/*************************************************************************
 * Counts the number of nodes in a pore and updates the pix_count field
 *************************************************************************/
```

Appendix 2

USER'S GUIDE TO MORPH-II, a software package for the analysis of scanning-electron-micrograph images for the assessment of the fractal dimension of exposed stone surfaces

Victor G. Mossotti and A. Raouf Eldeeb
U.S. Geological Survey
Menlo Park, CA
2000

U.S. DEPARTMENT OF THE INTERIOR
U.S. GEOLOGICAL SURVEY

CONTENTS

Page

Overview of MORPH-II ---------- 29

Installing EDGE.EXE ---------- 30

- Hardware requirements ---------- 30
- Format requirements ---------- 30
- Display driver ---------- 30
- Extended memory driver ---------- 31
- Installing EDGE.EXE ---------- 32

Operation of EDGE.EXE ---------- 32

- Starting EDGE.EXE ---------- 32
- Menu functions ---------- 33
  - Clip ---------- 33
  - Edit ---------- 34
  - Erase ---------- 35
  - Calibr8 ---------- 35
  - Trace ---------- 39
  - Save ---------- 41
  - Density ---------- 41
  - Loss ---------- 41
  - Exit ---------- 42
  - Print ---------- 42

Acknowledgments ---------- 42

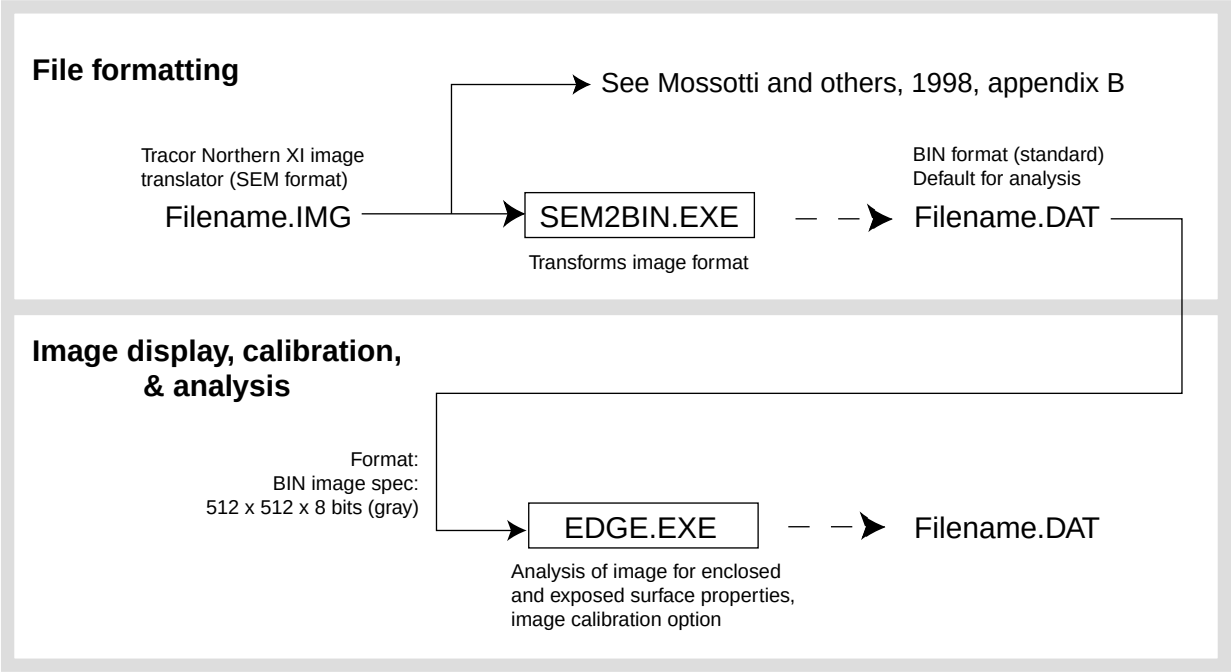


FIGURE 1.—Overview of MORPH-II.

Overview of MORPH-II

This manual documents the computer code for the software package MORPH-II. This package was created for the analysis of electron-micrograph images for the following material properties:

- fractal dimension of interior pore surface area,
- fractal dimension of the exterior surface area,
- fracture density of interior structure,
- length of structural displacements (*e.g.*, recession), and
- statistics on edge profiles.

Depending on how the specimen is prepared and presented to the electron microscope, the software developed for this work can estimate the fractal dimension(s) of either the enclosed-pore-space surface or the exterior exposed surface. The fractal dimension of the interior pore-space surface can be measured by examination of the cross-sectional profiles of the multitudinous exposed pores when the specimen has been transected on a plane through the bulk of the sample. Alternatively, the fractal dimension of the exterior surface area can be measured by analysis of the cross-sectional profile of the exterior surface of interest.

MORPH-II was developed for the analysis of electron-micrograph images stored in a binary file format and compliant with a 512 x 512 x 8-image specification. Figure 1 identifies the program modules in MORPH-II and the connection between MORPH-I and MORPH-II.[3]

[3]Source code and details on the design and operation of each of the modules in MORPH-I and MORPH-II are provided in Mossotti and others, 1998, and in the main text of this report, respectively.

We have made an effort to provide comprehensible instructions with a limited amount of background information. However, experience shows that very few users read thick manuals. The reader who wishes only for a cursory overview of the software operation can center her attention on the concise material provided in the text boxes.

Installing EDGE.EXE

Hardware requirements

The programs described in this report are executable on computers providing an MSDOS-compatible environment. Suitable command lines are provided by native MSDOS; OS/2 virtual DOS; and Windows 3.1, Windows 95, Windows 98, and Windows NT DOS emulation. The main image-analysis program requires Super VGA (SVGA) mode-101 display. Accordingly, an SVGA adaptor board with suitable drivers is required to run the program. A 60-MHz, 486-DX (math coprocessor) PC with at least 4 MB of RAM is recommended as the minimal system.

Format requirements

The image-analysis programs provided in MORPH-I and MORPH-II operate on linear binary files. Such binary files represent images consisting of 512 rows with 512 pixels per row, where each pixel is encoded with 8 bits on a 256-shade gray-scale palette; we will denote this image specification as 512 x 512 x 8. If the requisite file format is provided by the image source, the images can be displayed, calibrated, and analyzed without any further concern for format. Some SEMs may not provide suitable binary files with the requisite image specifications for the analysis programs in this package. In such cases, it is necessary to reformat the images collected from the image-generation device. The program SEM2BIN.EXE, as described in MORPH-I, accepts input from files created by the Tracor Northern XI image translator in the IMG format. This format is described in detail in MORPH-I. Ideally, the instrument producing the image will be capable of providing image files in the requisite image specification.

Display driver

The computer display memory consists of up to 256 KB of 16-bit memory (note: 1 K = 1,024 bytes) that can be configured in a variety of standardized display formats. These formats control the configuration of the display memory, the resolution of the display, the number of bits per pixel, the default character font, and the starting address of display memory. All display modes are classified into either alphanumeric (text) or graphic modes. There are important distinctions between these two types of modes. Text modes, which emulate the operation of the CGA color system, display preset alphanumeric characters under BIOS control. In contrast, the graphics modes allow the programmer to address all points on the display.

Although the flexible graphics features of the EGA/VGA standards provide the programmer with a set of powerful tools for control of the image in the graphics mode, it is important to recognize that the

EGA/VGA industry standards define only the minimal display operations. Super VGA (SVGA[4]), a superset of the VGA standard, improves on the VGA standard with better resolution, wider color selection, more memory planes, special character fonts, 132-character-wide screen formats, on-board arithmetic processors, and hardware zoom features.

Unless provision has been made for the operation of SVGA mode-101, EDGE.EXE will return an error message when it is started from the command line. EDGE.EXE changes the display mode to SVGA mode-101 when the program is started. When the program has finished its work, it returns the display mode to the standard VGA 640 x 480 x 4 before closing.

EDGE.EXE requires SVGA mode-101

Unlike the EGA and VGA boards, which include their own on-board BIOS, the SVGA adaptor boards generally require resident software drivers to control the SVGA display modes. These drivers, which are generally compliant with the VESA standard, must be installed before the main program, EDGE.EXE, is started. Please refer to the software installation instructions appropriate to the display adaptor on your computer.

Extended memory driver

Ordinarily, DOS loads itself into the first 640 KB of memory; this area is called **conventional memory**. Most non-Windows applications are confined to this part of memory. Because conventional memory can be quite restrictive for complex programs handling large data files (*i.e.*, images), the program can easily deplete the memory space available in conventional memory. If your system has more than 640 KB of memory, the memory in excess of 640 KB can be made available for operations in a memory area that is mapped beyond conventional memory.

Additional RAM in excess of 640 KB is mapped above the 1-MB mark; this area of memory is called **extended memory**. The memory area between 640 KB and the 1-MB mark is called **upper memory**; upper memory does not use RAM from the PC's memory chips and is reserved for memory on your hardware cards. For example, video memory is provided by your video adaptor card and is mapped into upper memory space.

Apart from a small 64-KB area above the 1-MB memory address, DOS cannot load and run programs in extended memory. However, because of a quirk in the way DOS addresses memory, DOS can directly access an additional 64 KB beyond the 1-MB mark; this directly-accessible segment is called **high memory**. When a PC has extended memory, the high memory is available to DOS and programs can run there as well as in conventional memory. The DOS extended-memory manager, HIMEM, turns

[4]Video Graphics Array (VGA) is the dominant PC graphics standard developed by IBM. VGA provides two graphics modes—640 x 480 pixels at 4 bits per pixel, and 320 x 200 pixels at 8 bits per pixel. Common SVGA resolutions are 800 x 600 pixels and 1,024 x 768 pixels at 8 bits per pixel.

the part of extended memory available to DOS into high memory area (HMA), where it can load and run one program at a time. Normally, DOS itself is loaded into HMA, freeing up a corresponding portion of conventional memory for other programs.

DOS can treat extended memory as a separate device and access it using an extended memory manager to handle the details. EDGE.EXE can hurdle the bounds of conventional memory if the appropriate extended memory manager has been loaded. Because DOS cannot access extended memory directly, an extended memory manager will have to be installed to make extended memory available to EDGE.EXE. DOS 6.22 includes an extended memory manager called HIMEM; if your machine has extended memory, you may want to use HIMEM to access and control it.

> If EDGE.EXE is executed from a Windows-3.1 or Windows-95/98/NT run line, the extended memory driver has already been loaded. If EDGE.EXE is run from a native-DOS 6.22 command line, the HIMEM.SYS device driver (or equivalent third-party memory manager) will have to be loaded on a line preceding the DOS=HIGH command in your CONFIG.SYS file. The following lines should be included in the CONFIG.SYS file:
>
> **DEVICE=C:\DOS\HIMEM.SYS**
> **DOS=HIGH**

Popular third-party memory managers include QEMM or 386MAX; these products offer more features than HIMEM.[5]

Installing EDGE.EXE

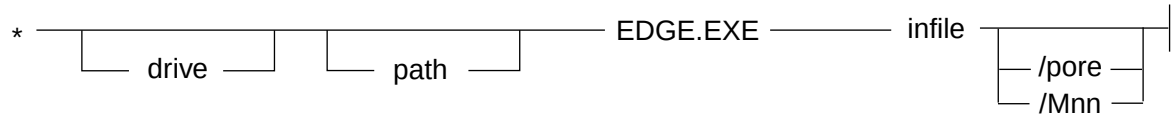
EDGE.EXE can be run from a diskette or from any hard disk partition. The program is installed to any partition simply by copying EDGE.EXE into the subdirectory of choice.

Operation of EDGE.EXE

Starting EDGE.EXE

The following diagram summarizes the command-line information required to properly start EDGE.EXE. In the diagram, "infile" refers to the name of the image file to be processed; if the file is not in the current directory, the name should be preceded by the full path (drive and directory containing the file). It is assumed that the file format conforms to the image file format described above. EDGE.EXE does not read any other formats.

[5]EMM386.EXE, a DOS driver, uses extended memory to simulate **expanded memory**. Expanded memory, an early solution to the limitations of conventional memory, is mapped into and accessed through upper memory. Expanded memory is no longer the standard of choice for increasing the memory available to program execution or data storage.



As a default, EDGE.EXE computes the fractal dimension and related statistics of the exposed surface of the specimen. The optional switch "/pore" cues the program to compute and report on the fractal dimension of every exposed pore in the cross-sectional image; implicit in the use of this switch is the assumption that the specimen has been prepared for enclosed-pore-surface analysis (image of cross-sectional cut through the rock).

The switch "/Mnn" advises the program on the linear resolution (magnification) of the current image. Because the definition of magnification is somewhat ambiguous, the symbol "nn" in the switch "/Mnn" represents the actual linear raster-scan resolution in units of μm/pixel. A default value of 17.4 μm/pixel, a raster-scan resolution corresponding to ≈10X magnification, is loaded into EDGE.EXE; use of the "/Mnn" switch is optional. The linear resolution is used by EDGE.EXE to calculate the distance between two planes through the images as controlled by the loss function (F9 from the main menu).

Menu functions

EDGE.EXE provides a continuously-updated graphical display of the image as each of the various menu functions are exercised. Except for print, each of these options can be invoked either by the appropriate function key, by typing the designated letter, or by highlighting the command with the cursor (left or right) and pressing ENTER. In the main menu, the user is given the following options:

Command	**F-key**	**Letter**	**Function**
Clip	F2	C	Clips a section of the image
Edit	F3	E	Edits individual pixels of the image
Erase	F4	R	Resets a region of the image
Calibr8	F5	B	Opens image-calibration tool
Trace	F6	T	Traces edge and finds fractal dimension
Save	F7	S	Saves the current image
Density	F8	D	Computes fracture density
Loss	F9	L	Opens caliper tool
Exit	F10	X	Exits the program
Print	——	P	Prints image to a Paintjet printer

A more detailed description of each of the menu items is provided in the following paragraphs.

CLIP Function key: F2 Fast key: C Return to main menu: ESC
Summary: clips and opens section of the image for analysis

When the exposed surface of the specimen is selected for analysis, most of the image information above and below the exposed surface is not used. In these situations, the file size can be reduced and the computation efficiency increased without compromise by discarding the portion of the image that

represents the bulk properties of the specimen and the void region above the surface. By selecting "clip" from the EDGE.EXE menu, the user can manipulate a set of graphical delimiters to isolate the region of the original image to be retained in the reduced file. After the section of the image to be retained has been selected with the delimiters on the displayed image, execution of the clip function (by pressing the ENTER key) will invoke a prompt for the name of the clipped file to be saved; CUT will always be the default extension. After entering the stem name for the newly-clipped file, the user presses the ESC key to return to the main menu. Before displaying the main menu, EDGE.EXE closes the original input file and loads the clipped file on the fly without the user having to reschedule EDGE.EXE.

EDIT Function key: F3 Fast key: E Return to main menu: ESC
Summary: edits individual pixels of the image

This function is invoked to edit specific pixels of the image or to adjust the overall brightness of the image. The edit screen has the following sub-options: F3, increases the brightness of every pixel in the image by a magnitude of 15 (on each activation of F3) such that the brightest pixel will not exceed maximum brightness (255); F4, decreases the overall brightness of the image by a magnitude of 15 (on each activation of F4) such that the dimmest pixel will not fall below 0.

Additional sub-commands are linked to a specific targeted pixel in the image that is highlighted by a cross-hair pattern. These sub-commands include:

Movement of cross-hair target for pixel selection:

NW (Home key) **NE** (PgUp key)

NORTH (Up arrow)

WEST (Left arrow) **EAST** (Right arrow)

SOUTH (Down arrow)

SW (End key) **SE** (PgDn key)

In addition, a display on the image shows the numerical values of the targeted pixel and of the first and second neighboring pixels in every direction. Pixel values less than that of the targeted pixel are displayed in green, those agreeing with it in red, and those exceeding it are in yellow.

+ increments the value of the targeted pixel
– decrements the value of the targeted pixel

U sets the upper limit of the boundary region to the value of the targeted pixel
L sets the lower limit of the boundary region to the value of the targeted pixel

ERASE Function key: F4 Fast key: R Return to main menu: ESC
Summary: rewrites selected region of the ----------image

This function is used to edit selected groups of pixels in the image by rewriting the intensity values of all of the pixels in the group. The erase function is useful to remove image noise, errors, or structures that do not contribute to the structural information of interest. The size and location of the region to be modified can be selected with the following sub-options:

Movement of region to be modified:

NORTH
(Up arrow)

WEST **EAST**
(Left arrow) (Right arrow)

SOUTH
(Down arrow)

Size of the region to be modified:
F5 or **PgUp** expands the edit region
F6 or **PgDn** shrinks the edit region

Activation of EDIT mode:
F3 toggles edit-mode ON/OFF, where
+ increments the intensity value of all pixels in the selected region, and
– decrements the intensity value of all pixels in the selected region.

CALIBR8 Function key: F5 Fast key: B Return to main menu: ESC
Summary: provides for image calibration

Image calibration is the first step in image analysis. A fundamental technical problem in automated SEM image analysis concerns the interpretation of contrast in an electron micrograph. The purpose of image calibration is to anchor the 256-shade gray-scale palette on unambiguous definitions of PORE (void) and MASS (solid phase) areas in the rendered image. In the image calibration step, the user's goal is to provide unambiguous definitions of the light and the dark areas of the image.

The definitions of "definitely dark" and "definitely light" pixels in the image are user-defined in terms of two threshold values representing the maximal image intensity of the logic state PORE and the minimal image intensity of the logic state MASS. For example, pixels with gray values less than or equal to the lower threshold (T_L) are defined as PORE. Similarly, pixels with gray values equal to or greater than the upper threshold (T_U) are defined as MASS. Areas of the image with uncertain identity will arise along

the edges, delimiting the pore space and solid phase of the specimen; pixels having an uncertain identity are logically defined as EDGE. As the state structure of the image evolves, all undefined pixels eventually emerge as either PORE or MASS, depending on the spatial environment of each pixel in the image.

Image calibration is conducted by visual inspection of the SEM image with the pixels tentatively identified as EDGE highlighted in red in the display. When an image is opened with the program EDGE.EXE, the calibration function first should be selected from the main menu. When the calibration function is invoked, the image appears with the highlighted EDGE pixels; on start-up, the EDGE state is based on default values for the upper and lower threshold values, T_U and T_L, respectively.

During the calibration procedure, the user can actively adjust the upper and lower thresholds to alter the population of pixels in the highlighted group. When the calibration utility is activated, the threshold parameters T_U and T_L are displayed below the image; the threshold parameters can be modified by exercising the following sub-options:

Modification of upper threshold (T_U):
F3 increments upper threshold (T_U)
F4 decrements upper threshold (T_U)

Modification of lower threshold (T_L):
F5 increments upper threshold (T_L)
F6 decrements upper threshold (T_L)

The user may elect to increase the sensitivity of the analysis to particular variegations along a contour line on the specimen by altering the threshold parameters to include pixels falling within a given range of gray values.

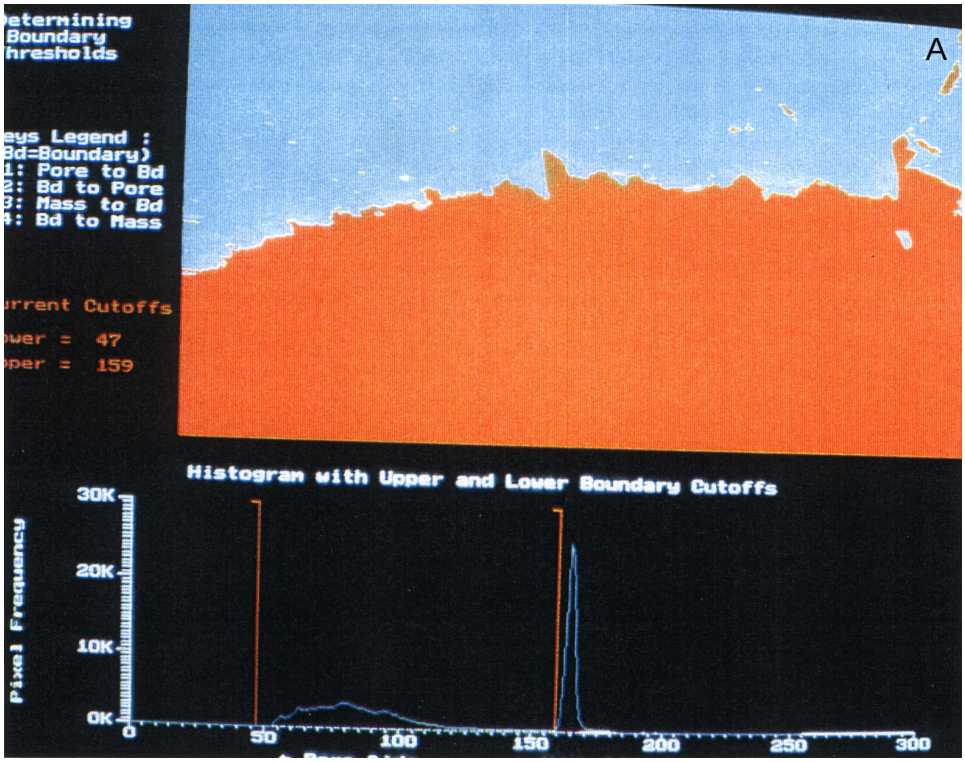
- The user can increase the sensitivity of the analysis to pixels on the light side of the lower threshold by moving the lower threshold to the left; this action increases the population of red pixels with values close to the lower threshold.

- The user can increase the sensitivity of the analysis to pixels on the dark side of the upper threshold by moving the upper threshold to the right; this action increases the population of red pixels with values close to the upper threshold.

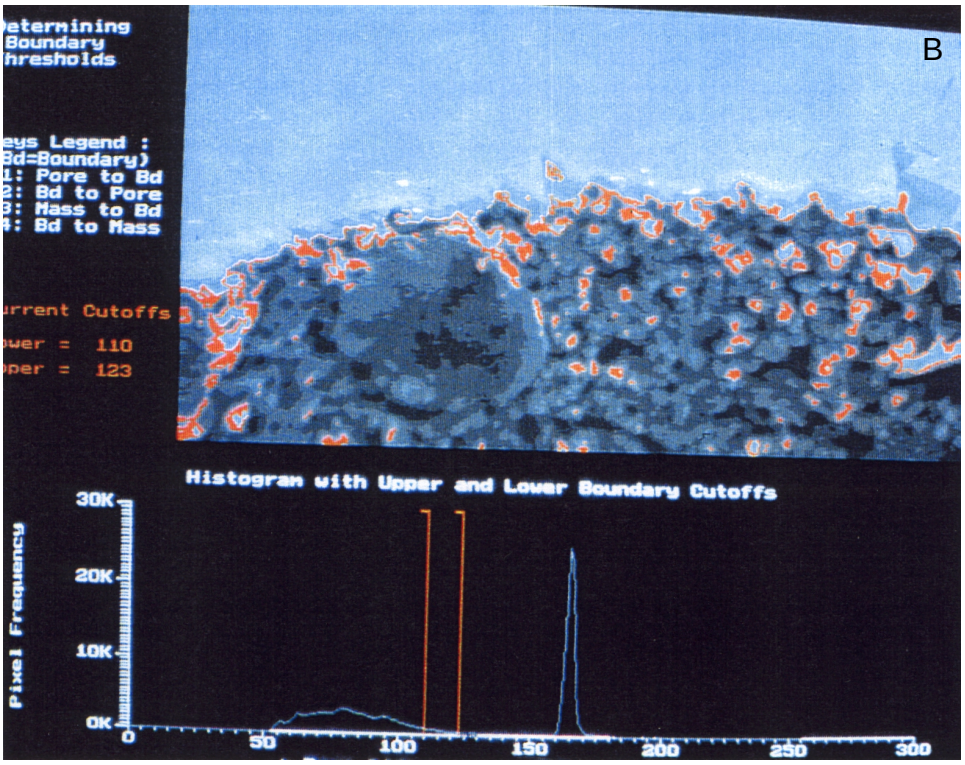
The adjustments of T_U and T_L are based on the user's experience with electron micrographs and on the user's knowledge of the physical structure of the material under examination. As a matter of computational efficiency, the user should attempt to minimize the width of the highlighted bands in the calibration display. In effect, the user is instructing the program to discover a boundary within the highlighted areas which conforms to a strict set of requirements, given the definitions of the PORE and MASS states as set by the threshold parameters.

Figures 2A and 2B, with a range of threshold settings, show examples of the calibration display[6] of a back-scattered electron micrograph of a highly polished (0.05 µm) specimen of Berkshire Lee marble (100X = 1.74 µm/pixel). Shown below the electron micrograph in each figure is the histogram of the electron-micrograph pixel intensities. As is common to most electron micrographs, the histogram shows a bimodal distribution with the darker pixels piling up near the left end of the histogram and the lighter pixels piling near the right end. The electron micrograph used for figure 2 provides an ideal example of the fundamental issues arising in the interpretation of contrast in an electron micrograph; note that the darker pixels correspond to the solid phase of the specimen and the lighter pixels correspond to void space.

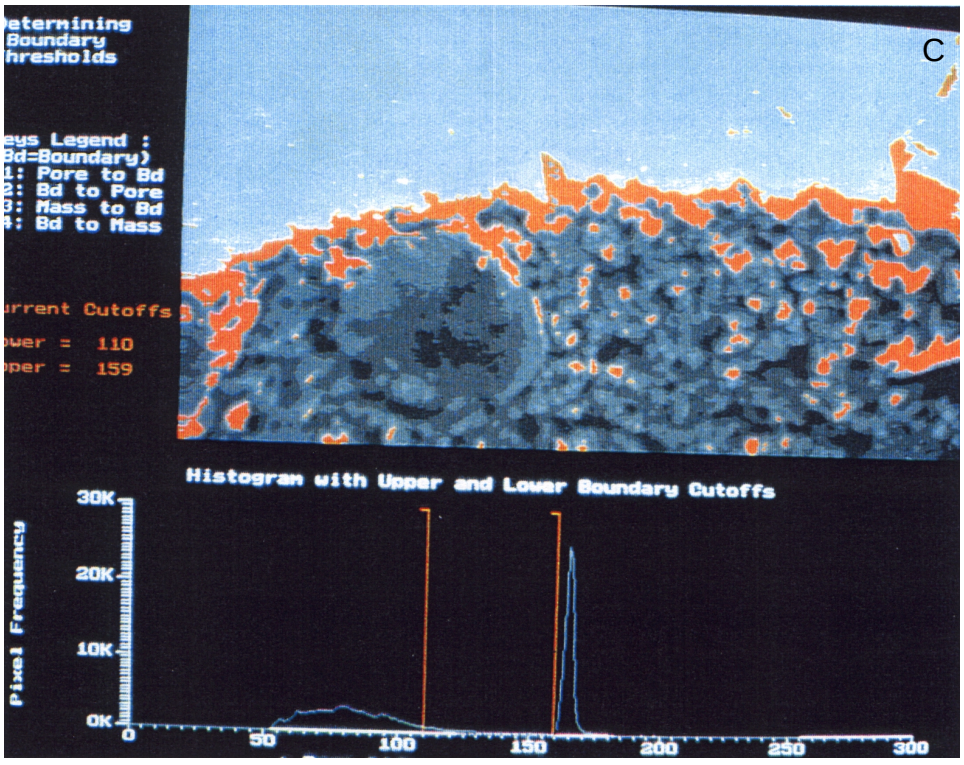
FIGURE 2.—Examples of the calibration display of a back-scattered electron micrograph of a highly polished (0.05 µm) specimen of Berkshire Lee marble (100X = 1.74 µm/pixel).

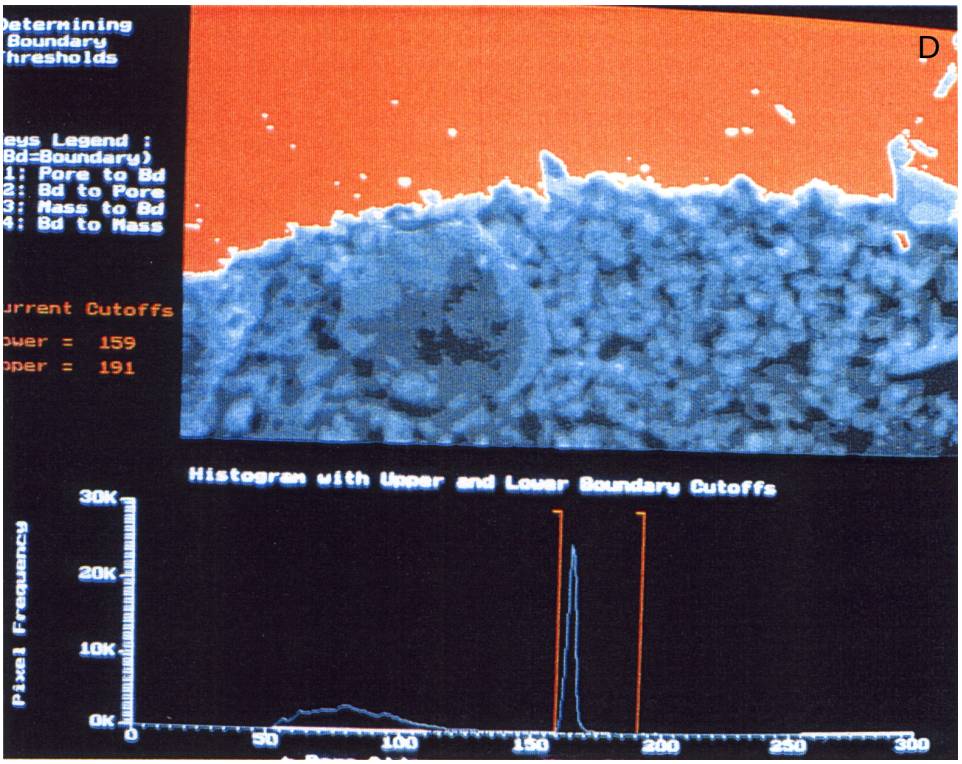


[6]The displays shown in figure 2 were obtained with the program PROFILE.EXE in the MORPH-I package (Mossotti and others, 1998).



In figure 2C, the lower and upper threshold levels at 110 and 159, respectively, isolate the set of EDGE pixels falling between the well-defined light and dark pixels; the EDGE pixels are highlighted in red in figure 2C. Under all circumstances, the pixels above the upper threshold are visually lighter than the pixels below the lower threshold.





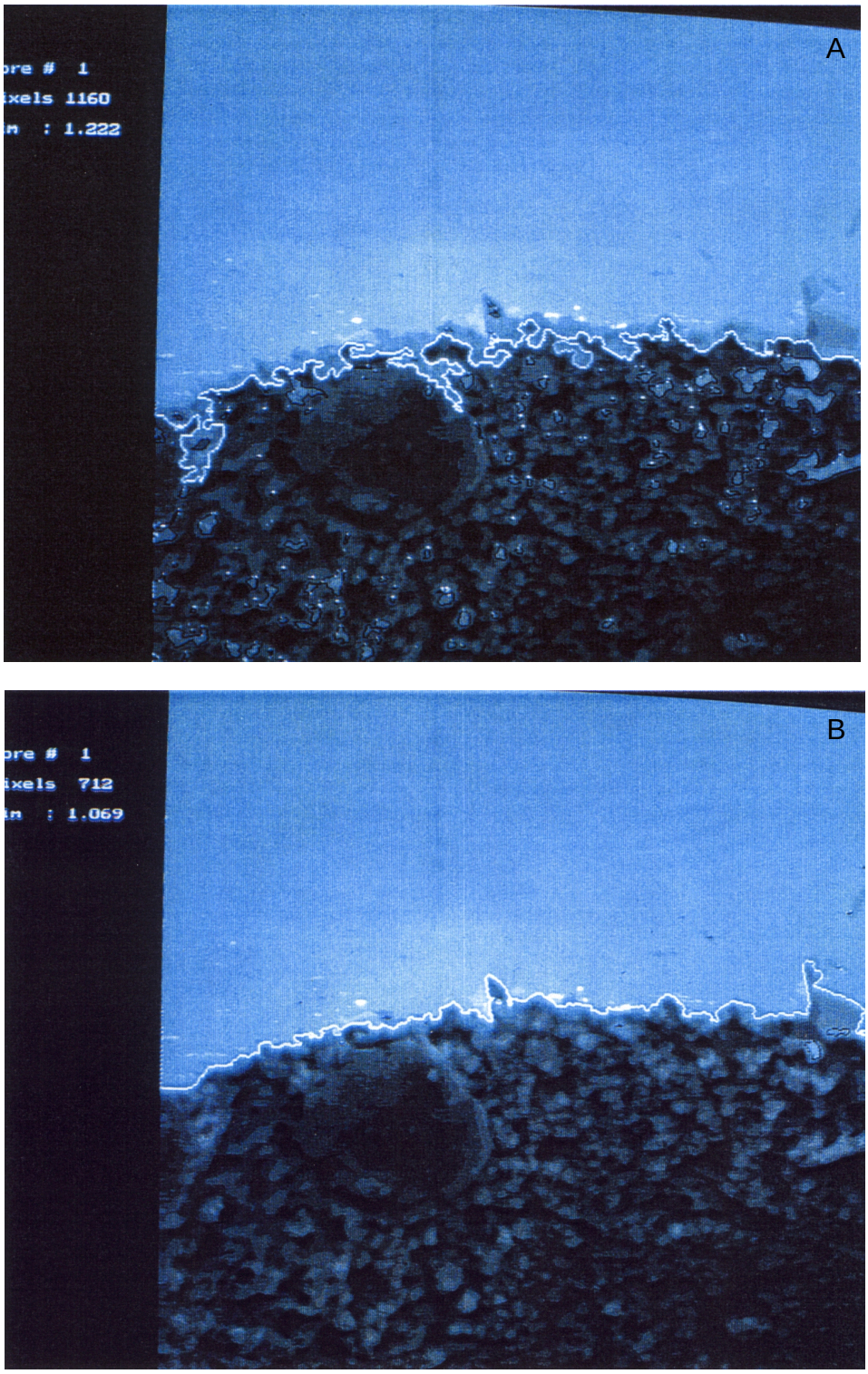
TRACE Function key: F6 Fast key: T Return to main menu: ESC
Summary: traces edge and reports fractal dimension

This function graphically highlights the profile over which the fractal and conventional statistics are computed. The linear regression line through the profile is usually displayed (not in the figures below, however) along with a summary of the statistical computations over the profile. The report displayed on the screen includes:

Dimension: fractal dimension of profile,
Crossings: number of profile crossings of regression line,
Deviation: standard deviation around regression line, and
PixlCount: number of pixels in profile.

Figures 3A and 3B show profiles displayed by the trace function corresponding to the calibration setups shown in figures 2B and 2C.

FIGURE 3.—Profiles displayed by the trace function.



SAVE Function key: F7 Fast key: S Return to main menu: ESC
Summary: saves current image with all changes

This operation will overwrite any previous files with the name of the current file without prompting the user. A message indicates that the image has been saved and gives the size of the image.

DENSITY Function key: F8 Fast key: D Return to main menu: ESC
Summary: computes fracture density

The density function displays a continuously-updated report on the percentage of PORE, MASS, and BOUNDARY pixels in an isolated area of the image and reports the percent fracturing in the selected area as the sum of the PORE and BOUNDARY percentages. The size and location of the isolated area on the image can be selected with the following sub-options:

Movement of selected area:

NORTH
(Up arrow)

WEST **EAST**
(Left arrow) (Right arrow)

SOUTH
(Down arrow)

Size of selected area:
F5 or **PgUp** expands the selected area
F6 or **PgDn** shrinks the selected area

LOSS Function key: F9 Fast key: L Return to main menu: ESC
Summary: opens caliper tool

This operation allows the user to measure the distance between two planes through the image by manipulation of a set of calipers superimposed on the image. The separation of the calipers is continuously updated and numerically reported on the display. The report, in units of μm, is based on linear pixel resolution of the image. The user can override the default resolution of 17.4 μm/pixel, a raster-scan resolution corresponding to 10X, by invoking the switch "/Mnn" on the command line when EDGE.EXE is scheduled. In the switch, "nn" represents the actual linear raster-scan resolution in units of μm/pixel. Use of the "/Mnn" switch is discussed in more detail above under "Image analysis: synopsis of the command line." The location, slope, and separation of the calipers can be controlled with the following sub-options:

Location of caliper set:
F3 moves set UP
F4 moves set DOWN

Slope of caliper set:
F5 INCREASES slope
F6 DECREASES slope

Separation of caliper set:
F7 OPENS calipers
F8 CLOSES calipers

EXIT Function key: F10 Fast key: X Return to main menu: ESC
Summary: exits the program

Before activating EXIT, the user should first save the current image, if desired, with the menu selection SAVE.

PRINT Fast key: P Return to main menu: ESC
Summary: generates ESC sequence and image file for a Paintjet printer

The print function, which explicitly is not selectable from the main menu, is activated simply by pressing the letter "P." Activation of the print function generates an ESC sequence and image file for a Paintjet printer with the PRN extension and prints an image on a reduced gray-scale palette to a Paintjet printer. Caution: this function will produce unpredictable results if the printer attached to printer port LPT1 is not a Paintjet printer. On completion of the print function, control is returned to the main menu.

Acknowledgments

The authors are indebted to the U.S. National Park Service for its support of this work.