5-1995

Parameter Estimation by Conditional Coding

Taylor Duersch
Utah State University

PARAMETER ESTIMATION WITH CONDITIONAL CODING

by

Taylor Duersch

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

of

MASTER OF SCIENCE

in

Statistics

UTAH STATE UNIVERSITY
Logan, Utah

1995

ABSTRACT

Parameter Estimation by Conditional Coding

by

Taylor Duersch, Master of Science

Utah State University, 1995

Major Professor: Kevin Hestir
Department: Mathematics and Statistics

Conditional coding is an application of Markov Chain Monte Carlo methods for sampling from conditional distributions. It is applied here to the problem of estimating the parameters of a computer-simulated pattern of fractures in an isomorphic, homotropic material under plane strain. We investigate the theory and procedures of conditional coding and show the viability of the technique by its application.

(122 pages)

ACKNOWLEDGMENTS

I am grateful to my graduate committee for their contributions of time and expertise to this thesis and toward my education in general. Kevin Hestir deserves special recognition as my major professor and mentor.

I appreciate the excellent teaching and support offered to me by all of the faculty and staff in the Department of Mathematics and Statistics at Utah State University.

I recognize my parents for their support and for the intangible characteristics they instilled in me that have made education both a success and joy to me.

Especially I thank my best friend, and companion, Melissa, for making it easier to accept my failures and wonderful to succeed.

Taylor Duersch

CONTENTS

LIST OF FIGURES

CHAPTER 1

CONDITIONAL CODING

To explain conditional coding in a rigorous way, we begin with a definition of conditional coding in mathematical terms. We follow this with a discussion of the topics and theory central to the mechanics of conditional coding. In particular, we will discuss the Gibbs distribution and the Metropolis algorithm with and without annealing.

1.1 Definition of Conditional Coding

Let $\mathbf{X}$ be a stochastic process. Suppose that we know how to simulate a realization $\mathbf{X}$ from a vector ω, of independent identically distributed uniform random variables on [0,1], using an algorithm g that maps ω to $\mathbf{X}$. We write that,

$$g(\omega) = \mathbf{X} \tag{1.1}$$

and call g a coding of $\mathbf{X}$. Let m be a function that represents measurements on $\mathbf{X}$ and let

$$\mathbf{M} = m(\mathbf{X}) + \mathbf{E}, \tag{1.2}$$

where $\mathbf{E}$ is a vector of random errors independent of $\mathbf{X}$. Let $f_{\mathbf{X} \mid \mathbf{M}}$ be the posterior distribution of $\mathbf{X}$ given $\mathbf{M}$. Because $g(\omega) = \mathbf{X}$, if ω_0 is a sample from the posterior distribution $f_{\omega \mid \mathbf{M}}$, then $g(\omega_0)$ is a sample from $f_{\mathbf{X} \mid \mathbf{M}}$. Sampling ω_0 from $f_{\omega \mid \mathbf{M}}$ and then taking $\mathbf{X} = g(\omega_0)$ to get a sample from $f_{\mathbf{X} \mid \mathbf{M}}$ is called conditional coding.

1.2 General Theory

Let $\mathbf{X}$, ω, g, $\mathbf{M}$, $\mathbf{E}$, and m be as given in the definition of conditional coding. Suppose that the probability density function of $\mathbf{E}$ is known and can be expressed in the form,

$$f_{\mathbf{E}}(\mathbf{e}) = c\exp(-h(\mathbf{e})), \tag{1.3}$$

with c a constant.

Now suppose that $\mathbf{X}$ is fixed. With $\mathbf{X}$ fixed, $\mathbf{M}$ can vary about $m(\mathbf{X})$ only according to the probability distribution of $\mathbf{E}$. By 1.3 we have the conditional probability density function

$$f_{\mathbf{M}\mid\mathbf{X}} = c\exp(-h(\mathbf{M} - m(\mathbf{X})). \tag{1.4}$$

Substituting 1.1 into 1.4 we get

$$f_{\mathbf{M}\mid\omega} = c'\exp(-h(\mathbf{M} - m\circ g(\omega)). \tag{1.5}$$

Bayes Rule for conditional probability distributions states that

$$f_{\omega\mid\mathbf{M}} = \frac{f_{\mathbf{M}\mid\omega}f_{\omega}}{f_{\mathbf{M}}}. \tag{1.6}$$

Here, $\mathbf{M}$ is considered fixed so that $f_{\mathbf{M}}$ is constant. Because ω is a vector of independent, identically distributed uniform random variables, f_{ω} is also constant. Hence,

$$f_{\omega\mid\mathbf{M}} = c''\exp(-h(\mathbf{M} - m\circ g(\omega))). \tag{1.7}$$

The probability distribution in 1.7 is called a Gibbs distribution. The Gibbs family of distributions has properties that make it possible to approximately sample any Gibbs distribution with a Monte Carlo simulation. Hence, sampling ω_0 from 1.7 is possible. Conditional coding is done by taking $g(\omega_0)$.

1.3 The Gibbs Distribution

The Gibbs Distribution is named after J.W. Gibbs, who did work in statistical mechanics. Spitzer (1971) asserts that in chemical physics the work of Gibbs produced mathematical models generally accepted as the simplest, most useful models of discrete gas.

Every Gibbs distribution is the stationary probability distribution of an aperiodic, irreducible Markov process. In general, the process state space is an arbitrary finite set. Gibbs distributions have the following form.

$$\pi(\omega) = c \exp(-h(\omega)/T) \tag{1.8}$$

In the context of chemical physics, ω represents the configuration of particles in a physical system or lattice. The constant c is a normalizing factor to insure that π is a probability measure. The function h measures the potential or energy associated with a configuration ω. The function h must have nonnegative range. The variable T measures temperature on a positive scale.

The temperature of a discrete gas affects the distribution of the configurations that the gas particles can assume. When T is small, the distribution of ω is concentrated on ω's where $h(\omega)$ is small. We use this fact later when we discuss the method for sampling from a Gibbs distribution called simulated annealing. Throughout the rest of our discussion about conditional coding we will refer to the function h as the energy function and to the variable T as temperature.

We have claimed without proof that 1.8 is the stationary distribution of an aperiodic irreducible Markov process. We will support this claim with two arguments. First, we introduce the Metropolis algorithm as a method for simulating an aperiodic, irreducible

Markov process. Second, we show that any distribution of the form 1.8 is the stationary probability distribution of a Markov process simulated by the Metropolis algorithm.

1.4 The Metropolis Algorithm

Metropolis et. al. (1953) introduced an algorithm to study the properties of interacting molecules in a lattice configuration. For a fixed $T > 0$ in 1.8, the Metropolis algorithm simulates a Markov chain $\omega_0, \omega_1, \omega_2, ...$ by the following method.

1. Begin with an initial state ω_i, where i is an integer index. Create ω_p by randomly perturbing some of the components of ω_i. The perturbation must be such that the probability of perturbing from ω_i to ω_p, hereby denoted $q(\omega_i \mid \omega_p)$, is the same as the probability of perturbing from ω_p to ω_i, hereby denoted $q(\omega_p \mid \omega_i)$.

2. Let $\omega_{i+1} = \omega_p$ with probability $p = min(1, exp(\frac{h(\omega_i)-h(\omega_p)}{T})$. Let $\omega_{i+1} = \omega_i$ with probability $1 - p$.

3. Repeat steps 1 and 2.

The condition that

$$q(\omega_i \mid \omega_p) = q(\omega_p \mid \omega_i) \ \forall \ \omega_i \text{ and } \omega_j \tag{1.9}$$

is necessary to insure that the Metropolis algorithm simulates a Markov process with a stationary distribution. We now show that any Markov process simulated by the Metropolis algorithm has a stationary distribution given by 1.8.

Let π be the stationary probability distribution function for a Markov process with transition matrix $\mathbf{P}$. By definition, stationary probability distribution functions must

satisfy

$$\boldsymbol{\pi}' = \boldsymbol{\pi}'\boldsymbol{P}. \tag{1.10}$$

Here $\mathbf{P}$ is the matrix of transition probabilities that describes the probability of moving from one state to another and π is a vector of stationary probabilities. Let S be a set of natural number indices, $1, 2, ..., N$, where N is intended to be the number of possible configurations ω can assume within a given physical system.

To show that 1.10 holds we must demonstrate that

$$\sum_k \pi_k P_{kj} = \pi_j \tag{1.11}$$

holds for all $j, k \in S$. The quantity $P_{k,j}$ is the probability of going from a configuration ω_k to a configuration ω_j in a single perturbation of ω_k. From our explanation of the Metropolis algorithm we see that for an arbitrary fixed j, such that $j \in S$, the following holds: for $k \neq j$,

$$P_{kj} = q(\omega_j \mid \omega_k) min(1, \exp(\tfrac{h(\omega_k) - h(\omega_j)}{T})) \tag{1.12}$$

and for $k = j$,

$$P_{kj} = P_{jj} = 1 - \sum_{k \neq j} q(\omega_k \mid \omega_j) min(1, \exp(\tfrac{h(\omega_j) - h(\omega_k)}{T})). \tag{1.13}$$

Applying 1.12 and 1.13 in 1.11,

$$\begin{aligned} \sum_k \pi_k P_{kj} \quad = \quad & (\sum_{k \neq j} c \exp(\tfrac{-h(\omega_k)}{T}) q(\omega_j \mid \omega_k) min(1, \exp(\tfrac{h(\omega_k) - h(\omega_j)}{T}))) + \\ & c \exp(\tfrac{-h(\omega_j)}{T})[1 - \sum_{k \neq j} q(\omega_k \mid \omega_j) min(1, \exp(\tfrac{h(\omega_j) - h(\omega_k)}{T}))]. \end{aligned} \tag{1.14}$$

Distributing, we get

$$\begin{aligned} \sum_k \pi_k P_{kj} \quad = \quad & (\sum_{k \neq j} c \exp(\tfrac{-h(\omega_k)}{T}) q(\omega_j \mid \omega_k) min(1, \exp(\tfrac{h(\omega_k) - h(\omega_j)}{T}))) + \\ & c \exp(\tfrac{-h(\omega_j)}{T}) - \sum_{k \neq j} c \exp(\tfrac{-h(\omega_j)}{T}) q(\omega_k \mid \omega_j) min(1, \exp(\tfrac{h(\omega_j) - h(\omega_k)}{T}))]. \end{aligned} \tag{1.15}$$

To show 1.11 we need only show that the sums

$$\sum_{k \neq j} c \exp(\tfrac{-h(\omega_k)}{T}) q(\omega_j \mid \omega_k) min(1, \exp(\tfrac{h(\omega_k)-h(\omega_j)}{T})) \tag{1.16}$$

and

$$\sum_{k \neq j} c \exp(\tfrac{-h(\omega_j)}{T}) q(\omega_k \mid \omega_j) min(1, \exp(\tfrac{h(\omega_j)-h(\omega_k)}{T})) \tag{1.17}$$

are equal. If 1.16 and 1.17 are equal, then 1.15 reduces to 1.11. The following argument shows the equality of the sums 1.16 and 1.17.

For any given $k \in S$ and $k \neq j$ one of two things is true.

1. It could be that $h(\omega_k) \geq h(\omega_j)$. In this case $exp(\frac{h(\omega_k)-h(\omega_j)}{T}) \geq 1$, so if we expand the sum in 1.16, the kth term is

$$c \exp(\tfrac{-h(\omega_k)}{T}) q(\omega_j \mid \omega_k). \tag{1.18}$$

   The kth term in the sum 1.17 is $c \exp(\frac{-h(\omega_j)}{T}) q(\omega_k \mid \omega_j) exp(\frac{h(\omega_j)-h(\omega_k)}{T})$, but this is equal to

$$c \exp(\tfrac{-h(\omega_k)}{T}) q(\omega_k \mid \omega_j). \tag{1.19}$$

   Since $q(\omega_k \mid \omega_j) = q(\omega_j \mid \omega_k)$ is a requirement of the Metropolis algorithm, we have that 1.18 is equal to 1.19.

2. Conversely, we could have that $h(\omega_k) < h(\omega_j)$. By symmetry, the argument from step (1) extends to this case proving that for all $k \in S$, 1.18 is equal to 1.19.

Since throughout our argument j was fixed arbitrarily, we have that 1.11 holds for all $j \in S$. Therefore, by the definition in 1.10, we have that any distribution of the form 1.8 is the stationary distribution of a Markov process simulated with the Metropolis algorithm.

We have established that the stationary distribution of a Markov process simulated by the Metropolis algorithm is a Gibbs distribution.

We can simulate Markov processes that have a stationary distribution that is a Gibbs distribution, but it is generally not known how many transition states we need to simulate before we can trust that our simulated Markov process is governed by the stationary distribution in 1.8. We will consider convergence more seriously when we apply conditional coding to an actual problem.

Assuming convergence, we have correctly proved that the Metropolis algorithm samples from a Gibbs distribution. This is the case when the Gibbs distribution is discrete. In the context of conditional coding, we use computer-driven algorithms to sample from a posterior distribution. Computers do discrete arithmetic using discrete representations of numbers. We defer to this fact and state that our previous arguments justify using the Metropolis algorithm to sample from a computer representation of a continuous Gibbs distribution. This is a point that is often overlooked and needs further investigation. General descriptions for sampling from a continuous Gibbs distribution using the Metropolis algorithm are cited by A.F.M. Smith and G.O. Roberts (1993).

1.5 Annealing

There are many methods besides the Metropolis algorithm for simulating and sampling from a Gibbs distribution. Such methods are generally classified as Markov chain Monte Carlo methods or MCMC methods for short. Among these methods Smith and Roberts (1993) include simulated annealing, the Metropolis algorithm, and the Metropolis-Hastings algorithm. Of interest to us here is a method called simulated annealing.

It is advantageous to discuss the effect of changing the temperature T during the course of the Metropolis algorithm. This topic is called annealing and can have some effect on the rate the Markov process converges to the Gibbs distribution from which we are interested in taking a sample.

Simulated annealing is a version of the Metropolis algorithm that methodically varies the temperature T during the simulation process. On page 37, *The New Lexington Webster's Encyclopedic Dictionary* (1990) defines annealing as "to improve the properties of by heating and then cooling." Simulated annealing gradually decreases T within the Metropolis algorithm. This allows us to sample from a Gibbs distribution that is highly concentrated at low energies.

Suppose that $h(\omega)$ is a positive valued function of $\mathbf{M} - m(\omega)$ where m is a measure on ω, perhaps made with some error, $\mathbf{E}$.

$$\mathbf{M} = m(\omega) + \mathbf{E} \tag{1.20}$$

If T is a small positive constant, then 1.8 is concentrated on points where $h(\mathbf{M} - m(\omega))$ is small. For such small T the Metropolis algorithm can require an exorbitant number of iterations before the simulated Markov process is governed by the Gibbs distribution. The idea behind simulated annealing is to run the Metropolis algorithm beginning with a large T that slowly decreases. Bertsimas and Tsistiklis (1993) assert that as the Metropolis algorithm iterates through a Markov process, slowly decreasing the value of T guides the Markov process to states concentrated on points where $h(\mathbf{M} - m(\omega))$ is small.

Difficulties arise with regard to simulated annealing when one tries to determine exactly how slowly to decrease T so that convergence is guaranteed without doing it so slowly that convergence is unduly delayed. Bertsimas and Tsistiklis (1993) offer arguments showing

that for any given Gibbs distribution, good cooling schedules exist but might begin at very high temperatures. Bertsimas and Tsistiklis go on to state that theoretically there are no rigorous results that make simulated annealing preferable to the Metropolis algorithm run alone with fixed T. There are, however, many examples of problems solved with simulated annealing where simulated annealing outperforms the Metropolis algorithm and other MCMC methods.

Once a computer program is in place to implement the Metropolis algorithm, simulated annealing is easy to implement as well. We have presented it here as a possible tool when sampling from a Gibbs distribution. How well it works depends on the application.

CHAPTER 2

A PROBLEM WITH NO ANALYTIC SOLUTION

We proceed to apply conditional coding to a research problem in the earth sciences. The problem is a parameter estimation problem. We want to estimate the parameters required by a particular algorithm to produce output with some predetermined characteristics. We can apply conditional coding to such problems and get good solutions. However, conditional coding and other applications of MCMC methods can be inefficient tools. Before using conditional coding, other reasonable approaches to finding a solution should be investigated. This chapter gives an explanation of the problem we desire to solve and examines the lack of an analytic solution. We also investigate relationships that might be exploited to solve the problem.

2.1 The Problem

Say that we can observe a patch of rock that displays a pattern of surface cracks. Martel et al. (1990) have taken what they know about the mechanics of fracture growth in rock to write a program that iteratively models fracture growth over time. At each iteration the program relies on a probability mechanism to decide if a fracture should grow and if so by how much. We will concentrate on the simplest model that assumes all rock fractures run parallel to each other and do not overlap.

The fracture generation program (fgp) that we use in this study is the one described by Martel et al. (1990). It is capable of generating nonparallel fractures at different orientations (see Appendix A.1). In the case where the fracture patterns are of parallel

non-overlapping cracks, the fgp requires the following input:

1. Three parameters that we will call *ngen*, *lmax*, and *gwsz*.
2. A set of starter cracks equal in number to the total number of fractures we desire in the simulation output. We code the starter cracks in the rows of a matrix with four columns. Each row contains the x y coordinates of starter crack endpoints.
3. A vector ω of uniform random numbers on [0,1] of sufficient length that every growth decision the program needs to make can be determined in turn using the values in the vector as output from a random number generator.

In the context of Chapter 1, the fgp is an explicit statement of $g(\omega)$.

The fgp iterates through a set of starter cracks *ngen* times, growing each crack according to a probability mechanism we explain later. The mechanism is complex enough that it is analytically impossible to study a simulation result in any traditional way to determine what parameters combined to produce it. The ability to do so would be useful to geologists. If geologists can match naturally occurring fracture patterns to parameters that reproduce those patterns in a simulation, then a categorization of fracture patterns is available on the basis of common physical characteristics that say something about how the fractures formed. Our goal is to match a set of simulation parameters, *ngen*, *lmax*, and *gwsz*, to a given fracture pattern.

Due to the probability mechanisms involved in simulation, unique solutions to this problem are not available. There are many parameter combinations that could produce a simulated fracture pattern to match some pattern we start with. Among these, some combinations are more likely to yield matching patterns than others. Likely combinations depend on the interaction of the different parameters with the probability mechanism

employed by the computer to model fracture growth. If possible, we want to associate fracture patterns with the parameter combinations that are most likely to produce a simulated match.

2.2 The Fracture Generation Program

The fracture generation program (fgp) is based on the recursive algorithm of Martel et al. (1990) to model fracture growth in homogeneous, isotropic, elastic materials under plain strain. The algorithm follows.

Define a growth area A. Consider a two-dimensional Poisson process operating at rate λ. We can randomly sample a point N from the probability mass function

$$\frac{(\lambda A)^k e^{-\lambda A}}{k!}. \tag{2.1}$$

N is an integer value. Place N points uniformly in A. At each point place a line segment of fixed length l_0, representing the beginning of a fracture.

Iterate through each fracture *ngen* times where *ngen* is a parameter value fixed in advance. The parameter *ngen* stands for the number of computer simulation generations that we want to occur. At each iteration fractures grow with probability p.

$$p = min(l/lmax, 1) \tag{2.2}$$

In the equation above $lmax$ is called the maximum cut-off length and l is the current length of the crack for which growth is being considered. The idea is that larger cracks have a higher probability of growth during a single generation than smaller cracks. Once a fracture is as long or longer than the length $lmax$, the probability of growth during every subsequent generation equals one.

The amount of growth a fracture achieves in a single generation also depends on the length of the fracture at the time growth occurs. Once we determine that a crack should grow, the new length is

$$l(1 + gwsz \cdot u). \tag{2.3}$$

The parameter $gwsz$ determines the possible amount of growth a crack may experience in a single generation. The variable u is a random variable chosen uniformly from the interval [0,1]. In the model, all growth is symmetric with respect to the midpoint of the fracture.

Figure 1 is an example of a fracture pattern created by the algorithm just explained. The parameters that produced this pattern were $N = 200$, $ngen = 100$, $lmax = 0.01$, and $gwsz = 0.1$. We note that N is a function of A and λ but these parameters are important only in that A defines the total area over which fractures are allowed to grow and from A and λ we get N.

From our description of the fracture generation algorithm we make some observations.

1. There are only three parameters we need to estimate that define the rules of fracture growth. They are $ngen$, $lmax$, and $gwsz$. We can observe N by counting. We will assume that l_0 is fixed and known.

2. The number of starter cracks we are required to have is the same as the number of fractures in the realization.

3. Fracture growth is independent of the vertical or horizontal position of the fracture. The only exception occurs during simulation when a fracture is close enough to the left or right boundary of the growth region that it grows beyond the boundary. Such fractures have censored length.

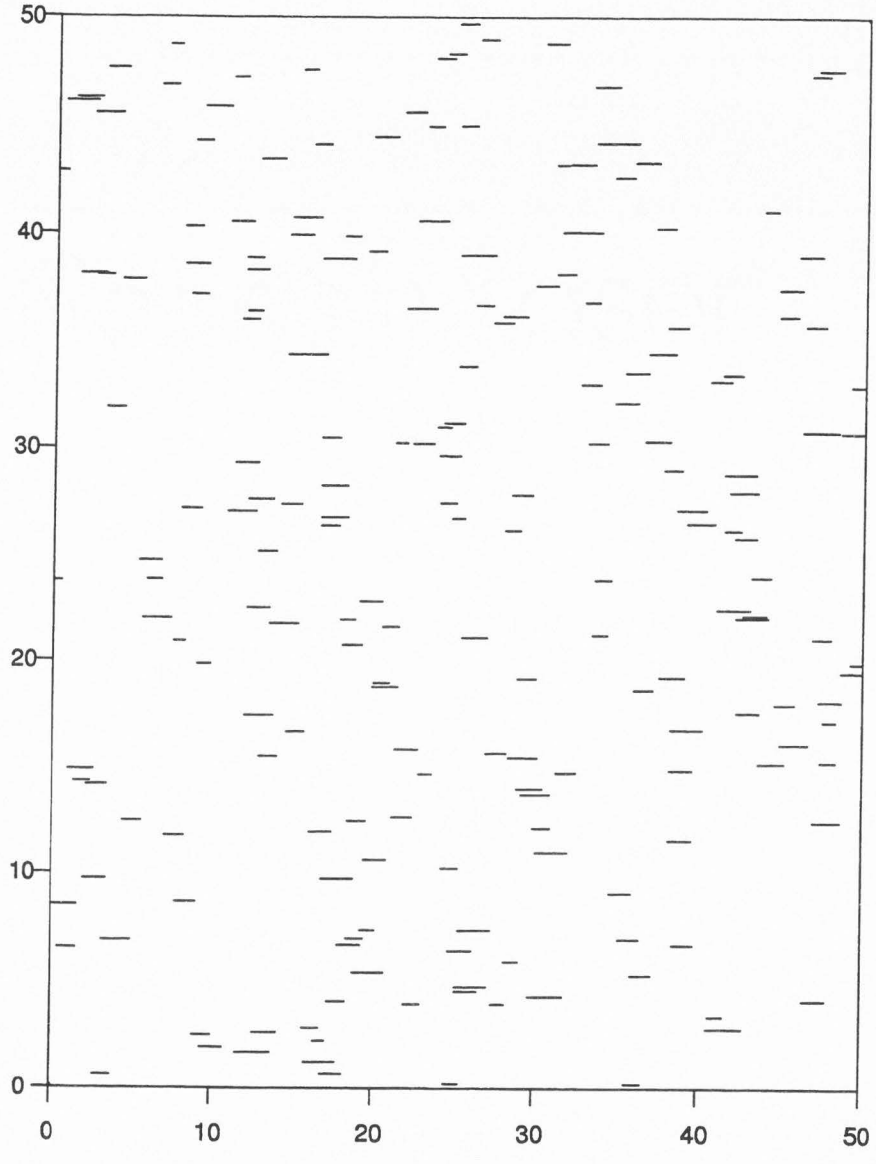

Figure 1. A simulated pattern of 200 parallel fractures.

Clearly, two different parameter sets can yield the same fracture pattern.

Suppose that Figure 1 is presented to us with the program used to produce it but that we have no knowledge of the specific parameter values employed by the program to yield this particular realization. After a brief look at some analytic relationships in the problem, we will devise a way of using conditional coding to estimate the parameters most likely to simulate Figure 1 through the fgp.

2.3 Some Analytic Relationships

In an attempt to learn more about the problem, we observe some relationships among fracture patterns and the growth parameters that produce them. To do this, divide the population of all possible fracture patterns into two groups. These groups may overlap. The first group of patterns we can simulate with appropriate values of *ngen*, and *gwsz* when $lmax \leq l_0$. The second group we can simulate when $lmax > l_0$.

When $lmax \leq l_0$, every fracture will grow at every iteration. The amount of growth in each crack at each iteration is $l \cdot gwsz \cdot u$, where l is the length of the fracture at the beginning of the iteration and u is a random variable uniform on [0,1]. In this case, it is easy to show that the expected length of a fracture i, $i = 1, 2, ..., N$, after $n = ngen$ iterations is given by

$$E[l_{i,ngen}] = l_0 \sum_{j=0}^{n} \binom{n}{j} (\frac{gwsz}{2})^j = (1 + \frac{gwsz}{2})^n. \tag{2.4}$$

This demonstrates that the expected value of fracture lengths for fixed *ngen*, *gwsz*, and $lmax \leq l_0$ can be modeled using an nth degree polynomial with positive coefficients. A unique polynomial exists for each value of $n = 1, 2, \ldots$. Figure 2 is a plot of expected values against *gwsz* for different values of n. Note that for every value of n there exists

a $gwsz$ so that any expected value greater than zero is possible. Note also, that as n gets bigger, the expected value curves are in close proximity to one another when $gwsz$ is small.

When $lmax > l_0$, things are different. Suppose that we have a long list of random variables uniform on [0,1]. The length of fracture i after iteration j is

$$l_{i,j} = l_{i,j-1} + (l_{i,j-1} \cdot gwsz \cdot u_{i,j}) I_{[u_{i+n,j} < l_{i,j-1}/lmax]}. \tag{2.5}$$

Here $u_{i,j}$ denotes a random variable uniform on [0,1].

We know that the amount of growth a fracture experiences on any given iteration is dependent on the growth probability $p = P(u_{i+n} < l_{i-1}/lmax)$. The distribution of p changes for each iteration where $l_{i-1} \neq l_i$. This is a difficult problem. It is difficult to predict even something as simple as the expected value of the crack lengths when $ngen$, $gwsz$, and $lmax$ are known.

In theory, we can construct a model that looks like 2.4 when $lmax > l_0$, but such a model incorporates indicator functions that lead to a very complex formulation. In the case that $lmax \leq l_0$, expression 2.4 serves to show that a given expected length is not unique to a single combination of growth parameters. Whether $lmax \leq l_0$ or not, we might find analytic inverse images based on the moments or a histogram of $\mathbf{X}$ but assigning a likelihood to them in the Baysiean sense is unrealistically complex.

Complications are also present whenever we try to match simulation parameters with fracture patterns that display cracks that are censored because they intersect the edge of the growth region. For each censored crack it is not known if the crack began outside the observable growth region extending in, or if it began inside the growth region extending out. A given set of parameters ($ngen_0$, $lmax_0$, $gwsz_0$) might simulate a given

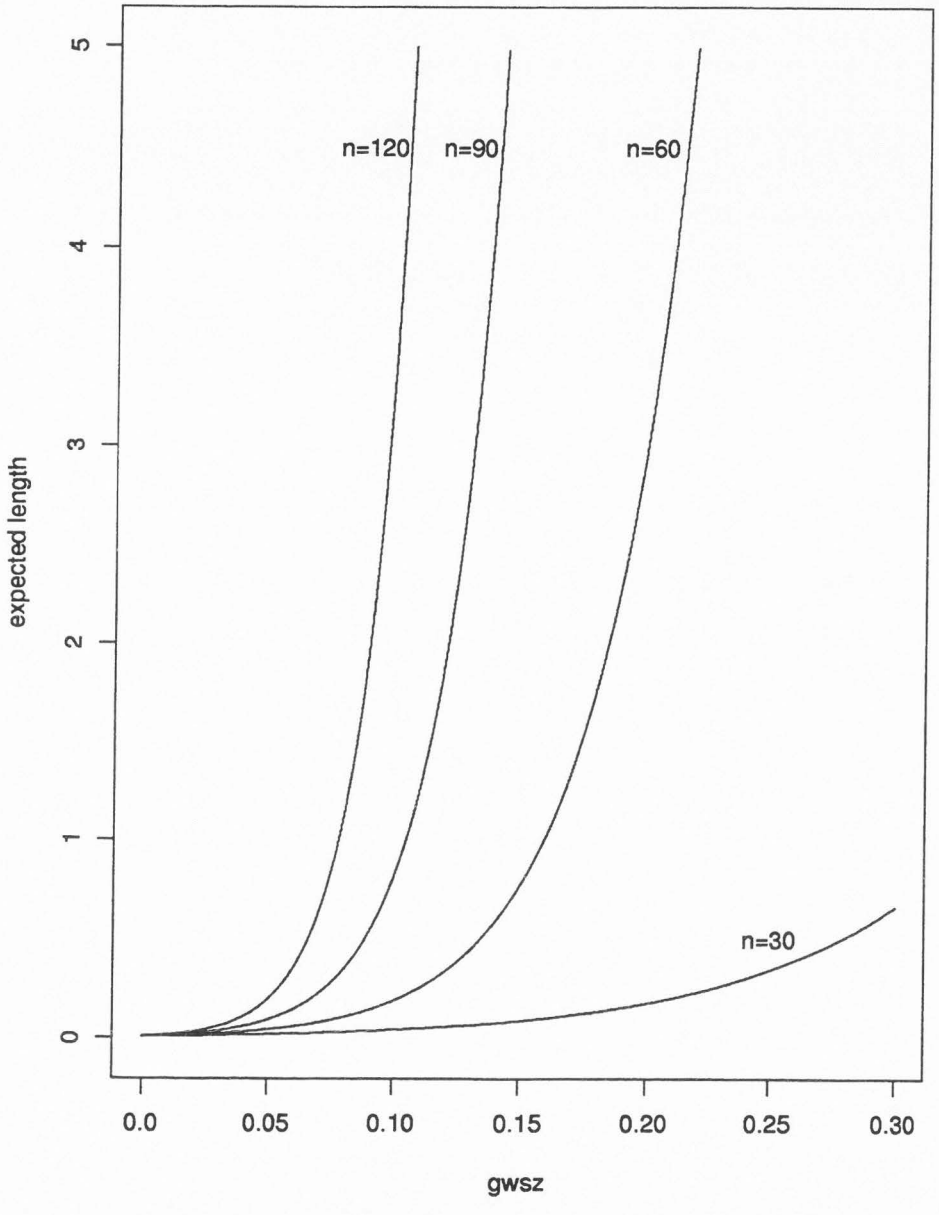

Figure 2. A sequence of polynomials that model expected fracture length for different values of n.

fracture pattern, $\mathbf{X}_0$, with censored cracks, but only if fractures near the boundary are started at correct positions inside or outside the observable growth region. Any attempt to match simulation parameters to a fixed fracture pattern must accommodate the difficulties present when some of the fracture lengths are censored.

It is evident that an analytic attempt at matching simulation parameters to a fixed outcome of fracture patterns is unwieldy and complex. Conditional coding provides a manageable way to solve this problem by sampling from the posterior distribution of ω given a fixed pattern of fractures and then constructing likelihood-like estimates based on the distribution of the sample.

CHAPTER 3

APPLYING CONDITIONAL CODING

There are several issues that we must address before we apply conditional coding to the problem presented in Chapter 2. These topics include energy functions, restricting the sample space, censored fractures, perturbing in the Metropolis algorithm, and determining convergence in distribution of a simulated aperiodic, irreducible Markov chain. This chapter ends with a conditional coding recipe applicable to the problem from Chapter 2.

3.1 Choosing an Energy Function

Choosing an energy function amounts to knowing the distribution of the errors, $\mathbf{E}$, in the statement $\mathbf{M} = m(\mathbf{X}) + \mathbf{E}$. This requires that for each problem we define $\mathbf{M}$ and m. The function m makes some true measurement on a realization $\mathbf{X}$. If we observe $m(\mathbf{X})$ with some independent random error, $\mathbf{E}$, then the value of the energy function, h, is dependent on the distribution of $\mathbf{E}$. For example, if the errors are normal with mean M and a standard deviation of 1, then by 1.3 $h(y) = \frac{1}{2}y^2$.

Suppose that we define $\mathbf{M}$ so that

$$\mathbf{M} = m(\mathbf{X}_0). \tag{3.1}$$

By this we mean that there is no measurement error. We stated previously that h is defined by the probability distribution of the measurement errors. If there is no measurement error, then we are free to choose h in many ways so long as h is nonnegative valued and achieves a minimum only for configurations of ω such that $m(g(\omega)) = m(\mathbf{X})$ or, equivalently, such

that $m(g(\omega)) = \mathbf{M}$. In general, the function h summarizes the difference between $\mathbf{M}$ and $m(\mathbf{X})$, where $\mathbf{X} = g(\omega)$.

When no measurement error is present, the best way to define h depends on the definition of m. In this section we consider three different definitions of m useful for solving the problem presented in Chapter 2. We also consider three corresponding ways of defining h. The reader is free to consider more and different choices of m and h.

3.1.1 Visual Matching

Two fracture patterns match visually if they look exactly the same. In the field of image analysis, Geman and Geman (1984) determined visual matches by comparing images pixel by pixel. Computers easily count the number of differing pixels between patterns. The resolution (number of pixels) a pattern enjoys determines how much time is required to determine a match. We can create our own version of resolution by superimposing a grid of arbitrary dimension over a fracture pattern. See Figure 3 for an example. Given a rectangular grid of fixed dimension, every fracture pattern imposes a new pattern on that grid.

Grid patterns form a matrix, $\mathbf{G} = (G_{i,j})$. If a fracture intersects the ith row and jth column that define the grid rectangle $r_{i,j}$, then the grid matrix stores a 1 in $\mathbf{G}_{i,j}$. Grid rectangles that do not bound a fracture, or some portion of a fracture, correspond to entries of 0 in the grid matrix. Two fracture patterns match if they have the same grid matrix of zeros and ones. The coarseness of the grid determines the quality of the match.

Let h be a function that sums the absolute difference of corresponding entries in a grid matrix $\mathbf{A}$ and a grid matrix $\mathbf{B}$. Let matrix $\mathbf{A}$ name the grid matrix imposed on a fixed

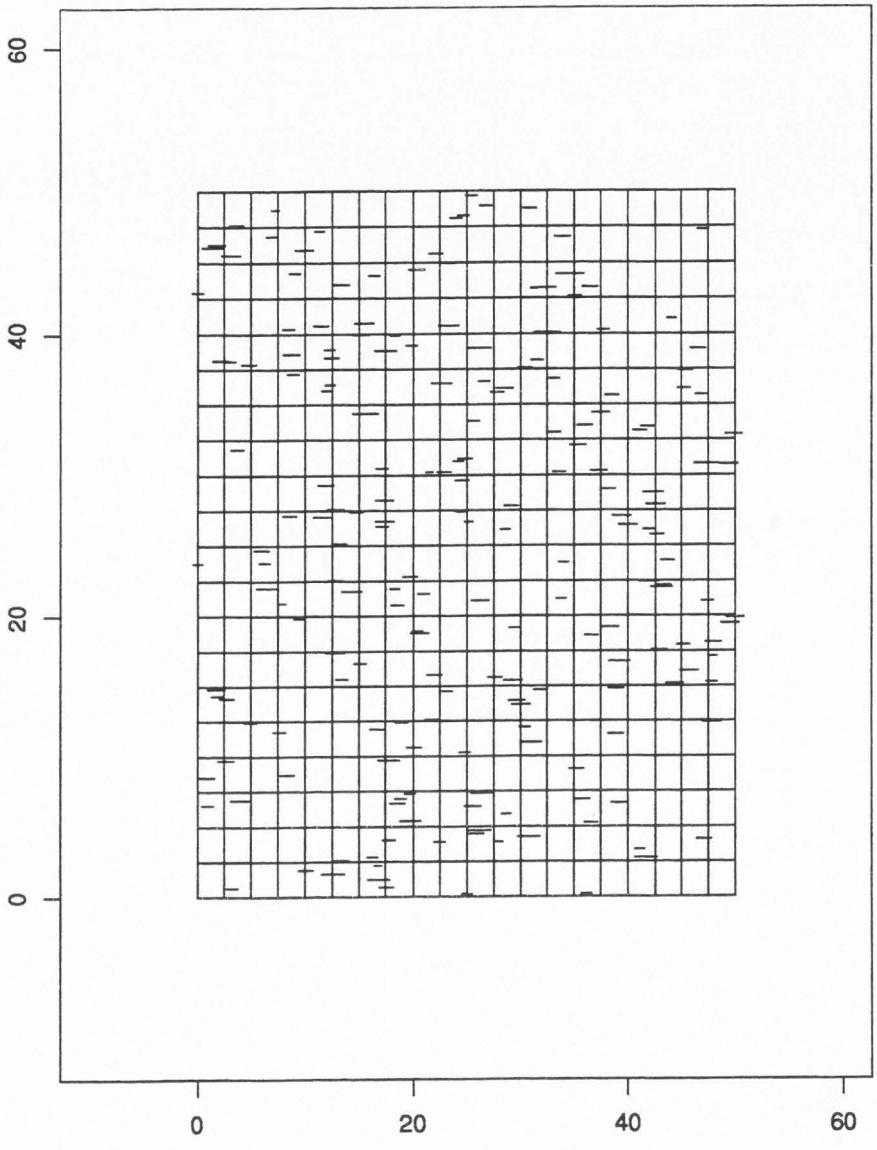

Figure 3. An example of a rectangular grid placed over a fracture pattern. Squares intersected by a fracture or any portion of a fracture have a value of 1 otherwise they have a value of 0.

fracture pattern $\mathbf{X}_0$. Let $\mathbf{B}$ name the grid matrix of a simulated fracture pattern, $\mathbf{X}$. Let $m(\mathbf{X}_0) = \mathbf{A}$ and $m(g(\omega)) = \mathbf{B}$, where $g(\omega) = \mathbf{X}$. Symbolically,

$$h(\omega) = \sum_{i=1}^{r} \sum_{j=1}^{c} |\mathbf{A}_{i,j} - \mathbf{B}_{i,j}|, \tag{3.2}$$

where r is the number of rows in $\mathbf{A}$ and $\mathbf{B}$, and c is the number of columns. To adjust the resolution of a match in h, change r and c. The larger the values of r and c, the better h determines different patterns. As the resolution becomes coarse, the value of m ceases to be unique for differing patterns of fractures. Meaningful samples of the vectors ω that simulate fracture patterns $\mathbf{X}$ such that $m(g(\omega)) = \mathbf{A}$ require that the function m is not too vague.

For the problem presented in Chapter 2, visual matching has one drawback. Namely, $m(\mathbf{X}_0)$ depends on both the horizontal and vertical position of each fracture defined in $\mathbf{X}_0$. Except for those fractures in $\mathbf{X}_0$ that intesect the boundaries of the growth region, the position of a fracture does not affect what the fractures tell us about *ngen*, *lmax*, and *gwsz*. Statistical pattern matching proves more flexible than visual matching in the context of the current problem. The next two examples of h match statistical information between fracture patterns.

3.1.2 Method of Moments

Using the idea that under suitable conditions two probability distributions match if they have matching theoretic moments, define h as follows. Let $\mathbf{M}$ denote the vector of sample moments from the fracture lengths in a fixed pattern, $\mathbf{X}_0$. The number of sample moments must be larger than one and preferably larger than three. Simulate a fracture pattern $\mathbf{X}$ and compute a vector $\mathbf{l}$ of fracture lengths. Suppose N is the total number of

fractures present in the natural fracture pattern. Define $\mathbf{B}$ so that

$$B_i = \frac{1}{N} \sum_{i=1,N}^{k} l_j^i. \tag{3.3}$$

The index i denotes the ith moment so that the length of $\mathbf{B}$ is the number of sample moments, k, that we set in advance. The variable l_j is the length of the jth fracture in the simulated pattern.

Let $m(g(\omega)) = \mathbf{B}$ and $m(\mathbf{X}_0) = \mathbf{M}$. Define h by

$$h(\omega) = \sum_{i=1}^{k} \mid B_i - M_i \mid . \tag{3.4}$$

The constants M_i are the ith sample moments of the fracture lengths in the fracture pattern $\mathbf{X}_0$. The constant k is a positive integer corresponding to the length of $\mathbf{M}$. The function h measures differences in fracture patterns better for N large. If N and k are large enough, h will not achieve a minimum value unless the fractures in X are from the same population as the fractures in the natural pattern. If we assume that all of the important information about a fracture pattern is summarized in the lengths of its fractures and that those lengths are uncorrelated to their position in the growth area, then using 3.3 in 3.4 makes sense. The energy function h as defined here makes no sense in the case of small N or with a natural pattern where more than a small proportion of the fractures extend outside the growth area. Censored cracks are a problem because the true length of the fracture is not known.

The h proposed here has a range on the nonnegative real numbers. This means that even though h achieves a true minimum at zero, we will need to consider two patterns as a match if h is close to zero. This has the disadvantage that we must interpret what *close* to zero means.

3.1.3 Matching Histograms

Akin to finding moments, but an intuitively better summarization of a fracture structure, is the fracture length histogram. This is a loose summary of the order statistics of fracture lengths in a pattern. Let h be a function that finds the absolute difference in fracture length frequencies in classes of fixed size. Let $\mathbf{A}$ and $\mathbf{B}$ denote vectors of length c. Each entry, A_i and B_i of $\mathbf{A}$ and $\mathbf{B}$, count the number of fractures in class i from a fixed fracture pattern, $\mathbf{X}_0$, and from a simulated fracture pattern, $g(\omega) = \mathbf{X}$, respectively. Let m be a function such that $m(\mathbf{X}_0) = \mathbf{A}$ and $m(g(\omega)) = \mathbf{B}$. Let c represent the number of frequency classes that we choose to define the histograms summarized by $\mathbf{A}$ and $\mathbf{B}$. Define h as

$$h(\omega) = \sum_{i=1}^{c} |A_i - B_i|. \tag{3.5}$$

Here we control the resolution of a match by our choice of c. An alternate definition of h sums the squared deviations between corresponding entries of $\mathbf{A}$ and $\mathbf{B}$. This definition has the advantage that it exagarates the difference between nonmatching patterns.

Defining m to summarize the histogram of fracture lengths in a pattern of fractures is the best choice for solving the problem from Chapter 2. This is because $\mathbf{X}_0$ (visually denoted by Figure 1) has censored fractures and because such an m does not incorporate the vertical position of any fracture in its value.

3.2 Censored Cracks and Boundaries

Conditional coding requires the simulation of many fracture patterns to work. In particular there must be a first simulation. All simulations result when the fgp operates on a list ω of random variables uniform on [0,1]. This list has two parts. The first part codes

ngen, *lmax*, and *gwsz* and a pattern of starter cracks. The second part is a long list of uniform random variables for use in the growth decision processes of simulation. Obviously *ngen*, *lmax*, and *gwsz* need not be bounded on [0,1]. At the very least, *ngen* and the x and y coordinates of the starter cracks need broader bounds. This requirement dictates that *ngen*, *lmax*, *gwsz*, and the starter crack coordinates are coded as linear functions of random variables uniform on [0,1]. This requires the introduction of parameters ancillary to fracture growth in the fgp. These parameters allow us to control the sampling region, S, for the growth parameters, and the growth region, A, for the starter cracks. Also, coding *ngen*, *lmax*, and *gwsz* as linear functions of random variables uniform on [0,1] is the same as giving them a uniform prior distribution. Assuming a uniform prior, conditional coding is taking a sample from the likelihood

$$\mathrm{f}_{\theta \mid M} = \frac{f_{M \mid \theta} f_{\theta}}{f_M} = c f_{M \mid \theta} I_S(\theta). \tag{3.6}$$

The capability to control the growth region is especially important in the case that $\mathbf{X}_0$ displays fractures censored by the growth boundary. If the natural pattern has cracks that intersect the growth boundary, then the beginnings of our simulated patterns cannot be restricted to the observed area of the natural pattern. Instead, the growth area must be extended so that starter cracks can grow from the outside in, since that might be what happened to form the natural pattern.

Deciding how far outside of the observed growth area to allow the placement of starter cracks requires some study of the pattern that we are trying to match. A reasonable approach is to keep all starter cracks within 3/4 the length of the longest fracture in the natural pattern away from the observable boundary. According to the fracture growth algorithm given in Chapter 2, all fracture growth is symmetric with respect to the mid-

point of the starter crack. This motivates us to allow the number of out-of-bounds starter cracks to match but not exceed the number of censored fractures in the natural pattern.

To perform conditional coding in the case of censored fractures, count the number of censored fractures in $\mathbf{X}_0$ and call the number z. Require that z starter cracks be placed randomly about the growth boundary each time a new fracture pattern is simulated for comparison to $\mathbf{X}_0$. We mean that up to z of the starter cracks will be allowed to take up positions outside the boundaries of observable growth region and through simulation have the opportunity to grow, possibly extending into the observable region where that portion of the fracture contributes to the simulated pattern.

Also relevant to this discussion is the topic of control over the sampling region. Conditional coding is more efficient if we sample *ngen*, *lmax*, and *gwsz* from a restricted space. For example, it is not wise to allow large *ngen* and *gwsz* when trying to match a pattern of many small fractures that are approximately the same size. If a fracture pattern has many small cracks and a few relatively large cracks, then a careful study of the fracture growth algorithm in Chapter 2 reveals that *lmax* is probably larger than l_0. In all cases, bounds on the allowable values of *ngen*, *lmax*, and *gwsz* need to be set or conditional coding may be slower than necessary.

3.3 Perturbing

During conditional coding, perturbing ω in the Metropolis algorithm is a concern. To keep track of what components in ω get coded to the growth parameters and starter crack coordinates and which are used as as probabilities, we impose an ordering in ω. Let elements 1-3 map to *ngen*, *lmax*, and *gwsz*, respectively. Let elements 4 through $3 + 2N$

map to the left end point of each starter crack. The even indices between 4 and $3 + 2N$ represent the x coordinates of the end points. The odd indices represent the y coordinates of the end points. Because all starter cracks enjoy a horizontal orientation, the right end points have the same y coordinate as the left endpoint. The x coordinates of the right end points of the starter cracks are just the left x coordinate plus l_0. All other elements in the list ω are uniform probabilities.

The length of ω must be sufficiently long so that all possible growth decisions during simulation have a corresponding entry in ω. To run the fgp, this means that ω must have at least $2BN + 2N + 3$ entries. Here N denotes the number of fractures in the natural pattern and B is the upper bound on the sampling interval for *ngen*. During simulation the most dramatic changes in fracture patterns from one simulation to the next occur when one of the three parameters *ngen*, *lmax*, or *gwsz* changes. Less dramatic effects occur when a ω is altered beyond the first three entries.

The idea behind conditional coding is to iterate through simulated fracture patterns such that the energy function is minimized for a particular pattern generated by a vector, ω. The ω vector evolves from perturbing in the Metropolis algorithm. There are many perturbing schemes available, but we require every acceptable scheme to perturb ω in such a way that 1.9 holds.

3.4 Monitoring Convergence

No theoretical signposts exist that prove convergence for this application of the Metropolis algorithm. However, the results of conditional coding are valid only under the assumption of convergence. We cannot observe convergence in MCMC methods directly but

we suppose that convergence has observable side effects.

One way to monitor convergence in MCMC methods requires the comparison of long disjoint strings of consecutive energy states in a nonparametric test of equal distribution. Since MCMC methods search for random lists of variables that have minimal energy configurations, it is reasonable to believe that as the Markov process internal to the MCMC method converges, the correlation between energy states that are fixed distances apart also converge to a fixed number. It is easy to keep track of such moving correlations and plot them as we go to check for convergence in the Metropolis algorithm.

These checks do not offer indisputable proof of convergence. However, they do offer some reassurance that we are moving toward viable solutions through whatever MCMC method we employ to do the conditional coding. As a postscript, note that these checks require a fixed method of perturbation for all variable lists that are compared to one another. These checks also require a fixed temperature.

3.5 The Conditional Coding Recipe

To conclude this chapter we offer a sequential view of the events necessary to yield the data in Chapter 5. First, start with a fracture pattern $\mathbf{X}_0$ of N fractures, z of which are censored. Consider $\mathbf{X}_0$ the natural fracture pattern. We are trying to find parameter points of the form $(ngen,lmax,gwsz)$ that produce populations of fracture patterns that overlap $\mathbf{X}_0$.

Choose an appropriate energy function. Randomly generate a vector ω_0 of random numbers uniform on [0,1] of sufficient length. The fgp modified for use with conditional coding interprets the first three entries of ω_0 as $ngen$, $lmax$, and $gwsz$, respectively. For

any vector $\omega = \omega_1, \omega_2, \omega_3, ...$, the fgp finds *ngen* as follows.

$$ngen = [\omega_1(b-a) + a] \tag{3.7}$$

Here a and b represent the upper and lower bounds on the sampling interval for *ngen*. The square brackets denote that *ngen* must be an integer. The variable ω_1 is the first entry in ω. The parameters *lmax* and *gwsz* are coded similar to *ngen*, using ω_2 and ω_3, but are not required to be integers.

The next $2N$ entries in any ω denote starter crack endpoints. Sequentially, every even element in ω from entry 4 to $(2N + 2)$ maps to an x-coordinate. Every odd element between the number 4 and $(2N + 3)$ entries in ω denotes y-coordinates that belong with the x-coordinate from the previous entry. If the growth region is rectangular, then the modified fgp reads the x and y coordinates according to the following equations.

$$x = (b-a)u_{2i} + a \tag{3.8}$$

$$y = (d-c)u_{2i+1} + c \tag{3.9}$$

Here i is an index from $2...[N/2]$. The variables a, b, c, d are upper and lower bounds for the growth rectangle. Equations 3.8 and 3.9 code the right endpoint coordinates of the starter cracks. The left endpoints have the same y-coordinates as the right and an x-coordinate that is l_0 less than the x-coordinate of the right endpoint. Let l_0 be fixed at a value of 0.01.

If $z > 0$, then the process of coding the starter cracks must be modified to accommodate the placement of up to z starter cracks outside the boundaries defined by the variables a, b, c, and d given in the previous paragraph. One way to perform this task is to force z of the N starter cracks to be placed within 3/4 the length of the longest fracture in $\mathbf{X}_0$ of

the growth boundary. Allow placement to occur with equal probability on either side of the boundary.

With h and ω_0 defined, apply the Metropolis algorithm described in Chapter 1. Each iteration of the Metropolis algorithm accepts or rejects an ω perturbed from the previous iteration. We perturb the vector ω by randomly assigning new values to a proportion of the elements in ω. For reasons that we explain later, run the Metropolis algorithm without annealing.

Run the Metropolis algorithm for a long number of iterations, and until the energy function is at a minimum. Generally, a large number of iterations is necessary for convergence in distribution of the Markov chain generated in the Metropolis algorithm. At this point, record the first three entries from the ω that minimized h. Perturb ω and continue until a sample of parameter points $(ngen,lmax,gwsz)$ of suitable size is acquired.

We seek a simple random sample of the parameter points $(ngen,lmax,gwsz)$ that produce patterns matching $\mathbf{X}_0$ in the sense that $m(g(\omega)) = m(\mathbf{X}_0)$. For the sample points to be independent, we must start the Metropolis algorithm over after each sample is taken or we must perturb the last ω that minimized h many times before accepting one of those perturbations into our sample.

Conditional coding is difficult to implement on the problem from Chapter 2. The results in Chapter 5 are enough to show the potential of conditional coding. The rest of our work details the successes and difficulties of conditional coding as it is applied here.

CHAPTER 4

CONDITIONAL CODING APPLIED

Appendix A.4 contains a table of 100 sample points (*ngen*,*lmax*,*gwsz*) obtained using conditional coding and Figure 1 as $\mathbf{X}_0$. To understand the sample, we start with a description of the energy function, perturbation scheme, sampling region, and temperature setting used to get it. This is followed by a discussion of time constraints on the experiment and the role they played causing adjustments to the procedure. Finally, we discuss issues of convergence in the Metropolis algorithm by investigating correlated energies and a non-parametric test of equal distribution between disjoint pieces of the Markov chain that results from the Metropolis algorithm. Chapter 5 offers an analysis of the data.

4.1 The Energy Function

$\mathbf{X}_0$ is the pattern of 200 fractures represented in Figure 1. Six of those fractures run out of bounds. It is easy to create an 8-class histogram such that the number of fracture lengths in each class is equal. Censored fractures are included. Call the vector of frequencies $\mathbf{A}$. Let m be a function defined so that $m(\mathbf{X}_0) = \mathbf{A}$. For any simulated pattern, $\mathbf{X}$, we can count the frequency of fractures with lengths in each frequency class used to define $\mathbf{A}$. We store these frequencies in a vector $\mathbf{B}$. We say that $m(g(\omega)) = \mathbf{B}$. The energy function, h, sums the absolute difference between corresponding entries in $\mathbf{A}$ and $\mathbf{B}$ according to the equation below.

$$h(\omega) = \begin{cases} \sum_{i=1}^{c+1} |A_i - Bi| & \text{if } \sum_{i=1}^{c+1} |A_i - Bi| > d \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

To keep track of censored fractures we include the absolute difference between the number of censored fractures from patterns $\mathbf{X}_0$ and $\mathbf{X}$. The constant c denotes the number of classes that define $\mathbf{A}$. The $(c+1)$st entries in $\mathbf{A}$ and $\mathbf{B}$ are the number of censored fractures in $\mathbf{X}_0$ and $\mathbf{X}$, respectively. The flag d is a positive whole number. In a sense, d controls the resolution of a match without changing m. The sample we obtained is the result of 4.1 with $d = 10$. Chapter 5 seeks to justify the introduction of d to h in the context of the current problem. It seems intuitive that for d large, h loses meaning, but that d small can reduce the time required to sample ω using h. Time constraints on the process forced the introduction of d.

4.2 The Perturbation Scheme

Settling on a perturbation scheme is a trial-and-error process. We found that it was advantageous to perturb the first three entries in ω about half of the time. The other half of the perturbing occurs at random on about 25% of the remaining entries in ω at a time. This perturbation scheme does not take into account the energy associated with ω. The perturbing we did was the result of experiments to see what worked best. Perturbing this way the Metropolis algorithm required 28,241 as a median number of steps to minimize the energy function with 42,960 steps as an average.

4.3 The Sampling Region

Experience shows that the smaller the sampling region for *ngen*, *lmax*, and *gwsz*, the less time it takes conditional coding to produce a sample. Recall that Figure 1 denoting $\mathbf{X}_0$ is actually the result of a single run of the fgp on a series of 200 starter cracks placed at random in a 50 by 50 box. The parameters that produced $\mathbf{X}_0$ are known. We know

that $ngen_0 = 100$, $lmax_0 = 0.01$, and $gwsz_0 = 0.1$. The experiment detailed in this chapter is designed to see if conditional coding produces a sensible sample of parameters capable of simulating $\mathbf{X}_0$. This motivated us to use conditional coding with a sampling region that included ($ngen_0$,$lmax_0$,$gwsz_0$). We allowed ω to code values of *ngen* between 85 and 105, values of *lmax* between 0 and 0.03, and values of *gwsz* between 0 and 0.3. This relatively restrictive sampling region was necessary to produce results in a tolerable period of time during the research process. There is nothing special about the way the sampling space is restricted except that *ngen*, *lmax*, and *gwsz* are bounded to scale and the point ($ngen_0$,$lmax_0$,$gwsz_0$) is not at the center of the parallelepiped that defines the sampling region.

4.4 Temperature

We can choose to perform conditional coding with a fixed temperature in the Metropolis algorithm or by starting with a high temperature and cooling it slowly. The cooling process with the Metropolis algorithm is called annealing. Authors Bertsimas and Tsitsiklis (1993) show that when annealing is performed, the Metropolis algorithm generates Markov chains that converge to a stationary distribution if and only if the following hold.

1. The cooling schedule is slow enough but converges to zero. One common schedule is

$$T(t) = \frac{d}{log(t)}, \tag{4.2}$$

   where t is a time or counting parameter and d is a constant.

2. The parameter d is sufficiently large. Bertsimas and Tsitsiklis (1993) show that for this cooling schedule, d must be at least as large as the smallest difference between the energy of any vector, ω, that minimizes h and the energy of any vector, ω, that

does not.

By our choice of h we know that the largest observable h on a pattern of 200 fractures divided into eight frequency classes given $\mathbf{X}_0$ is 550. If we let $d = 550$, then annealing in the Metropolis algorithm theoretically works. In practice, Bertsimas and Tsitsiklis (1993) point out that annealing may not reduce the number of iterations in the Metropolis algorithm necessary to get a solution. Early experiments seemed to show this to be the case for our sample.

Our sample was generated at a fixed temperature of 19. Annealing with the cooling schedule given by 4.2, with $d = 550$, takes one million iterations of the Metropolis algorithm before the temperature gets cooled to 39.8, and on the order of 10^{12} iterations to get down to 19. There are other, faster, cooling schedules that we tried including a geometric schedule, but they were no better than running at a fixed temperature of 19. Trial and error indicated that temperatures much higher than 19 delay results because the Metropolis algorithm easily rejects ω vectors with small energies in favor of ω vectors with larger energies.

When T is fixed small, the probability of escaping parameter configurations of locally minimum energy is small. This can result in a sample of points at certain minima while excluding points at some other minima. It is possible that for T not large enough, some parameter combinations with the potential of minimal energy fail to communicate in an appreciable way with other parameter combinations of equal or more potential for minimal energy.

We say that two parameter combinations communicate if there exists some positive probability of movement from one to the other in a finite number of steps through the

Markov chain generated by the Metropolis algorithm that describes the respective energies associated with each parameter combination. Two parameter combinations communicate in an appreciable way if correspondence between the two can be observed with relatively high probability after a large number of iterations in the Metropolis algorithm.

Since the starting state for the Markov chain generated with the Metropolis algorithm is chosen at random, it is possible that certain minima are not sampled with conditional coding, not because they are not likely, but because they are not likely given the starting state. High temperatures slow the sampling process, but avoid this difficulty by allowing the low energy states in Markov chains generated by the Metropolis algorithm to communicate easily with states of high energy so that no subset of the sampling region is nonaccessible to conditional coding. We do not attempt here to prove that with $T = 19$ this is true for conditional coding applied to the problem in Chapter 2. For this problem when $T = 19$ there was always a 5% chance of moving from one parameter combination to another that demonstrates an energy measure 56 units larger than the previous and jumps of smaller magnitude were increasingly more likely.

4.5 Time Constraints

The data set in Appendix A.4 came after gaining some months of experience with the fgp and the Metropolis algorithm. Many smaller data sets went before. What one learns from this process is that conditional coding can be exceedingly slow. Weeks were required to see what effect the latest adjustments in perturbing, temperature, sampling region and choice of energy function have on the process of conditional coding. Months of this kind of work led to the perturbing scheme, sample region, temperature, and minimum energy

acceptance region that produced the data. This sample via conditional coding took more than 1000 hours to generate on a SPARC 10 work station. This exorbitant amount of time results from the high number of iterations required by the Metropolis algorithm to minimize h compounded by the amount of time required by the fgp to simulate 200 crack fracture patterns.

4.6 Convergence

In spite of the large number of iterations made by the Metropolis algorithm to find configurations of ω that simulate fracture patterns of minimal energy, we cannot assume convergence. In fact, it is a discouraging truth that we can say little about the reality of convergence for this problem. We tried monitoring covariance, looking at histograms, and running nonparametric tests of equal distribution between disjoint pieces of the Markov chain that came out of the conditional coding. The results follow.

Figure 4 is a plot of the covariance among all the elements of the Markov chain generated by the Metropolis algorithm when conditional coding was applied to the inverse image problem of Chapter 2. The y-axis measures the covariance and the x-axis measures the distance between the states for which the covariances were computed. Ideally, as the Markov chain gets long enough that the resulting states are governed by a stationary distribution, the covariances should converge in a decreasing fashion to a constant of relatively low magnitude. Figure 4 demonstrates this behavior somewhat. However, there appears to be a great deal of correlation between states in the chain even when they are a long distance from one another. The greatest proportion of damping in covariance is complete by the time the Markov chain is 30,000 states long.

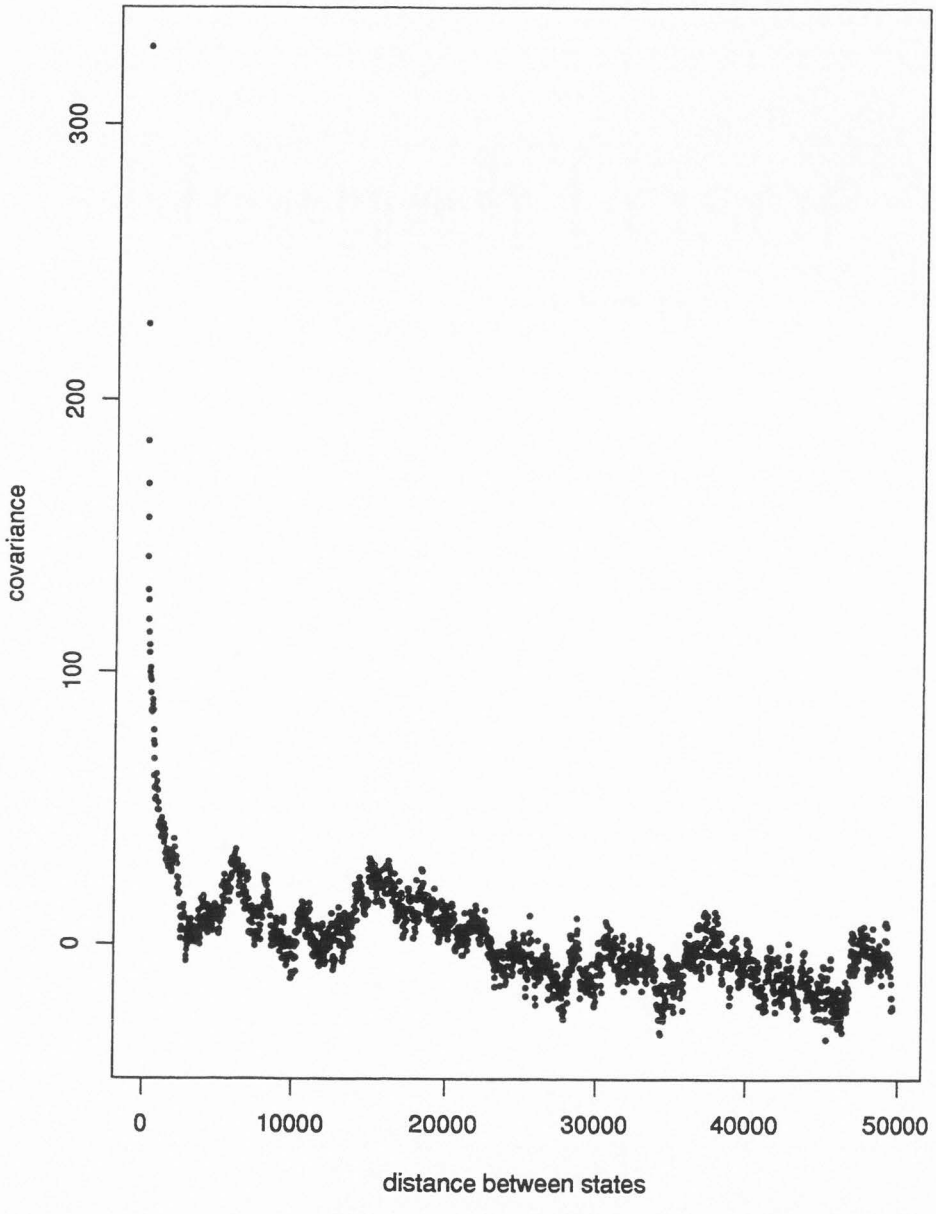

Figure 4. Covariance plot of moving correlations.

Figure 5 pictures a sequence of histograms summarizing disjoint pieces of the Markov chain. The first histogram covers the successive states from 30,000 to 130,000 and the second histogram covers the states from 150,000 to 250,000. The histograms have the same general shape and range but differ too much for such large sample sizes to conclude that they represent the same distribution. An application of the Kolmogorov-Smirnov nonparametric test of equal distribution seems to provide some evidence that the two histograms are from differing populations. With such large sample sizes, Conover (1980) approximates the critical value of the test to be 0.0060716 at the $\alpha = .05$ level. The test statistic computed with the FORTRAN code in appendix A.3 was 0.0250897. Because the test statistic is larger than the critical value, we reject the null hypothesis that the data summarized by the two histograms in Figure 5 are from the same population. We present these results noting that the Kolmogorov-Smirnov test assumes independent, identically distributed data points. Figure 4 demonstrates the violation of this assumption, making our test statistic less meaningful. These failures motivated another experiment.

After the collection of the 50th sample point the process of conditional coding had iterated the Metropolis algorithm 2,218,018 times. If sometime between the first sample point and sample point number 50 the Markov chain had converged in distribution, then there might be some fundamental difference in the points drawn at the first of the sample and those drawn at the end. Splitting the sample in half, we see little difference between the two sets of 50 points. Figure 6 is an all-pairs plot showing the distribution of the first 50 against the second 50 data points in the sample. Figure 7 is an all-pairs plot of the first 20 against the last 20 data points in the sample. In each set the range of the respective parameters is essentially the same. The means and medians are essentially the same, and

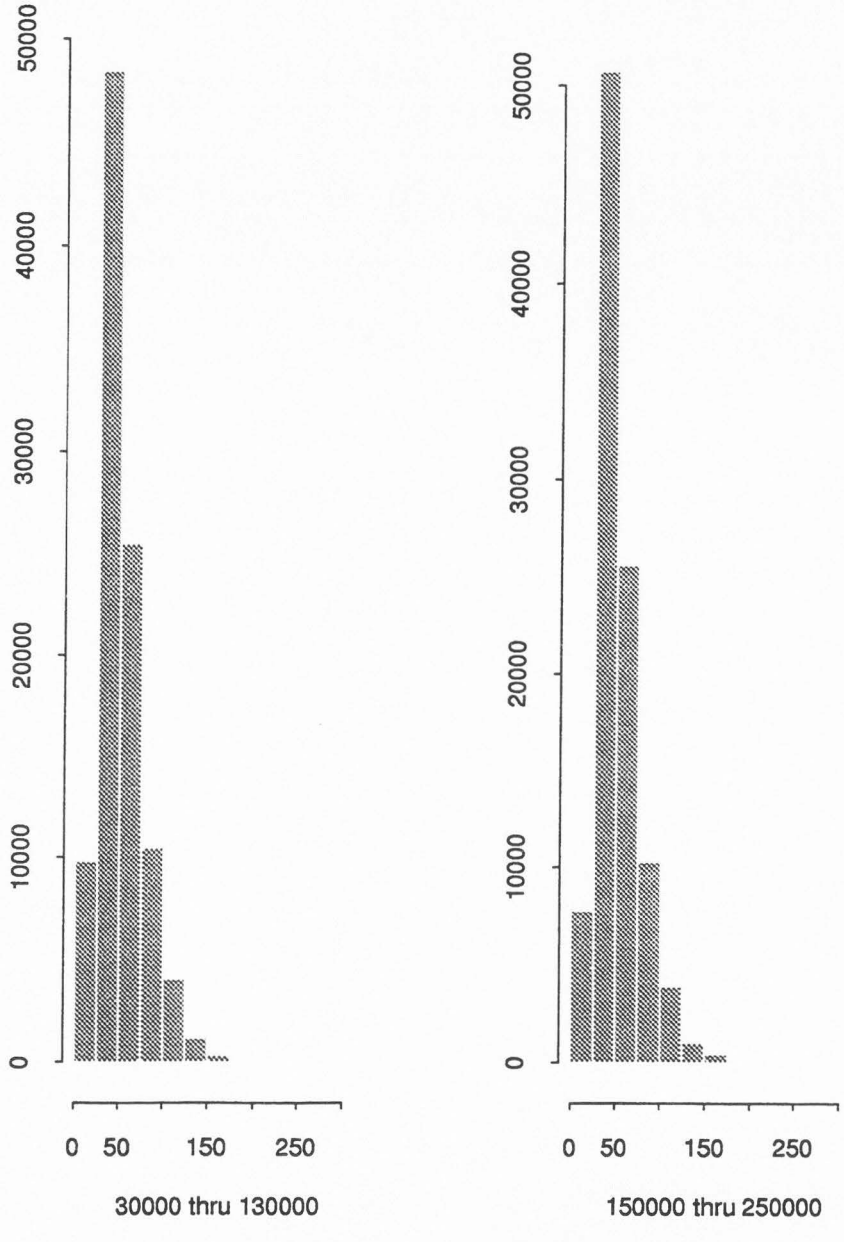

Figure 5. Frequency counts of energies as states in the Markov chain generated by the Metropolis algorithm. We sampled from a single long chain. This figure compares the distribution of energies observed in two disjoint pieces of the chain.

the interaction between the parameters themselves is indistinguishably the same.

The issue of convergence emerges as the most serious threat to the validity of our sample. Even if the histograms of Figure 5 were the same, there could be no reason to conclude that convergence occurred. Rosenthal (1994) supports the conclusion that convergence may be so slow that differences between long disjoint pieces of the Markov chain are statistically indiscernible and yet model poorly the true stationary distribution. Taken all together there is more evidence to support a denial of convergence than to verify convergence. Only the characteristics of the sample analyzed in the next chapter encourage the conclusion that the iterates have converged.

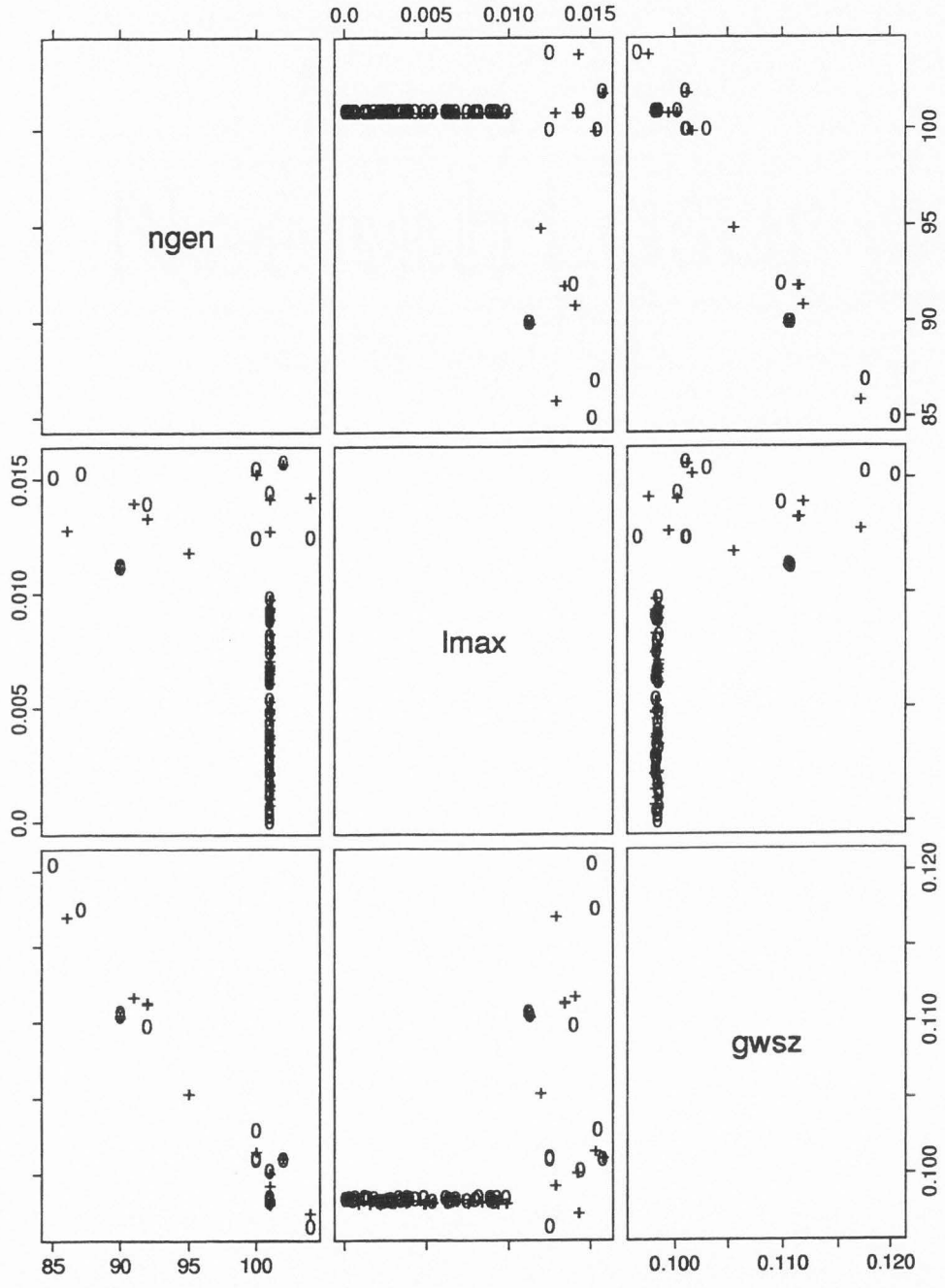

Figure 6. Pairs plot of 100 sample points. The first 50 points in the sample are denoted with "+" and the second 50 are denoted with "0".

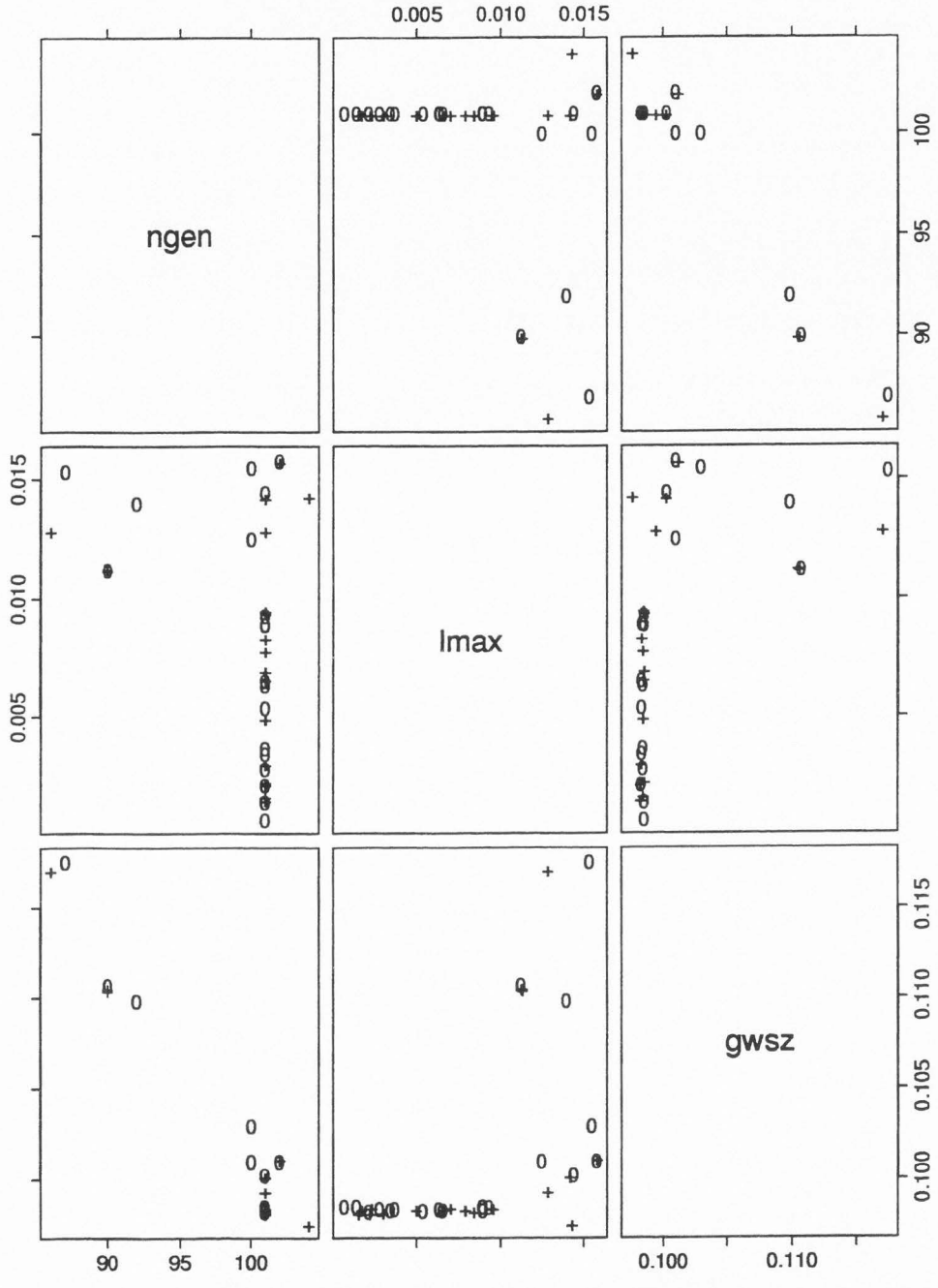

Figure 7. Pairs plot of 40 sample points consisting of the first 20 sample points and the last 20 sample points from an original sample of size equal to 100. The first 20 points in the sample are denoted with "+" and the second 20 are denoted with "0".

CHAPTER 5

DATA ANALYSIS

In this chapter we summarize the relationships between $ngen$, $lmax$, and $gwsz$ in the sample that we obtained with conditional coding. From our summary we propose the most likely estimates of $ngen_0$, $lmax_0$, and $qwsz_0$ corresponding to $\mathbf{X}_0$, based on the empirical distribution of our sample under the assumption that $ngen_0$, $lmax_0$, and $gwsz_0$ are in the sampling region given in Chapter 4. We conclude this chapter with some arguments in defense of the validity of the sample.

Appendix A.4 presents the sample of 100 points we obtained by conditional coding under the provisions explained in Chapter 4. Figure 8 is an all-pairs plot of the data points from the sample. The data appear in two groups. The first grouping of the data is for all points where $lmax \leq 0.01$. Recall that within the fgp each starter crack has an initial length of $l_0 = 0.01$. It makes sense that the relationship between $lmax$ and the other variables $ngen$ and $gwsz$ depends on l_0. For each run of the fgp where $lmax \leq l_0$, each fracture grows with probability one at every iteration. If $lmax > l_0$, then for each fracture in the pattern there is some positive probability that on certain iterations no growth will occur.

Figure 9 is a histogram of the $lmax$ values in the sample. The range of the sampled values of $lmax$ goes from 5.54681E-5 to 0.0157606. The sampling interval for $lmax$ was $[0, 0.03]$. Of the 100 values of $lmax$ collected, 74 of those observations occurred where $lmax \leq l_0$. Under the assumption that our sample is valid, this provides good evidence

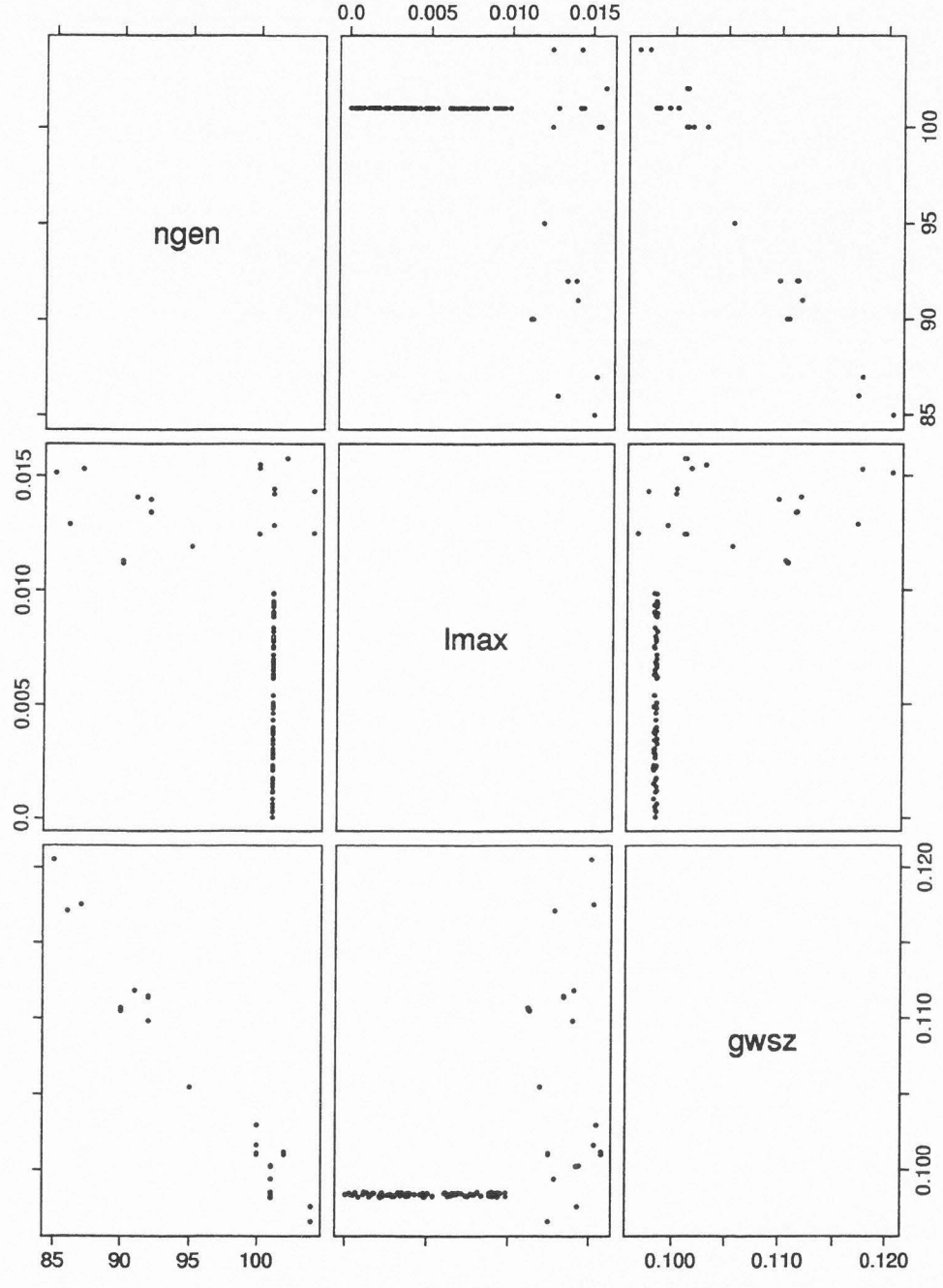

Figure 8. Pairwise plot of parameters sampled via conditional coding.

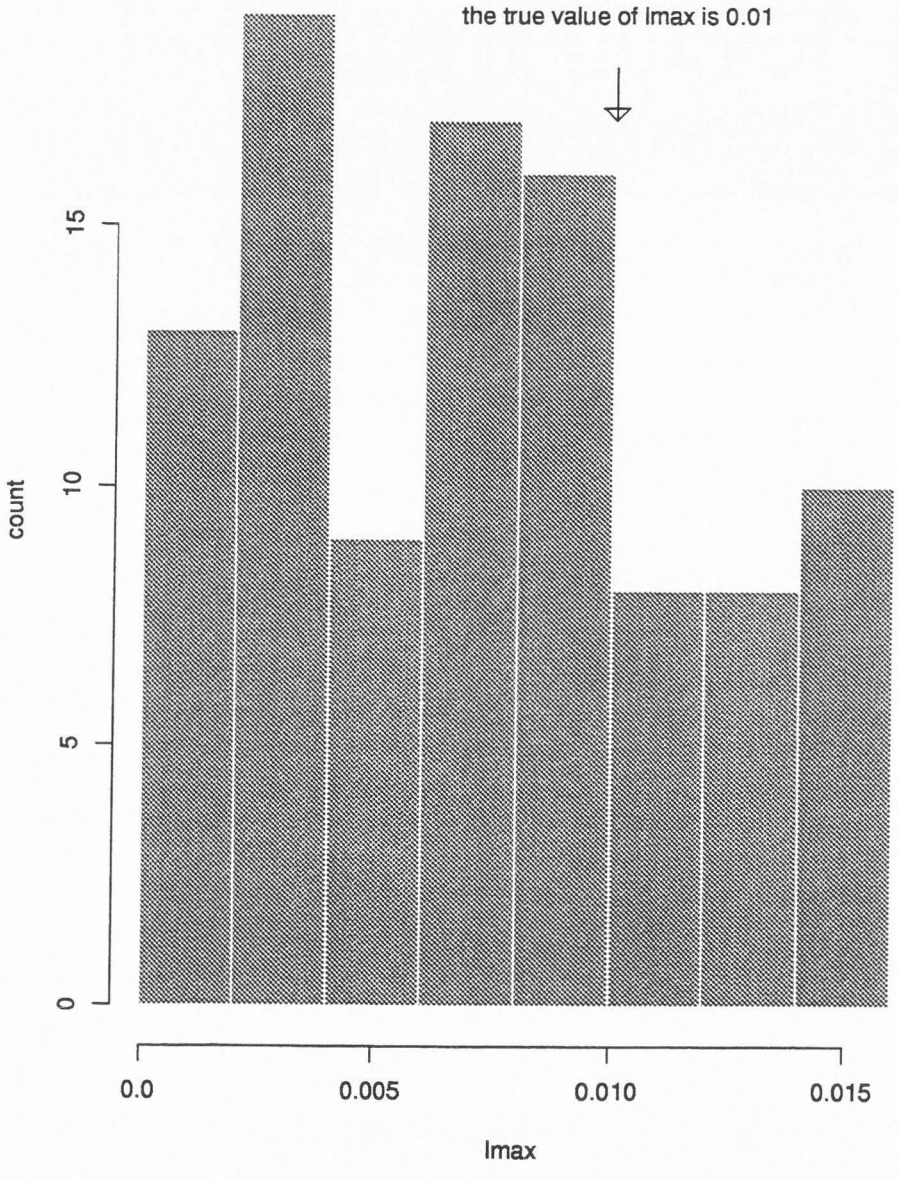

Figure 9. Histogram of sampled *lmax* values.

that $\mathbf{X}_0$ depicted in Figure 1 is more likely the result of the fgp run with $lmax \leq 0.01$ than with $lmax > 0.01$. Here it is good enough to say that we estimate $lmax_0$ is less than 0.01 because the fgp creates the same output for all values of $lmax$ less than l_0.

Figure 10 is a histogram of the *ngen* values in the sample. The range of the sampled *ngen* values is from 85 to 104. The sampling interval for *ngen* was $[85, 105]$. There are 77 observations where $ngen = 101$. Of these 77, 74 belong to sample points where $lmax \leq 0.01$. Refering back to Figure 8, we note that *ngen* is highly correlated with *gwsz*. The coefficient of correlation is -0.9991569. Due to the magnitude of this coefficient of correlation, we expect to see a high mode in the histogram of *gwsz* similar to the mode in Figure 10.

Figure 11 is a histogram of the *gwsz* values in the sample. The mode in this histogram coincides with the mode in Figure 10. The range of the sampled *gwsz* values is from 0.0966182 to 0.120508. The sampling interval for *gwsz* was $[0, 0.3]$. Of the 100 points in the sample, 71 of them had a *gwsz* component from the interval $[0.0982087, 0.985334]$. This interval has a width that is 1.35% of the range of the sampled values. All 71 of these observations belong to data points where $ngen = 101$ and $lmax \leq 0.01$.

Assuming convergence in the Markov chain generated in the Metropolis algorithm, there is strong evidence that $ngen = 100$, $lmax = 0.01$, and $gwsz = 0.1$ are not the parameters most likely to produce $\mathbf{X}_0$ in simulation through the fgp. Likewise, our sample does not undermine its own validity by excluding the true parameters from those reasonable with some empirically appreciable probability.

We introduced the parameter d into the energy function in Chapter 4. The introduction of d was necessary to speed the process of sampling with conditional coding. We admit

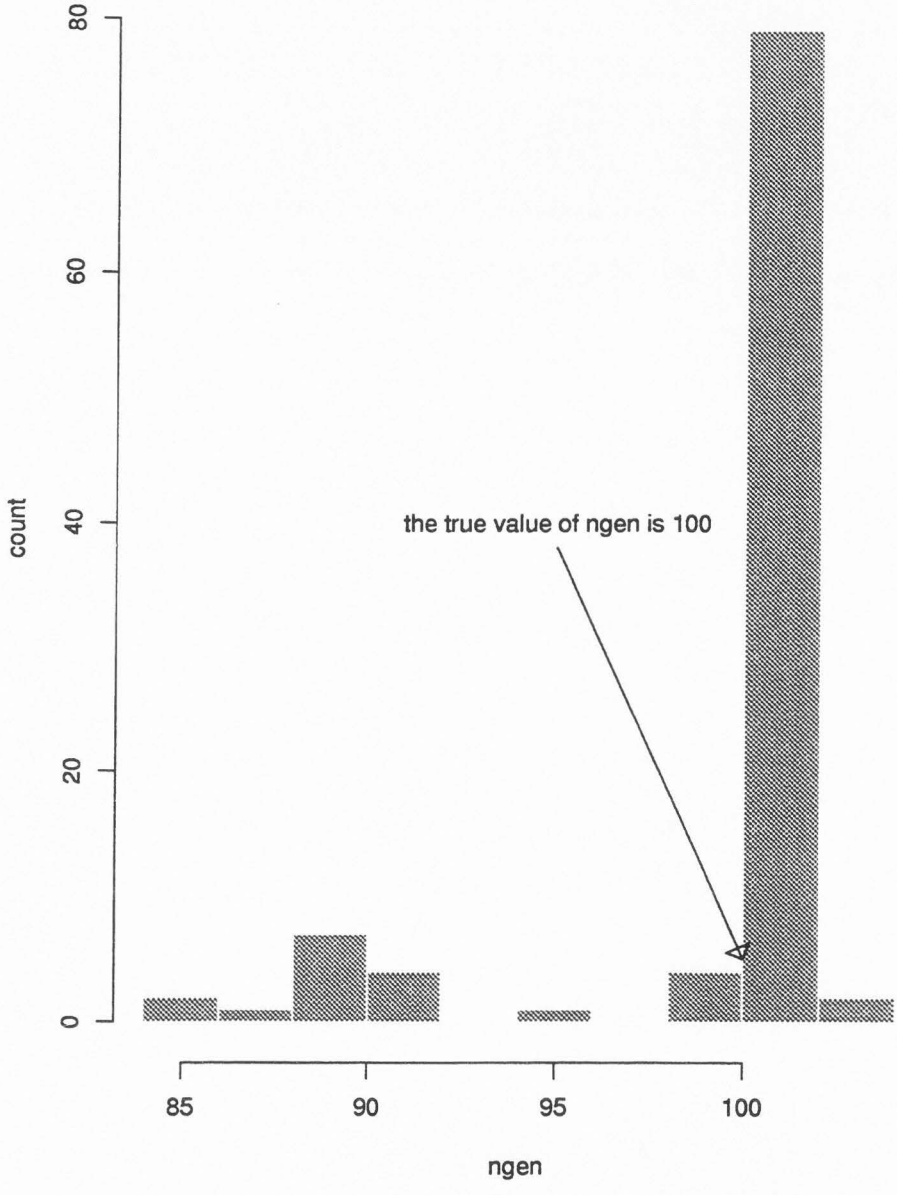

Figure 10. Histogram of sampled *ngen* values.

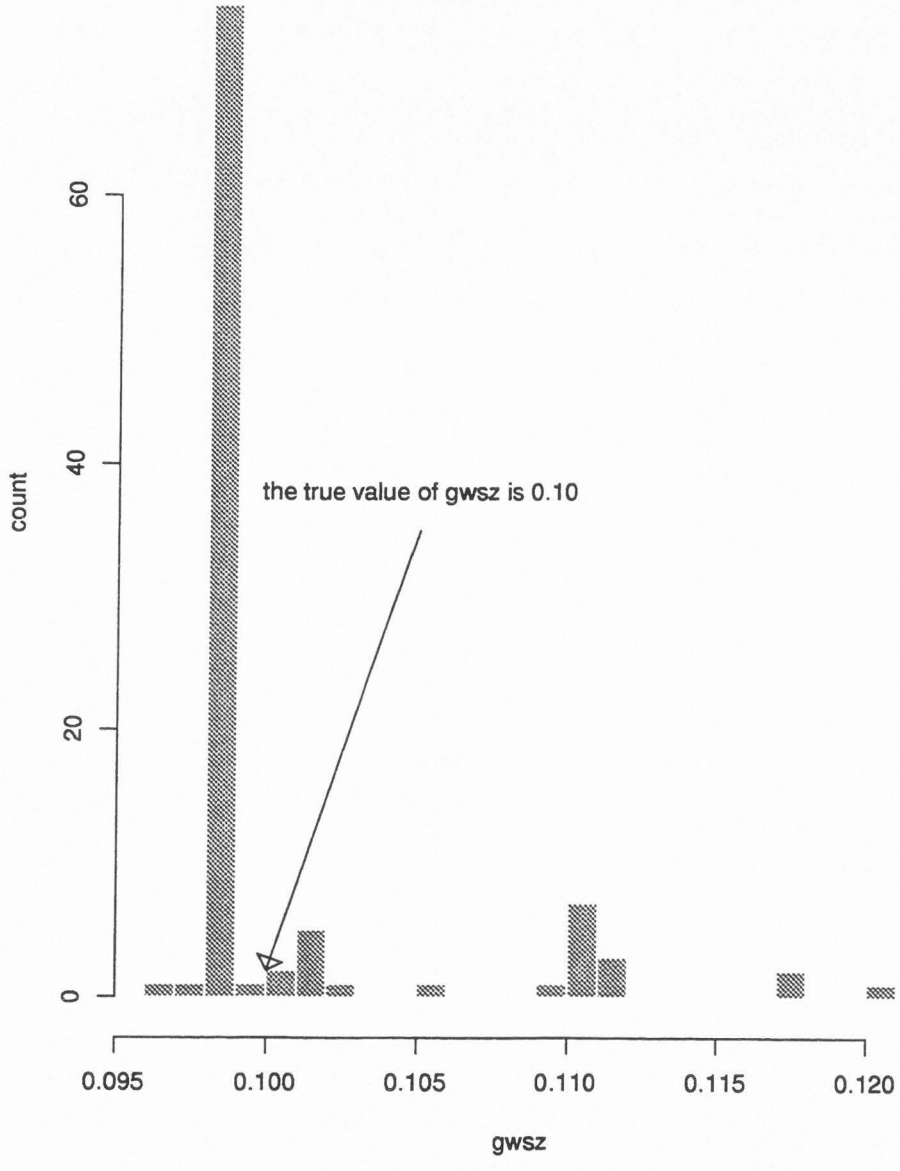

Figure 11. Histogram of sampled *gwsz* values.

that by introducing d into the energy function, h, we run the risk of distorting the meaning of h so that any sample we obtain using h is uninterpretable. By letting $d = 10$ in 4.1, we did not, in a statistical sense, distort what it means for h to be 0 and hence what it means for a simulated pattern $\mathbf{X}$ to match $\mathbf{X}_0$. We justify this with χ^2 goodness-of-fit tests.

For each ω sampled via the Metropolis algorithm, we recorded the vector $\mathbf{B}$ of fracture length frequency counts, including the number of censored fractures, from the fracture pattern $\mathbf{X}$ generated by $g(\omega)$, where $h(\omega) = 0$ in 4.1 and $d = 10$. We test individually the hypotheses that each vector $\mathbf{B}$ summarizes fractures from the population of fractures defined by $\mathbf{A}$. We defined $\mathbf{A}$ in section 1 of Chapter 4. Recall that the vector $\mathbf{A}$ has nine entries. The first eight record the frequency counts of fracture lengths occurring in $\mathbf{X}_0$. The ninth is the number of censored fractures in $\mathbf{X}_0$. Under the null hypothesis, the expected count for each entry in $\mathbf{B}$ is the corresponding entry in $\mathbf{A}$. In this case $\mathbf{A}$ is the vector (28,25,23,26,26,24,23,25,6). Computing the χ^2 statistic in the usual way, we find that all of the 8-degree-of-freedom test statistics for the vectors $\mathbf{B}$ have p-values in the interval (0.99,1.00). Thus, we fail to reject the null hypothesis for any of the vectors $\mathbf{B}$ that correspond to the sample points ω that we obtained by conditional coding. By letting $d = 10$ in 4.1, we still allowed h to determine a match among patterns that are statistically alike. Appendix A.5 contains a complete table of the χ^2 satistics just mentioned.

CHAPTER 6

SUMMARY

In the preceding chapters we have defined and applied conditional coding. The theory is general but the application is unique to the problem presented in Chapter 2. In spite of this, there are issues of a general nature that arise in the application of conditional coding to this problem. We conclude with a summary of conditional coding and a summary of our attempt to match parameters *ngen*, *lmax*, and *gwsz* to the fracture pattern of Figure 1 that we called $\mathbf{X}_0$. During the course of this summary we distinguish the successes and failures of applying conditional coding to the fracture pattern inverse image problem.

Conditional coding is a method that applies MCMC methods to sample from a conditional distribution. Classically, sampling such distributions with MCMC methods requires an explicit statement of the posterior likelihood of the variable we want to sample given some variable dependent measure that we know. This measure might include some error. Conditional coding requires no analytic formulations. It requires only computer-implemented algorithms that simulate the random variables and the measurements that are made on them.

We have applied conditional coding to the problem of finding the inverse image of a fracture pattern, $\mathbf{X}_0$, simulated with rules determined by an algorithm, g, and three parameters, *ngen*, *lmax*, and *gwsz*. The rules govern fracture growth determined by the elements of a stochastic process given as a list of random numbers from a standard random number generator. Since g relies on random numbers to determine $\mathbf{X}_0$, we suppose

that certain combinations of $ngen$, $lmax$, and $gwsz$ are more likely to simulate fracture patterns with fracture length frequency counts matching $\mathbf{X}_0$ than others. The sample that conditional coding produced by matching parameters to Figure 1 bears this out. From the sample it appears that parameter combinations with $ngen = 101$, $lmax \leq 0.01$ and $gwsz \in [0.0982087, 0.0985334]$ are more likely to simulate fracture patterns with fracture length histograms of 8 near equal probability classes that match the corresponding fracture length histogram of $\mathbf{X}_0$. This conclusion comes with some assumptions because the sample referred to above comes from the small sampling region given in Chapter 4. When interpreting the meaning of this sample, it is understood that it only details the relationship between $ngen$, $lmax$ and $gwsz$ within that region. There are other restrictions on the interpretation of the sample as well.

We ran the Metropolis algorithm as part of conditional coding with a fixed temperature of $T = 19$. Our sample is a sample over all parameter combinations in the sampling region that minimize our energy function if T is not too small. If T is too small, then the probability of escaping parameter configurations of locally minimum energy is small. This can result in a sample of points at certain minima while excluding points at some other minima. We never address whether or not using $T = 19$ is a problem in this regard. Energy states greater than 250, though less than 275, regularly occur throughout the Markov chain used to simulate the distribution from which we sample. With the energy function that we chose, the maximum observable energy for any parameter combination is 550. Our sample is biased unless every combination of parameters with the potential of simulating fracture patterns of minimum energy communicates with every other combination that has the same potential. This communication must not occur through combinations of

parameters that simulate fracture patterns with energies of more than 275. This might be a restrictive assumption and is one that we have not tested.

To avoid problems with temperature in future experiments, we recommend that the Metropolis algorithm be started over after sampling each point. The initial energy state of the new Markov chain generated for finding the next sample point should come from a combination of parameters chosen at random. The hope is that starting fresh at a random point after each sample will allow sampling of all parameter combinations capable of simulating fracture patterns of minimal energy. Another solution might be to formulate the energy function with a small number of states. The energy function should be very sensitive to change in patterns of minimal or near minimal energy and insensitive to broad differences in patterns that are far from matching anyway.

Convergence in MCMC methods is the most problematic component of conditional coding. This area more than any other requires future research and resolution before conditional coding can find widespread and reliable application. Our attempts to show convergence in the Markov chain that we used to generate our sample failed. In spite of this, the theory is sound and the sample we obtained is evidence that conditional coding is viable. A completely valid sample of all the parameter combinations capable of simulating patterns of minimal energy lacks only a better understanding of convergence in MCMC methods.

BIBLIOGRAPHY

Bertsimas, D., and Tsitisiklis, J. (1993), "Simulated Annealing," *Statistical Science 8(1),* 10-15.

Conover, W.J. (1980), *Practical Nonparametric Statistics,* New York: John Wiley & Sons Inc.

Geman, S., Geman, D.(1984), "Stochastic Relaxation, Gibbs Distribution and the Bayesian Restoration of Images," *IEEE Transactions on Pattern Analysis and Machine Intelligence 6,* 721-741.

Hollander, M., Wolfe, D.A. (1973), *Nonparametric Statistical Methods,* New York: John Wiley & Sons Inc.

Martel, S., Hestir, K., and Long, J.C.S. (1990), "Generation of Fracture Patterns Using Self-Similar Iterated Function Systems Concepts," *Earth Sciences Division Annual Report LBL-29700,* Berkeley, CA: Lawrence Berkeley Laboratory, University of California, pp. 52-56.

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., and Teller, A.H. (1953), "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics 26(6),* 1087-1092.

Rosenthal, J.S. (1994), "Theoretical Rates of Convergence for Markov Chain Monte Carlo," *Conference Proceedings, Interface '94.* (Also available via anonymous ftp at ustat.tronto.edu in the file */jeff/ augment.ps.Z.*)

Smith, A.F.M., and Roberts, G.O. (1993), "Bayesian Computation via the Gibbs Sampler and Related Markov Chain Monte Carlo Methods," *Journal of the Royal Statistical Society 55(1),* 3-23.

Spitzer, F. (1971), "Markov Random Fields and Gibbs Ensembles," *American Mathematical Monthly 78,* 142-154.

The New Lexington Webster's Encyclopedic Dictionary of the English Lanquage, (1990), New York: Lexington Publications Inc.

Vetterling, W.T. (1985), *Numerical Recipes in FORTRAN*, Cambridge, England: Cambridge University Press.

APPENDICES

APPENDIX A

PROGRAMS

A.1 Fracture Growth Program

A description of the full capacity of the fgp can be found in an article titled "Generation of Fracture Patterns Using Self-Similar Iterated Function System Concepts." The article appears in the June 1990 Annual Report LBL-2700 of the Lawrence Berkeley Laboratory at the University of California at Berkeley. This FORTRAN program codes the algorithm from that article. The algorithm is more sophisticated than the description given in Chapter 2. This program simulates Figure 1 and more complex patterns of fractures that are not all oriented in the same direction. This program also has the capacity to generate daughter fractures in close proximatey to the existing fractures that spawn them. The inputs include a list of parameters including *ngen*, *lmax* and *gwsz* as well as a parameter that lets the program spawn new fractures, and one that allows fractures to grow non-parallel to one another. The code given here is courtesy of Dr. Kevin Hestir of Utah State University.

```
c This is the 'header' of common declarations.
      integer npar,ngen


      real prob(10)
      real g(4,10)
```

```
      real x(4,10000),xp(4,10000)
      real lmax
      real gwsz
      integer n,np

      common /n/ npar,ngen
      common /p/ prob
      common /g/ g
      common /l/ lmax
      common /gr/ gwsz

      subroutine ftrans(xfrac,yfrac)
      real xfrac(4),yfrac(4)
      real theta
      real xl,xxm,xym
      data tol /0.00001/
c This subroutine rotates, scales, and translates
c a reference frame in which the parent (x) fracture has endpoints
c (-0.5,0), (+0.5,0) (the x2 frame) to a frame in which
c the parent has its actual endpoints (the x0 frame).
c The coordinates of a daughter (y) fracture are known in
c the x2 frame and are calculated for the x0 frame.
```

```
c Determine the orientation of the parent fracture in the
c x0 frame.  This angle is theta and is positive in the
c counterclockwise direction from the x-axis.
      theta=hatan(xfrac)
c Rotate the coordinate frame clockwise so that the parent
c fracture is in the correct orientation and calculate the
c orientation of the daughter in the x0 reference frame.
      call rotan(yfrac,theta)
c Scale the new reference frame and calculate the coordinates
c of the daughter fracture in the rescaled reference frame.
c In the new reference frame the parent fracture has its true
c length.
      xl=xlen(xfrac)
      do i=1,4
       yfrac(i)=xl*yfrac(i)
      enddo
c Translate the reference frame such that the
c the midpoint of the parent fracture is in the
c correct location.  Calculate the coordinates
c of the daughter fracture endpoints in the
c translated coordinate system.
      xxm=(xfrac(1)+xfrac(3))*0.5
      xym=(xfrac(2)+xfrac(4))*0.5
```

```
      do i=1,3,2
       yfrac(i)=yfrac(i)+xxm
       yfrac(i+1)=yfrac(i+1)+xym
      enddo
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine gen(x,n,xp,np)
      include 'header'
      integer i,j,iflag
      real y(4),z(4)
c This program places the endpoints for parent fractures and
c any daughter fractures into an array called xp.
      np=0
      do i=1,n
c This do loop operates on each of the n fractures
c Collect the endpoints y(1), y(2), y(3), y(4) for the
c ith fracture
       do j=1,4
         y(j)=x(j,i)
       enddo
c Determine whether and where a daughter fracture will be grown
c near the tip of the ith parent fracture
```

```
        call ifsgen(y,z,iflag)
c Place the parent fracture endpoints into the xp storage array
        np=np+1
        do j=1,4
          xp(j,np)=y(j)
        enddo
        if(iflag.gt.1) then
c A daughter fracture was grown near (but not at) the tip of the
c parent fracture.  Put the daughter fracture into the xp storage array.
        np=np+1
        do j=1,4
          xp(j,np)=z(j)
        enddo
        endif
      enddo
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      function hatan(y)
      real tol
      real hatan,y(4)
      real delx,dely
      data tol /0.00001/
```

```
c This function gives the orientation of a line
c with respect to the x-axis given the coordinates
c of its endpoints y(1), y(20, y(3), and y(4).
      delx=y(1)-y(3)
      dely=y(2)-y(4)
      if(abs(delx).lt.tol) then
       hatan=2.0*atan(1.)
      else
       hatan=atan(dely/delx)
      endif
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine ifsgen(y,z,iflag)
      include 'header'
      real y(4),z(4)
      real p
      integer iflag
c This subroutine decides whether to grow a new fracture near the
c tip of a pre-existing one and decides where to grow it.
c The points y(1), y(2), y(3), and y(4) mark fracture end points.
c The odd number indices are x coordinates.
c The even number indices are y coordinates.
```

```
c The parameter p is a random number between 0 and 1.
        pi = 4.*atan(1.)
c Select a random number p to determine whether the fracture
c will grow or nucleate a new fracture
c Fracture growth probabilities are scaled to
C a parameter called lmax.
        iflag=1
        p=ran1(idum)
C If p>xlen/lmax, then no fracture growth occurs,
C and the suboutine is exited.
        if(p.gt.(xlen(y)/lmax)) return
C If p<xlen/lmax, then fracture growth occurs.
C The idea is that the probability of fracture growth should
C be proportional to the fracture energy release rate G.
C G in turn, is proportional to K*K, where K is the stress
C intensity factor and is proportional to the square root of the
C crack length.  So p should be proportional to the crack length,
C which is normalized here by the parameter lmax.
C Now pick a random number to determine whether the parent
C crack will grow itself or spawn a daughter
C If p>prob(1) then a daughter crack will grow (go to 600)
C If p<prob(1) then the parent crack will grow
      p = ran1(idum)
```

```
      if (p.ge.prob(1)) goto 600
C ROUTINE FOR PARENT CRACK GROWTH
C In the following scheme, the old (short) and new (long)
C parent crack have the same midpoint.
C The maximum relative growth increment is b = gwsz
C Both crack endpoints move to increase the
C crack length by b*ran1 %.
      p = ran1(idum)
      b = gwsz * p
           xm = (y(1) + y(3)) *0.5
           ym = (y(2) + y(4)) *0.5
           y(1)=(b+1.0)*y(1) - b*xm
           y(2)=(b+1.0)*y(2) - b*ym
           y(3)=(b+1.0)*y(3) - b*xm
           y(4)=(b+1.0)*y(4) - b*ym
       return
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
  600 continue
C DAUGHTER CRACK GROWTH ROUTINE
       iflag = 2
C Growth will (will not) occur at both ends of the crack
C The parameter b is the distance over which the near tip field
C is assumed to be appropriate.
```

```
C The maximum growth increment is also b = gwsz
      b = gwsz
C Now pick two random numbers to give the coordinates r, theta
C for the center of the new daughter crack.
C These numbers will be used to locate the daughter based on
C probability distributions derived from the near tip expression
C for sigma yy.
C This coordinate system is centered at and aligned with
C the parent crack tip.
C First for r:
      p=ran1(idum)
      r=p*p*b*xlen(y)
C and now for theta
      p=ran1(idum)
C Solve for theta using Newton-Raphson method
      call rtnewt(p,theta,0.01,pi)
C Now determine the orientation of the parent
      ang = atan2((y(4)-y(2)),(y(3)-y(1)))
C Now determine which end of the parent to grow the daughter near
C d = 1 corresponds to positive end of parent
C d = -1 corresponds to positive end of parent
      d = 1.
      p = ran1(idum)
```

```
        if (p.gt.0.5) d = -1.
C Pick the appropriate parent crack endpoint coordinates
        if (d.gt.0.) then
        w1 = y(3)
        w2 = y(4)
        else
        w1 = y(1)
        w2 = y(2)
        end if
C Now pick the length of the daughter crack
        p = ran1(idum)
c fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix??
        astar = p*b*xlen(y)
c fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix?? fix??
C Now locate the daughter crack midpoint
        xm = w1+d*r*cos(ang+theta)
        ym = w2+d*r*sin(ang+theta)
C Now locate the daughter crack endpoints
        z(1) = xm-cos(ang)*astar
        z(2) = ym-sin(ang)*astar
        z(3) = xm+cos(ang)*astar
        z(4) = ym+sin(ang)*astar
      return
```

```
      end
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       subroutine rtnewt(p,root,tol,pi)
       parameter (jmax=20)
       x1 = -1.*pi
       x2 = pi
C Set initial guess
       root = 2.*(p-0.5)*pi
       do 11 j=1,jmax
       call funcd(p,root,f,df)
       dx=f/df
       root=root-dx
       if (abs(dx).lt.tol) return
   11  continue
       return
       end
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine funcd(p,root,f,df)
C function to calculate the integral of the normalized
C near-tip stress sigma 11 (f) and sigma 11 (df).
       phi = root/2.
       s1 = sin(phi)
       s3 = sin(phi)**3.
```

```
      s5 = sin(phi)**5.
      f = (2.*s1+2.*s3-1.6*s5+2.4)*(5./24.)-p
      df = cos(phi)*(1+s1*sin(3*phi))*(5./24.)
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Master program to recursively generate line fractures
c
c The following statement defines variables and common blocks
c used throughout the ifs fracture generation program:
      include 'header'
      character nameo*40
c Read the input data
      call rdpar
c     call normal(prob,npar)
c Read the initial fracture endpoints from the file start.inp
      call read4(x,n)
c
c Now enter the do loop that generates daughter fractures, filters
c out duplicate fractures, and restores the parent and daughter
c fractures back into storage array x.  The number of generations
c for which fractures can be grown is ngen.
      do i=1,ngen
```

```
        call gen(x,n,xp,np)
        call filter(xp,np)
        call swap(x,n,xp,np)
        write(6,*)i,n
       enddo
c
       write(6,307)
  307 format(//,' enter name of output file')
       read(5,90)nameo
   90 format(a)
       open(unit=10,file='markov.out',status='unknown')
       open(unit=11,file=nameo,status='unknown')
c Write out the endpoints of the final set of fractures
       do i=1,n
        write(10,*) x(1,i),x(2,i),x(3,i),x(4,i)
        write(11,*) x(1,i),x(2,i),x(3,i),x(4,i)
       enddo
       close(unit=10,status='keep')
       close(unit=11,status='keep')
       end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       subroutine normal(prob,npar)
       real prob(10),sum
```

```
      integer npar,i
c This subroutine converts relative growth rule probabilities
c to absolute probabilities whose sum is 1.
      sum=0.0
      do i=1,npar
      sum=sum+prob(i)
      enddo
      do i=1,npar
      prob(i)=prob(i)/sum
      enddo
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      FUNCTION RAN1(IDUM)
      DIMENSION R(97)
      PARAMETER (M1=259200,IA1=7141,IC1=54773,RM1=3.8580247E-6)
      PARAMETER (M2=134456,IA2=8121,IC2=28411,RM2=7.4373773E-6)
      PARAMETER (M3=243000,IA3=4561,IC3=51349)
      DATA IFF /0/
c Subroutine to generate a random number between 0 and 1
c From the Numerical recipes book.
      IF (IDUM.LT.0.OR.IFF.EQ.0) THEN
        IFF=1
```

```
          IX1=MOD(IC1-IDUM,M1)
          IX1=MOD(IA1*IX1+IC1,M1)
          IX2=MOD(IX1,M2)
          IX1=MOD(IA1*IX1+IC1,M1)
          IX3=MOD(IX1,M3)
          DO 11 J=1,97
            IX1=MOD(IA1*IX1+IC1,M1)
            IX2=MOD(IA2*IX2+IC2,M2)
            R(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
11        CONTINUE
          IDUM=1
        ENDIF
        IX1=MOD(IA1*IX1+IC1,M1)
        IX2=MOD(IA2*IX2+IC2,M2)
        IX3=MOD(IA3*IX3+IC3,M3)
        J=1+(97*IX3)/M3
        IF(J.GT.97.OR.J.LT.1)PAUSE
        RAN1=R(J)
        R(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
        RETURN
        END
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
        subroutine rdpar
```

```
      character bogus*1
      character name*40
      include 'header'
c This subroutine reads the growth parameters from the desired
c input file par.inp.
c The lines below that say "read (15,*)bogus" read the titles
c for the growth criteria.
c ngen = number of generations in which fracture growth is allowed
c idum = seed for random number generator
c npar = # of growth rules
c lmax = maximum allowable fracture length
c prob(i) = probability for growth rule i
c g(i) = location of daughter fracture endpoints relative
c        to a parent fracture at (-0.5,0), (0.5,0)
c gwsz = growth increment % for a parent fracture that lengthens
c        (as opposed to a fracture that grows a daughter)
c
      write(6,80)
   80 format(//,' enter name of input file for growth parameters')
      read(5,85)name
   85 format(a)
   90 format(a)
      open(unit=15,file=name,status='old')
```

```
      read(15,90)bogus
      read(15,*)ngen
      read(15,90)bogus
      read(15,*)idum
      read(15,90)bogus
      read(15,*)npar
      read(15,90)bogus
      read(15,*)lmax
      do i=1,npar-1
      read(15,90)bogus
      read(15,*)prob(i)
      read(15,*)g(1,i),g(2,i),g(3,i),g(4,i)
      enddo
      read(15,90)bogus
      read(15,*)prob(npar)
      read(15,*)gwsz
      close(unit=15,status='keep')
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine read4(x,n)
      include 'header'
c This subroutine reads the endpoint coordinates
```

```
c for the array of starter cracks and counts how
c many starter cracks there are.  The number is n.
      open(unit=10,file='start.inp',status='old')
      n=0
  100 continue
      n=n+1
      read(10,*,end=200)x(1,n),x(2,n),x(3,n),x(4,n)
      goto 100
  200 continue
      n=n-1
      close(unit=10,status='keep')
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine rotan(y,theta)
      real y(4),dum(4)
      real theta,cost,sint
c This subroutine rotates a line segment (as defined by its
c endpoints) by a counterclockwise angle theta and then gives the
c new endpoints.  The rotation center is the origin.
      do i=1,4
       dum(i)=y(i)
      enddo
```

```
      cost=cos(theta)
      sint=sin(theta)
      y(1)=cost*dum(1)-sint*dum(2)
      y(2)=sint*dum(1)+cost*dum(2)
      y(3)=cost*dum(3)-sint*dum(4)
      y(4)=sint*dum(3)+cost*dum(4)
      return
      end
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine swap(x,n,xp,np)
      include 'header'
c This subroutine sets the x array equal to the xp array
      do i=1,np
       do j=1,4
       x(j,i)=xp(j,i)
       enddo
      enddo
      n=np
      end
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      function xlen(y)
      real xlen,y(4)
c This function determines the length of a line segment
```

```
c given the coordinates of its endpoints.
      xlen=sqrt((y(1)-y(3))**2. + (y(2)-y(4))**2.)
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine filter(x,n)
      real x(4,10000)
      real xp(4,10000)
      integer idic(10000)
      integer n,np
      data tol /1.0e-08/
c This subroutine checks to make sure there are no
c duplicate fractures in the ouput produced by
c subroutine gen
c Copy the endpoints for each fracture from the permanent
c x array to a temporary xp array
     do i=1,n
      idic(i)=0
      do j=1,4
       xp(j,i)=x(j,i)
      enddo
     enddo
c March through the fracture array (checking each fracture
```

```
c against all the fractures that follow it in the array)
c to see whether duplicate sets of fracture endpoints occur.
c If there is a duplicate set, flag the fracture that is
c closer to the end of the array.
      do 100 i=1,n-1
       if(idic(i).gt.0) goto 100
       do 200 j=i+1,n
           if(abs(x(1,i)-x(1,j)).ge.tol) goto 200
           if(abs(x(2,i)-x(2,j)).ge.tol) goto 200
           if(abs(x(3,i)-x(3,j)).ge.tol) goto 200
           if(abs(x(4,i)-x(4,j)).ge.tol) goto 200
           idic(j)=1
  200   continue
  100 continue
c Write all unflagged (i.e. nonduplicate) sets of fracture
c endpoints back to the x array.
      np=0
      do i=1,n
      if(idic(i).lt.1) then
       np=np+1
       do j=1,4
        x(j,np)=xp(j,i)
       enddo
```

```
      endif
      enddo
c The number of fractures now in the x array is defined as n
      n=np
      return
      end
```

A.2 Modified Fracture Growth Program

This program is a modified version of the fgp that does conditional coding. Some of the routines from the original fgp are deleted here becuase they make no contribution to solving the problem from Chapter 2. There are also many new routines specific to the application of conditional coding.

```
c This is the 'header' of common declarations.
      integer npar,ngen

      real prob(10)
      real g(4,10)
      real x(4,10000),xp(4,10000)
      real lmax
      real gwsz
      integer n,np

      common /n/ npar,ngen
      common /p/ prob
      common /g/ g
      common /l/ lmax
      common /gr/ gwsz

c-----------------------------------------------------------------

      subroutine gen(x,n,xp,np,drand,cnts)
```

```
      include 'header'
      integer i,j,iflag,cnts
      real y(4),z(4),drand(80000)
c This program places the endpoints for parent fractures and
c any daughter fractures into an array called xp.
      np=0
      do i=1,n
c This do loop operates on each of the n fractures


c Collect the endpoints y(1), y(2), y(3), y(4) for the
c ith fracture
         do j=1,4
         y(j)=x(j,i)
       enddo
c Determine whether and where a daughter fracture will be grown
c near the tip of the ith parent fracture
      call ifsgen(y,z,drand,cnts,iflag)
c Place the parent fracture endpoints into the xp storage array
      np=np+1
      do j=1,4
        xp(j,np)=y(j)
      enddo
      if(iflag.gt.1) then
```

```
c A daughter fracture was grown near (but not at) the tip of the
c parent fracture.  Put the daughter fracture into the xp storage array.
        np=np+1
        do j=1,4
          xp(j,np)=z(j)
        enddo
        endif
      enddo
      return
      end
c-----------------------------------------------------------------
      subroutine ifsgen(y,z,drand,cnts,iflag)
      include 'header'
      integer cnts
      real y(4),z(4),drand(80000)
      real p
      integer iflag
c This subroutine decides whether to grow a new fracture near the
c tip of a pre-existing one and decides where to grow it.
c The points y(1), y(2), y(3), and y(4) mark fracture end points.
c The odd number indices are x coordinates.
c The even number indices are y coordinates.
c The parameter p is a random number between 0 and 1.
```

```
      pi = 4.*atan(1.)
c Select a random number p to determine whether the fracture
c will grow or nucleate a new fracture
c Fracture growth probabilities are scaled to
c a parameter called lmax.
      iflag=1
      p=ran1(drand,cnts)
c If p>xlen/lmax, then no fracture growth occurs,
c and the suboutine is exited.
      if(p.gt.(xlen(y)/lmax)) return
c If p<xlen/lmax, then fracture growth occurs.
c The idea is that the probability of fracture growth should
c be proportional to the fracture energy release rate G.
c G in turn, is proportional to K*K, where K is the stress
c intensity factor and is proportional to the square root of the
c crack length.  So p should be proportional to the crack length,
c which is normalized here by the parameter lmax.

c ROUTINE FOR PARENT CRACK GROWTH
c In the following scheme, the old (short) and new (long)
c parent crack have the same midpoint.
c The maximum relative growth increment is b = gwsz
c Both crack endpoints move to increase the
```

```
c crack length by b*ran1 %.
        p = ran1(drand,cnts)
        b = gwsz * p
             xm = (y(1) + y(3)) *0.5
             ym = (y(2) + y(4)) *0.5
             y(1)=(b+1.0)*y(1) - b*xm
             y(2)=(b+1.0)*y(2) - b*ym
             y(3)=(b+1.0)*y(3) - b*xm
             y(4)=(b+1.0)*y(4) - b*ym
         return
       end
c-----------------------------------------------------------------
         subroutine simulate(x,n,drand,cnts,nc)
          integer n,cnts,nc
          real drand(80000)

c Master program to recursively generate line fractures

c The following statement defines variables and common blocks
c used throughout the ifs fracture generation program:
      include 'header'
c Read the input data.
      call rdpar(drand,cnts)
```

```
c Read the initial fracture endpoints from the file start.inp
      call read4(x,n,drand,cnts,nc)
c Now enter the do loop that generates daughter fractures, filters
c out duplicate fractures, and restores the parent and daughter
c fractures back into storage array x.  The number of generations
c for which fractures can be grown is ngen.
      do i=1,ngen
       call gen(x,n,xp,np,drand,cnts)
       call filter(xp,np)
       call swap(x,n,xp,np)
      enddo
      return
      end
c-----------------------------------------------------------------
      function RANDOM(seed)
      double precision seed,m,ranhd
      integer istart
      data istart /-1/
c ref: ripley, stochastic. simulation, page 46
      if(istart.lt.0) then
       m=1.0
       do i=1,32
        m=m*2.0
```

```
        enddo
        istart=1
        ranhd=seed
       endif
c
       ranhd=mod((69069*ranhd+1.),m)
       RANDOM=float(ranhd/m)
       return
       end
c-----------------------------------------------------------------
       FUNCTION ran1(drand,cnts)
        integer cnts
        real drand(80000)

         cnts=cnts+1
         ran1=drand(cnts)
      return
      end
c-----------------------------------------------------------------
      subroutine rdpar(drand,cnts)
       integer cnts
       real drand(80000)
       include 'header'
```

```
c This subroutine reads the growth parameters from the desired
c input file par.inp.
c The lines below that say "read (15,*)bogus" read the titles
c for the growth criteria.
c ngen = number of generations in which fracture growth is allowed
c idum = seed for random number generator
c npar = # of growth rules
c lmax = maximum allowable fracture length
c prob(i) = probability for growth rule i
c g(i) = location of daughter fracture endpoints relative
c        to a parent fracture at (-0.5,0), (0.5,0)
c gwsz = growth increment % for a parent fracture that lengthens
c        (as opposed to a fracture that grows a daughter)
cnts = 0
        ngen=80+20*ran1(drand,cnts)+5
        npar=1
        lmax=ran1(drand,cnts)*0.03
        prob(1) =1.0
gwsz=ran1(drand,cnts)*0.3
      return
      end
c-----------------------------------------------------------------
      subroutine read4(x,n,drand,cnts,nc)
```

```
      integer cnts,nc
      real drand(80000)
      include 'header'
c This subroutine reads the endpoint coordinates
c for the array of starter cracks and counts how
c many starter cracks there are.  The number is n.
do 10 i=1,nc
         x(1,i)=ran1(drand,cnts)*60.0 + 15.0
         x(3,i)=x(1,i)-0.01
   10   continue
do i = nc+1,n
 x(1,i)=ran1(drand,cnts)*50.0 + 20
 x(3,i)=x(1,i)-0.01
enddo
      return
      end
c-----------------------------------------------------------------
      subroutine swap(x,n,xp,np)
      include 'header'
c This subroutine sets the x array equal to the xp array
      do i=1,np
       do j=1,4
       x(j,i)=xp(j,i)
```

```
        enddo
       enddo
       n=np
       return
       end
c-----------------------------------------------------------------
       subroutine filter(x,n)
       real x(4,10000)
       real xp(4,10000)
       integer idic(10000)
       integer n,np
       data tol /1.0e-08/
c This subroutine checks to make sure there are no
c duplicate fractures in the ouput produced by
c subroutine gen
c Copy the endpoints for each fracture from the permanent
c x array to a temporary xp array
      do i=1,n
       idic(i)=0
       do j=1,4
        xp(j,i)=x(j,i)
       enddo
      enddo
```

```
c March through the fracture array (checking each fracture
c against all the fractures that follow it in the array)
c to see whether duplicate sets of fracture endpoints occur.
c If there is a duplicate set, flag the fracture that is
c closer to the end of the array.
      do 100 i=1,n-1
       if(idic(i).gt.0) goto 100
       do 200 j=i+1,n
           if(abs(x(1,i)-x(1,j)).ge.tol) goto 200
           if(abs(x(2,i)-x(2,j)).ge.tol) goto 200
           if(abs(x(3,i)-x(3,j)).ge.tol) goto 200
           if(abs(x(4,i)-x(4,j)).ge.tol) goto 200
           idic(j)=1
  200   continue
  100 continue
c Write all unflagged (i.e. nonduplicate) sets of fracture
c endpoints back to the x array.
      np=0
      do i=1,n
      if(idic(i).lt.1) then
       np=np+1
       do j=1,4
        x(j,np)=xp(j,i)
```

```
        enddo
       endif
       enddo
c The number of fractures now in the x array is defined as n
       n=np
       return
       end
c-----------------------------------------------------------------
        subroutine create(rand,prand)
         integer i
         real rand(80000),prand(80000)
         double precision idum
         common /seed/ idum
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c inputs: rand and prand denote vectors of uniform random variables c
c outputs: intialized arrays rand and prand                         c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
         do 10 i = 1,80000
          rand(i) = RANDOM(idum)
          prand(i) = 0.0
   10    continue
         return
        end
```

```
c-------------------------------------------------------------
c-------------------------------------------------------------
        subroutine energy(x,Ematrix,n,clsnum,min,max,l)
         integer i,n,clsnum,jdum
         real x(4,10000),l(10000,5),Ematrix(1000),min,max,cw(7)
         data cw(1)/0.9227/,cw(2)/1.0308/,cw(3)/1.1294/,cw(4)/1.253/,
     $        cw(5)/1.3884/,cw(6)/1.5342/,cw(7)/1.78159/
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c inputs: x=the fracture matrix, Ematrix stores energy measures,n=  c
c         the # of fractures in x, clsnum=# of frequency classes,   c
c         min=minimum crack length in x, max=maximum crack length   c
c         in x, l=vector of fracture lengths in x.                  c
c output: Ematrix comes out of this routine storing the class       c
c         frequencies for a histogram of the fracture lenghts in x. c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
         do i = 1,clsnum+1
          Ematrix(i) = 0.0
         enddo
         do i =1,n
          if (l(i,1).LT.cw(1)) Ematrix(1) = Ematrix(1) + 1
          if (l(i,1).GE.cw(1).AND.l(i,1).LT.cw(2)) then
           Ematrix(2) = Ematrix(2) + 1
          end if
```

```
      if (l(i,1).GE.cw(2).AND.l(i,1).LT.cw(3)) then
       Ematrix(3) = Ematrix(3) + 1
      end if
      if (l(i,1).GE.cw(3).AND.l(i,1).LT.cw(4)) then
       Ematrix(4) = Ematrix(4) + 1
      end if
      if (l(i,1).GE.cw(4).AND.l(i,1).LT.cw(5)) then
       Ematrix(5) = Ematrix(5) +1
      end if
      if (l(i,1).GE.cw(5).AND.l(i,1).LT.cw(6)) then
       Ematrix(6) = Ematrix(6) + 1
      end if
      if (l(i,1).GE.cw(6).AND.l(i,1).LT.cw(7)) then
       Ematrix(7) = Ematrix(7) + 1
      end if
      if (l(i,1).GE.cw(7)) Ematrix(8) = Ematrix(8) + 1
     enddo
     do i = 1,n
      Ematrix(clsnum+1) = Ematrix(clsnum+1) + l(i,2)
     enddo
    return
    end
c-----------------------------------------------------------------
```

```
      subroutine compare(E,E3,en3,flag,clsnum)
       integer en3,clsnum
       real E(1000),E3(1000),flag
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c inputs: E=#energy matrix of the natural pattern,E3=#energy matrix c
c         of a simulated pattern, clsnum=# of frequency classes in  c
c         E and E3                                                  c
c output: en3=energy of the simulated pattern.                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       flag = 0.0
       en3 = 0
       do i = 1,clsnum+1
        en3 = en3 + abs(E3(i)-E(i))
       enddo
       if (en3.LT.31) flag = 1.0
      return
      end
c-----------------------------------------------------------------
      subroutine pfactor(pf,pprob)
       real pf,pprob
       double precision idum
       common /seed/ idum
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

```
c inputs: pprob=a test probability.                                    c
c output: pf=flag to perturb.                                          c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
          pf = 0.0
          if (RANDOM(idum).LT.pprob) pf =  1.0
         return
         end
c-----------------------------------------------------------------------
         subroutine perturb(rand,prand,cnts)
          integer i,j,cnts,m,n,en1
          real pf,rand(80000),prand(80000),pprob
          double precision idum
          common /seed/ idum
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c inputs: rand=array of uniform random variables, cnts=# of            c
c          elements in rand.                                           c
c output: prand=perturbed rand.                                        c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
         if (RANDOM(idum).LT.0.5) then
         do i = 1,3
          prand(i) = rand(i)
         enddo
         i = int(3*RANDOM(idum)) + 1
```

```
          prand(i) = rand(i) + RANDOM(idum)
          if (prand(i).GT.1.0) prand(i) = prand(i) - 1.0
          else
          do j = 4,cnts
           pprob = 0.25
           call pfactor(pf,pprob)
           prand(j) = rand(j) + pf*RANDOM(idum)
           if (prand(j).GT.1.0) prand(j) = prand(j) - 1.0
           prand(j) = rand(j)
          enddo
          end if
         return
         end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c NOTE: This is not necessarily the best way to perturb rand.      c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c-----------------------------------------------------------------------
         subroutine solution(n,x,u1,u2,u3,pcnts)
          integer ngen,n,pcnts
          real x(4,10000),u1,u2,u3,lmax,gwsz
          double precision idum
          common /seed/ idum
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

```
c inputs: x=fracture matrix,n=# of fractures,u1-3=coded parameters c
c          ngen,lamx, and gwsz,pcnts=# of metropolis iteraations to c
c          get u1-3.                                                c
c output: A file of ngen,lmax,gwsz, and pcnts                       c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
          ngen = 80 + u1*20.0 + 5
          lmax = u2*0.03
          gwsz = u3*0.3
          write(15,*) pcnts,ngen,lmax,gwsz
          call flush(15)
          if (RANDOM(idum).LT.0.05) then
           do i = 1,n
            write(16,*) x(1,i)-20,x(2,i),x(3,i)-20,x(4,i)
           call flush(16)
           enddo
          end if
         return
         end
c-----------------------------------------------------------------------
         subroutine length(x,n,l,xl,xu)
          integer i,n
          real x(4,10000),l(10000,5),xl,xu,cx1,cx2
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

```
c inputs: x=fracture matrix,n=# of fractures,xu=upper bound on      c
c          the x-coordinates in x,xl=lower bound on the x-coord-    c
c          inates in x.                                             c
c output: l=double array of the lengths of fractures in x and a     c
c          flag indicating if the fracture is censored.             c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
          do i = 1,n
           l(i,1) = 0.0
           l(i,2) = 0.0
           cx1 = x(1,i)
           cx2 = x(3,i)
           if (cx1.GT.xu) then
            cx1 = xu
            l(i,2) = 1.0
           end if
           if (cx2.LT.xl) then
            cx2 = xl
            l(i,2) = 1.0
           end if
           l(i,1) = cx1 - cx2
           if (l(i,1).EQ.0.0) l(i,2) = 1.0
          enddo
         return
```

```
      end
c-----------------------------------------------------------------------
      subroutine maxmin(l,n,min,max)
       integer i,n
       real min,max,l(10000,5)
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c inputs: l=double array of fracture lengths with a censor flag,n= c
c         of fractures.                                              c
c output: min=minimum in l, max=maximum in l                         c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       min = 1000000
       max = 0
       do i = 1,n
        if (l(i,2).NE.1.0) then
         if (l(i,1).LT.min) min = l(i,1)
         if (l(i,1).GT.max) max = l(i,1)
        end if
       enddo
       if (max.GT.50.OR.max.EQ.0) then max = 50.0
       if (min.GT.50) then min = 50.0
      return
      end
c-----------------------------------------------------------------------
```

```
      subroutine stats(En,pcnts,clsnum)
       real En(1000)
       integer pcnts,clsnum
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c inputs: En=energy matrix for a matching pattern, pcnts=# of      c
c         metropolis iterations to get En, clsnum=working row      c
c         rank of En.                                              c
c output: A file of the entries in En with pcnts.                  c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       do i = 1,clsnum+1
        write(17,*) pcnts,En(i)
       enddo
       call flush(17)
      return
      end
c---------------------------------------------------------------------
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                    c
c                              MAIN                                  c
c                                                                    c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c This program applies conditional coding to the problem detailed in  c
c Chapter 2. Not everything in this code is optimal, but it works and c
```

```
c generated the large data set in Chapter 4. The notes that accompany c
c each subroutine in this program should help the reader follow how   c
c the conditional coding reciepe at the end of Chapter # is coded.    c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      integer n,en1,en2,edum,samsize,idiv,cnts,pcnts,nc,limit,
     $          clsnum
      real E,E1,E2,rand,prand,x,xl,xu,yb,flag,t,unif,rinc,r,min,max,l
      dimension rand(80000),x(4,10000),E1(1000),prand(80000),E(1000),
     $           E2(1000),l(10000,5)
      double precision idum
      common /seed/ idum
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c initialize the appropriate variables.                               c
c n=number of fractures in the natural pattern.                       c
c cnts=loop counting variable.                                        c
c idum=random number generation seed.                                 c
c yb,xl,xu=boundaries for a rectangular growth region.                c
c clsnum=# of histogram classes used to find energy.                  c
c t,idiv,rinc,m=variables used to implement Simmulated Annealing.     c
c edum=energy flag that tracts the lowest energy obtained.            c
c limit=# of iterations that must pass between solutions so that the c
c       data points are independent.                                  c
```

```
c samsize=# of sample points to collect before ending the program.   c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
        n = 200
        cnts = 0
        idum = 12005
        yb = 50.0
        xl = 20.0
        xu = 70.0
        clsnum = 8
        t = 19.00
        k = 0
        m = 1
        pcnts = 0
        idiv = 45
        rinc = 1.0
        edum = 5000
        limit = 10000
        samsize = 1000
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Read the natural pattern and assign it an energy matrix by find-  c
c ing the lengths it's  fractures, nc=3 of censored fractures.      c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
        open(unit=11,file='markov.out',status='old')
```

```
      do 10 i=1,n
       read(11,*) x(1,i),x(2,i),x(3,i),x(4,i)
       x(1,i) = x(1,i) + 20
       x(3,i) = x(3,i) + 20
10    continue
      call length(x,n,l,xl,xu)
      call maxmin(l,n,min,max)
      call energy(x,E,n,clsnum,min,max,l)
      nc = E(clsnum+1)
      call stats(E,pcnts,clsnum)
      call create(rand,prand)
      call simulate(x,n,rand,cnts,nc)
      call length(x,n,l,xl,xu)
      call energy(x,E1,n,clsnum,min,max,l)
      call compare(E,E1,en1,flag,clsnum)
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c What follows is a series of commands designed to slowly lower the c
c temperature during our application of the Metropolis Algotithm.   c
c Also we record the energy of the simulated pattern in the last    c
c iteration if it is lower than any previous energy.                c
c The way this is coded annealing is ignored.                       c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      r = 17.98/18.98
```

```
20    if (edum.GT.en1) then
       write(18,*) t,pcnts,en1,en2
       call flush(18)
       edum = en1
      end if
      if (m/idiv.EQ.1) then
       m = 1
       if (t.LT.0.5) then
        rinc = rinc*r
        t = t - rinc
       end if
      end if
      print*, t,pcnts,en1,en2
      if (k.LT.2) write(14,*) en1,en2
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c This is the Metropolis Algorithm                                    c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      pcnts = pcnts + 1
      call perturb(rand,prand,cnts,m,n,en1)
      call simulate(x,n,prand,cnts,nc)
      call length(x,n,l,xl,xu)
      call energy(x,E2,n,clsnum,min,max,l)
      call compare(E,E2,en2,flag,clsnum)
```

```
          if (flag.EQ.1.0.AND.pcnts.GT.limit) then
           write(18,*) t,pcnts,en1,en2
           call flush(18)
           call solution(n,x,prand(1),prand(2),prand(3),pcnts)
           call stats(E2,pcnts,clsnum)
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c The following lines are commented out because there are two ways c
c to get solutions. The first is to use the Metropolis Algorithm   c
c to find a first solution and then create a new rand vector and   c
c start over. This is what the commented lines do. The second is   c
c to continue as if a solution was not found at all. Doing this we c
c cannot accept a new solution until after many more iterations in c
c the Metropolis Algorithm so that the next solution is independ-  c
c ent of the previous one. this is why pcnts must be greater than  c
c limit before we call the solution routine.                       c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c         call create(rand,prand)
c         call simulate(x,n,rand,cnts,nc)
c         call length(x,n,l,xl,xu)
c         call energy(x,E1,n,clsnum,min,max,l)
c         call compare(E,E1,en1,flag,clsnum)
          k = k + 1
          m = 1
```

```
      t = 19.00
      rinc = 1.0
      pcnts = 0
      edum = 5000
     else
      p = exp(real((en1-en2)/T))
      unif = RANDOM(idum)
      if (p.GT.unif.OR.en1.EQ.en2) then
       do i = 1,cnts
        rand(i) = prand(i)
       enddo
       en1 = en2
      end if
     end if
     m = m+1
     if (k.LT.samsize) go to 20
    end
```

A.3 Kolmogorov Smirnov Test

The following code computes the Kolmogorov Smirnov test for the equality of two distributions of data. A good explanation of the test is given by Conover (1980) and by Hollander and Wolfe (1973). Critical values for this test are given by Conover (1980).

The test includes a subroutine SORT2 from a book of examples of numerical recipes by Vetterling (1985). SORT2 sorts a two dimensional array. The subroutine TEST performs the Kolmogorov Smirnov test by finding the largest difference between the emperical distributions of the two data sets. The main driver reads two data sets of lengths m and n respectively and then calls the routines necessary to test the hypothesis that the two distributions the data represent are equal. This is a two-sided test.

```
      SUBROUTINE SORT2(N,RA,RB)
      DIMENSION RA(N),RB(N)
      L=N/2+1
      IR=N
10    CONTINUE
        IF(L.GT.1)THEN
          L=L-1
          RRA=RA(L)
          RRB=RB(L)
        ELSE
          RRA=RA(IR)
```

```
        RRB=RB(IR)
        RA(IR)=RA(1)
        RB(IR)=RB(1)
        IR=IR-1
        IF(IR.EQ.1)THEN
          RA(1)=RRA
          RB(1)=RRB
          RETURN
        ENDIF
      ENDIF
      I=L
      J=L+L
20    IF(J.LE.IR)THEN
        IF(J.LT.IR)THEN
          IF(RA(J).LT.RA(J+1))J=J+1
        ENDIF
        IF(RRA.LT.RA(J))THEN
          RA(I)=RA(J)
          RB(I)=RB(J)
          I=J
          J=J+J
        ELSE
          J=IR+1
```

```
        ENDIF
       GO TO 20
       ENDIF
       RA(I)=RRA
       RB(I)=RRB
     GO TO 10
     END


subroutine TEST(x,y,m,n,tot,max)
      integer i,m,n,tot,xcnt,ycnt
      real z1(10000000),z2(10000000),x(10000000),y(10000000),
    $       max,s(10000000),f(10000000),g(10000000)


      do i = 1,m
       z1(i)=x(i)
       z2(i)=1.0
      enddo
      do i = 1,n
       z1(m+i)=y(i)
       z2(m+i)=-1.0
      enddo
      call SORT2(tot,z1,z2)
      xcnt = 0
```

```
      ycnt = 0
      do i =1,tot
       if (z2(i).EQ.1.0) then
        xcnt = xcnt + 1
       else
        ycnt = ycnt + 1
       end if
       f(i) = real(xcnt)/real(m)
       g(i) = real(ycnt)/real(n)
       s(i) = abs(f(i)-g(i))
       if (s(i).GT.max) max = s(i)
      enddo
     return
     end

c    ***     main     ***     c

     integer m,n,tot
     real x,y,max
     dimension x(10000000),y(10000000)

     m = 100001
     n = 100001
```

```
open(unit=11,file='test.x',status ='old')
do i = 1,m
 read(11,*) x(i)
enddo
close(unit=11,status='keep')
open(unit=12,file='test.y',status ='old')
do i = 1,n
 read(12,*) y(i)
enddo
close(unit=12,status='keep')
tot = n + m
max = 0.0
call TEST(x,y,m,n,tot,max)
write(999,*) max
end
```

APPENDIX B

DATA

B.1 Sampled Data

DATA VALUES

obs #	required iterations	parameters		
		ngen	*lmax*	*gwsz*
1	131259	86	1.28736E-02	0.117111
2	22448	101	1.69371E-03	9.84146E-02
3	104125	90	1.12279E-02	0.110570
4	17331	101	8.37228E-03	9.83248E-02
5	13204	101	4.98033E-03	9.84111E-02
6	24956	101	2.31331E-03	9.85148E-02
7	32995	101	1.28499E-02	9.94313E-02
8	19801	90	1.12652E-02	0.110579
9	44540	101	7.85158E-03	9.84165E-02
10	15580	101	6.99394E-03	9.85154E-02
11	132991	104	1.43033E-02	9.75941E-02
12	71489	101	6.63592E-03	9.84599E-02
13	15340	101	9.42496E-03	9.85250E-02
14	96964	90	1.12974E-02	0.110465
15	40859	102	1.57606E-02	1.01201E-01
16	25511	101	1.52725E-03	9.82259E-02
17	10264	101	9.53623E-03	9.84867E-02
18	11316	101	3.03582E-03	9.82606E-02
19	10203	101	2.19245E-03	9.82087E-02
20	97660	101	1.42278E-02	1.00223E-01
21	93761	95	1.19243E-02	0.105466
22	105646	92	1.34159E-02	0.111375
23	12904	101	7.55368E-03	9.82602E-02
24	38999	101	7.76191E-03	9.84288E-02
25	28505	101	2.23071E-03	9.84118E-02
26	27081	101	1.14812E-03	9.84192E-02
27	43374	101	3.92505E-03	9.83381E-02
28	67877	101	4.87672E-03	9.82180E-02
29	53394	101	7.53962E-03	9.82390E-02
30	83462	100	1.53068E-02	1.01670E-01
31	37667	101	6.77138E-03	9.84965E-02

DATA VALUES CONTINUED

obs #	required iterations	parameters		
		ngen	*lmax*	*gwsz*
32	24317	101	9.86650E-03	9.82628E-02
33	23367	91	1.40550E-02	0.111844
34	76540	101	3.06710E-03	9.83142E-02
35	27336	101	9.87238E-03	9.82883E-02
36	33471	101	3.50080E-03	9.85292E-02
37	11721	101	4.63842E-03	9.84059E-02
38	13028	101	7.19305E-03	9.84980E-02
39	12423	101	5.08360E-03	9.84675E-02
40	55159	101	9.02065E-03	9.82998E-02
41	25015	101	5.39893E-03	9.83130E-02
42	46075	101	9.01157E-03	9.85434E-02
43	36210	101	4.93131E-03	9.82280E-02
44	100482	101	6.21190E-03	9.85224E-02
45	26503	101	8.35376E-04	9.82204E-02
46	37273	101	3.78901E-03	9.82824E-02
47	10640	101	9.36311E-03	9.82233E-02
48	33049	92	1.34244E-02	0.111474
49	59334	101	6.87519E-03	9.83858E-02
50	34569	101	1.77815E-03	9.84845E-02
51	19577	101	5.54681E-05	9.84149E-02
52	19364	101	9.85733E-03	9.85168E-02
53	55954	100	1.24660E-02	1.01159E-01
54	15944	101	6.32383E-03	9.82649E-02
55	54256	101	7.46771E-03	9.83064E-02
56	20405	101	7.98046E-03	9.83890E-02
57	49735	104	1.24787E-02	9.66182E-02
58	216597	101	2.32578E-03	9.82836E-02
59	13733	101	8.18514E-03	9.85344E-02
60	20005	101	6.77731E-03	9.84240E-02
61	16430	101	6.15324E-03	9.85013E-02
62	32907	101	2.89702E-03	9.83157E-02
63	41222	101	2.65247E-03	9.83545E-02
64	87993	85	1.51219E-02	0.120508
65	24119	101	2.90186E-04	9.84594E-02
66	64560	90	1.12212E-02	0.110657
67	52652	90	1.12800E-02	0.110498
68	63293	101	3.28148E-03	9.85211E-02
69	27962	101	3.99335E-03	9.84768E-02
70	146719	90	1.12071E-02	0.110639

DATA VALUES CONTINUED

obs #	required iterations	parameters		
		ngen	*lmax*	*gwsz*
71	19876	101	5.13900E-04	9.83643E-02
72	27970	101	4.84029E-03	9.84825E-02
73	35644	101	1.20116E-03	9.85356E-02
74	26525	101	1.76375E-03	9.84756E-02
75	21072	101	8.89209E-03	9.83307E-02
76	11483	101	3.83527E-03	9.84937E-02
77	20204	101	2.90214E-03	9.82420E-02
78	43199	101	3.42522E-04	9.84086E-02
79	57568	101	9.06471E-03	9.82422E-02
80	48496	101	4.31197E-03	9.84145E-02
81	10716	101	6.34428E-03	9.83766E-02
82	10312	92	1.39805E-02	0.109835
83	47110	101	6.57295E-04	9.85187E-02
84	88872	101	6.52332E-03	9.83291E-02
85	30012	101	1.44113E-02	1.00280E-01
86	81054	101	9.27806E-03	9.84869E-02
87	14227	101	8.86546E-03	9.84889E-02
88	15101	101	3.43804E-03	9.83184E-02
89	15078	90	1.11910E-02	0.110703
90	20726	101	2.80284E-03	9.83711E-02
91	205064	101	8.90699E-03	9.83328E-02
92	10036	101	3.69026E-03	9.83785E-02
93	10119	101	2.12618E-03	9.82579E-02
94	18330	87	1.53008E-02	0.117522
95	41612	101	1.39713E-03	9.85072E-02
96	12711	100	1.24638E-02	1.01020E-01
97	41915	102	1.57533E-02	1.01012E-01
98	11496	100	1.54658E-02	1.02961E-01
99	10004	101	8.89243E-03	9.85033E-02
100	27978	101	5.38666E-03	9.82781E-02

B.2 Chi-Square Goodness-of-Fit Results

CHI-SQUARE RESULTS

obs #	Chi-Square Statistics	p -value
1	0.625326485109094	0.00031048598939689
2	0.492084726867336	0.000125516248714271
3	0.658751393534002	0.000377362085777758
4	0.485987418378723	0.000119699106176212
5	0.485987418378723	0.000119699106176212
6	0.664186176142698	0.000389132938459921
7	1.24367892976589	0.00380866424230806
8	0.563311833094442	0.000209543310494334
9	0.492084726867336	0.000125516248714271
10	0.664186176142698	0.000389132938459921
11	1.32820512820513	0.00479345792811754
12	0.492084726867336	0.000125516248714271
13	0.664186176142698	0.000389132938459921
14	0.65321707278229	0.000365639228936025
15	0.654264214046823	0.000367837051941046
16	0.629925147316452	0.00031913924545982
17	0.492084726867336	0.000125516248714271
18	0.701831501831502	0.000477977091310066
19	0.629925147316452	0.00031913924545982
20	0.974839146360885	0.00159760066510796
21	0.828685300207039	0.000883665410361491
22	1.01276556776557	0.00183356986019414
23	0.554688644688645	0.000197678352252847
24	0.492084726867336	0.000125516248714271
25	0.485987418378723	0.000119699106176212
26	0.487799012581621	0.000121406621035923
27	0.404047619047619	5.90809950045466e-05
28	0.629925147316452	0.00031913924545982
29	0.482782290173595	0.000116720818443436
30	0.974041248606466	0.0015928762872764
31	0.492084726867336	0.000125516248714271
32	0.400842490842491	5.73014471431891e-05
33	1.17257525083612	0.00309451500817984
34	0.404047619047619	5.90809950045466e-05
35	0.400842490842491	5.73014471431891e-05
36	0.664186176142698	0.000389132938459921
37	0.485987418378723	0.000119699106176212
38	0.492084726867336	0.000125516248714271

CHI-SQUARE RESULTS CONTINUED

obs #	Chi-Square Statistics	p -value
39	0.492084726867336	0.000125516248714271
40	0.480970695970696	0.000115061384607836
41	0.485987418378723	0.000119699106176212
42	0.664186176142698	0.000389132938459921
43	0.554688644688645	0.000197678352252847
44	0.664186176142698	0.000389132938459921
45	0.629925147316452	0.00031913924545982
46	0.400842490842491	5.73014471431891e-05
47	0.629925147316452	0.00031913924545982
48	0.855677655677656	0.000993915025075553
49	0.485987418378723	0.000119699106176212
50	0.492084726867336	0.000125516248714271
51	0.492084726867336	0.000125516248714271
52	0.664186176142698	0.000389132938459921
53	0.730028666985189	0.000553346571761239
54	0.400842490842491	5.73014471431891e-05
55	0.480970695970696	0.000115061384607836
56	0.485987418378723	0.000119699106176212
57	1.10636645962733	0.00251708594174349
58	0.400842490842491	5.73014471431891e-05
59	0.664186176142698	0.000389132938459921
60	0.485987418378723	0.000119699106176212
61	0.492084726867336	0.000125516248714271
62	0.404047619047619	5.90809950045466e-05
63	0.485987418378723	0.000119699106176212
64	0.973172479694219	0.00158774332675998
65	0.492084726867336	0.000125516248714271
66	0.658751393534002	0.000377362085777758
67	0.577502787068004	0.000230172500490363
68	0.664186176142698	0.000389132938459921
69	0.492084726867336	0.000125516248714271
70	0.735273132664437	0.000568241273288539
71	0.485987418378723	0.000119699106176212
72	0.492084726867336	0.000125516248714271
73	0.664186176142698	0.000389132938459921
74	0.492084726867336	0.000125516248714271
75	0.404047619047619	5.90809950045466e-05
76	0.664186176142698	0.000389132938459921
77	0.554688644688645	0.000197678352252847
78	0.485987418378723	0.000119699106176212

CHI-SQUARE RESULTS CONTINUED

obs #	Chi-Square Statistics	p -value
79	0.862380952380952	0.00102272277104632
80	0.492084726867336	0.000125516248714271
81	0.485987418378723	0.000119699106176212
82	1.02501194457716	0.00191464284915403
83	0.664186176142698	0.000389132938459921
84	0.485987418378723	0.000119699106176212
85	0.579900461856984	0.000233796774607755
86	0.492084726867336	0.000125516248714271
87	0.492084726867336	0.000125516248714271
88	0.404047619047619	5.90809950045466e-05
89	1.10579710144928	0.00251246964781495
90	0.485987418378723	0.000119699106176212
91	0.485987418378723	0.000119699106176212
92	0.485987418378723	0.000119699106176212
93	0.400842490842491	5.73014471431891e-05
94	1.28080267558528	0.00422243135664749
95	0.664186176142698	0.000389132938459921
96	0.939705367096671	0.00139863742304815
97	1.39899108138239	0.0057391488030792
98	0.58989409141583	0.000249345860405259
99	0.664186176142698	0.000389132938459921
100	0.400842490842491	5.73014471431891e-05