STUDY OF SIMULATED ANNEALING FOR LEAST SQUARES OPTIMIZATION

by

PRANAB KUMAR BANERJEE

B.Tech. (Hons.) Indian Institute of Technology, Kharagpur, 1985

A MASTER'S THESIS
submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Physics
KANSAS STATE UNIVERSITY
Manhattan, Kansas
1988

Approved by:

O. L. Weaver

Major Professor

ACKNOWLEDGEMENT

First of all I would like to express my special thanks for my father and mother for their love, encouragement, and guidance. I would like to express my sincere thanks to Professor O. L. Weaver for his valuable advice on this thesis work. I would like to thank Dr. R. M. Panoff for introducing me to the problem and helping me to develop the main ideas behind this thesis. I also thank Dr. Legg and Dr. McGuire for being in the committee and providing me with useful suggestions.

CONTENTS

CHAPTER I

INTRODUCTION

Optimization can be defined as the task of finding the minimum or maximum value of any function. There are different types of problems in this field of optimization, and so there also are different techniques of optimization. This thesis deals with a stochastic optimization technique for optimization of functions which depend on many independent parameters.

Optimization problems can be "constrained" or "unconstrained". In the first case, the function is to be optimized by taking care of the fact that it must always satisfy a set of predefined conditions or "constraints". For eaxmple, in this thesis, we minimize a function with the constraint that the independent variables can never be negative. This is, therefore, a constrained optimization problem. On the other hand, if the function is free from any constraints, then we have the unconstrained optimization problem.

The complexity of an optimization task depends on the complexity of the function to be optimized, and also on the constraints involved. In fact, the complexity is directly related to the number of independent variables governing the function, or to say technically, on the degrees of freedom of the function. The higher this degree of freedom, the more complex the optimization problem is. Technically, the function that is to be minimized or maximized is called the "cost function" or the "objective function". In practice, it is not very uncommon

to come across situations where the optimization process involves hundreds of independent variables, resulting in a very high dimensional problem, and in these cases, the conventional deterministic approaches do not work very efficiently, or sometimes, are practically inapplicable because of the constraints in time and machine computability. Even the familiar technique of matrix inversion can be computationally very cumbersome for large matrices. Besides, this familiar technique of matrix inversion may fail in those situations where the matrices are singular and hence non-invertible. In these cases, stochastic optimization approaches produces a higher probability of finding the optimized state with higher computational efficiency. The stochastic optimization approach studied in this thesis is that of "Simulated Annealing" which will be discussed later in this thesis. This optimization technique is so called because of the close similarity of the concept behind it to the process of metallurgical annealing.

In this work, the method of "simulated annealing" has been applied for solving a least squares optimization with a particular application to the inverse problem. The goal here is to study the effects of several parameters (discussed in later chapters) on a particular simulated annealing algorithm developed here, and we do not intend to find the best possible optimization algorithm. In fact, the results obtained by the current algorithm are quite poor. However, they do show the effects of the different parameters on the algorithm.

We give here a map of the contents of this thesis. Chapter II explains the inverse problem that is dealt with in this thesis. Chapter III explains stochastic optimization techniques, and compares them to deterministic

optimization techniques. **Chapter IV** explains the application of simulated annealing to least squares optimization with particular application to the inverse problem. Finally **Chapter V** presents the results and application.

CHAPTER II

THE INVERSE PROBLEM

In this chapter we will give a general description of a class of inverse problems, along with the particular inverse problem that is of interest in this thesis.

An important type of problem arises when we know a function $g(a,b)$, $a \in R$, $b \in R$, (where R refers to the real space), with

$$g(a,b) = \int_{\Omega} K(a,b,\omega) f(\omega) d\omega \tag{2.1}$$

where K is a known function called the 'Kernel', f is an unknown function, and ω is a variable of integration such that $\omega \in \Omega$, the domain of integration, and the task is to construct the function f, the unknown function [1]. From the computational point of view, the task is to find $\{f(\omega_i)\}_{i=1}^{k}$, $k \in N$, for a set of values of $\omega_i \in \Omega$, given a set of values of g (i.e. the set $\{g_1, g_2, ..., g_k\}$) while K is a known function. Here, g_i means $g(a_i, b_i)$ where a_i and b_i are some values of a, and b. This class of problems is very commonly encountered in many fields, for example in analyzing remote sensing data, reconstructing three dimensional images from a set of two dimensional perspectives, etc.

There are various methods that can be used to solve this inverse problem. One may apply the method of matrix inversion [2] by converting this continuous problem into a discrete one, with the assumption that the function $f(\omega_i)$ remains fairly constant in the neighborhood of the coordinates $\{\omega_i\}_{i=1}^{k}$. The discretization is implemented by segmenting the continuous domain Ω of the function f into k

discrete regions over which f is assumed to remain constant. In this case, we can write down k equations corresponding to the set of k values of $\{g_1, g_2, ..., g_k\}$ as

$$g_1 = K'(a_1, b_1, \omega_1)f(\omega_1) + K'(a_1, b_1, \omega_2)f(\omega_2) + ... + K'(a_1, b_1, \omega_k)f(\omega_k)$$

$$g_2 = K'(a_2, b_2, \omega_1)f(\omega_1) + K'(a_2, b_2, \omega_2)f(\omega_2) + ... + K'(a_1, b_1, \omega_k)f(\omega_k)$$

.....

$$g_k = K'(a_k, b_k, \omega_1)f(\omega_1) + K'(a_k, b_k, \omega_2)f(\omega_2) + ... + K'(a_k, b_k, \omega_k)f(\omega_k)$$

Here, $K'(a_i, b_i, \omega_j)$ is the value of the Kernel over a neighborhood of ω_j computed through the relation

$$K'(a_i, b_i, \omega_j) = \int_{\Omega_j} K(a_i, b_i, \omega) d\omega$$

where Ω_j is the neighborhood of ω_j. The above set of equations can be rewritten in matrix notation as

$$\tilde{G} = \tilde{K}\tilde{F} \tag{2.2}$$

where G is the column matrix

$$\tilde{G} = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{pmatrix}$$

$\tilde{K}$ is the coefficient matrix whose (ij)th element is given by $\{k_{ij}\} = K'(a_i, b_i, \omega_j)$, and $\tilde{F}$ is the matrix given by

$$\tilde{F} = \begin{pmatrix} f(\omega_1) \\ f(\omega_2) \\ \vdots \\ f(\omega_k) \end{pmatrix}$$

So, it is easy now to see that the inverse problem boils down to solving the above matrix equation to get $\tilde{F}$, and this can be achieved through the relation

$\tilde{F} = \tilde{K}^{-1}\tilde{G}$. Now it is worth noting that during the manipulations of the equations so far, it has been assumed that the number of data points g_i is at least equal to the number of segments of Ω over which we intend to find the values of the function f. Hence,this method fails to produce any unique solution if we have insufficient number of data points in the set g_i. On the other hand, if we have more than k data points, there are different ways of handling the situation. One of them is to consider k data out of the whole set such that these are the least correlated among themselves, so that we get the maximum information out of them, and then proceed along the matrix inversion technique described above. Even if we have sufficient data, it is not very uncommon in practice to find situations where the inversion of the matrix $\tilde{K}$ happens to be an unstable process because of the presence of very small entries in the matrix, resulting in the inapplicability of this method of solving for $\tilde{F}$ by coefficient-matrix inversion [2]. This method, therefore, has a very restricted use since its applicability depends on a very stringent set of conditions which are quite often not satisfied in practical problems.

To avoid these difficulties, the inverse problem can be solved as an optimization problem, where we try to approximate the values of f over the segmented domain Ω, as described above, and the goodness of the approximation is measured through a "cost function" C defined by

$$C = \sum_{i=1}^{k} (g_i - \tilde{g}_i)^2 \tag{2.3}$$

where g_i are the actual data and $\tilde{g}_i$ are the values of the function g corresponding to the approximate values of $f(\omega_i)$. It is clear that the motivation behind this technique is to find the set of approximate values $\{f^*(\omega_i)\}$ corresponding to the

minimum value of the "cost function" C which is the sum of the squared error for each set of locations of the transmitter and the receiver. This is, therefore, a least squares minimization problem, and hence can be solved by different minimization or optimization techniques. The conventional minimization techniques, such as the gradient search method, work well if the cost function is purely convex, and it does find the global minimum in a computationally efficient manner. But this does not perform so well in cases where the cost function is not uniformly convex, and has local minima. In this case, it can easily produce a local minimum of the cost function, and may not reach the global minimum. Besides, all exact methods for optimization require exponential increase in computing effort with the number of dimensions N [3], so that in practice, exact solutions can be attempted only on problems involving a few hundred independent parameters. A different optimization technique is therefore required to treat this difficulty, and simulated annealing, which is described in the next chapter, is one such remedy.

We will now describe the particular inverse problem that has been used in this thesis. Physically, this problem can be described as finding the index of refraction of the different portions of the Earth's structure below its surface, from data taken only on the surface. This problem was solved by Professor A. Ramm at the Department of Mathematics at Kansas State University, and a treatment on the recovery of underground layers from data taken only on the surface can be found in [4]. The inverse problem dealt with in this thesis is a modification of the problem in [5], and here the objective is to recover the index of refraction as a function of x and y as well as z, and hence the current problem is an optimization with more degrees of freedom than the problem in [4]. As will

be shown here, mathematically this problem is equivalent to an inverse problem which can often result in a singular coefficient-matrix, as described above, and hence demands special attention as far as the methods of solving this problem are concerned. The problem of detecting geophysical strata from surface data is of much practical importance in fields like oil exploration, and geophysical fault detection for the prediction of earthquakes, etc. For this purpose, seismic signal generators are used which can generate signals that can travel through the Earth's crust, and are reflected by the different layers of the underground strata to appear back on the surface of the Earth with some modifications, which are intimately related to the underground structure. These reflected signals are the sources of information about the structure of the strata below. Mathematically, the reflected signal properties are functions of the index of refraction of different regions of the underground strata, and also of the locations of the signal transmitter and receiver. A function that can be determined from the reflected signal amplitude is $g(a, b)$ given by [5]

$$g(a,b) = \int_{\Omega} \frac{w(\bar{z})}{\mid \bar{z} - a \mid . \mid \bar{z} - b \mid} . d\bar{z} \tag{2.4}$$

where $w(\bar{z})$ is the difference of the index of refraction at the point $\bar{z} \in \Omega$ from an assumed constant background value, and $a \in A$, $b \in B$ where A and B are sets

$$A = \{a : a \; is \;\; a \;\; location \;\; of \;\; the \;\; transmitter\} \tag{2.4a}$$

$$B = \{b : b \;\; is \;\; a \;\; location \;\; of \;\; the \;\; receiver\} \tag{2.4b}$$

In practice, the function in the above equation that is determined from experiment is $g(a, b)$ and the function to be computed is $w(\bar{z})$ for different values of $(\bar{z})$. This is an inverse problem when looked at from a mathematical point of view, because

when compared with the equation (2.1) above, it is seen that $w(\bar{z})$ corresponds to $f(\omega)$ and $\frac{1}{|\bar{z}-a|.|\bar{z}-b|}$ corresponds to $K(\omega)$.

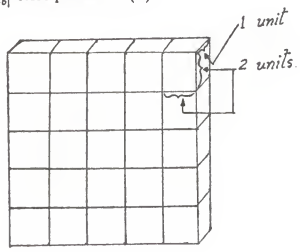

Fig - 1

In the present problem, a three dimensional block of Earth's crust is considered, (see figure above) and its dimensions are chosen to be of 1, 10, and 10 units in the x, y, and z directions respectively. The x, and the y axes are aligned along the surface of the Earth, and the z axis is perpendicular to the surface pointing directly inwards. This is the region Ω for our integration. Ω is then segmented into 25 boxes, each having a volume of 4 cubic units. The segmented Ω is shown above. Once segmented, it is assumed that the index of refraction remains constant over each box, and the integral in equation (2.4) above is computed over each box. Therefore, the equation for $g(a,b)$ can then be transformed into the following form:

$$g(a,b) = \sum_{i=1}^{25} w_i \int_{box_i} \frac{1}{|\bar{z}-a|.|\bar{z}-b|} . d\bar{z} \qquad (2.5)$$

where box_i refers to the segmented region corresponding to the ith box of Ω, and w_i refers to the constant value of the index of refraction over the box_i. Therefore, we see that once the integrals over each of the boxes are computed, it is possible

to compute the values of $g(a,b)$ for any set of $\{\omega_i\}$ by simply multiplying the integrated value by the corresponding w_i, called **weight** hereafter, and adding these products over all the boxes. The objective then is to treat this inverse problem as a least squares optimization problem, and hence minimize the cost function

$$C = \sum_{(a,b)\in(A\times B)} (g(a,b) - \tilde{g}(a,b))^2 \tag{2.6}$$

where $A \times B$ is the Cartesian product of the two sets A and B.

This is briefly the description of the inverse problem dealt with in this thesis, and the application of simulated annealing in solving it will be discussed in chapter 4.

References:

[1]. Analytic inversion of remote sensing data - C.D. Capps, R.L. Henning, and G.M. Hess, pp 3581, Oct. 1982, Vol. 21, No. 19, Applied Optics.

[2]. Analysis of Numerical Methods - Eugene Isaacson and Herbert Bishop Keller - John Wiley & Sons, Inc., New York, Library of Congress Catalog Number: 66-17630.

[3]. Optimization by simulated annealing - S. Kirpatrick, C.D. Gelatt, Jr., M.P. Vecchi, pp. 671, 13th May 1983, Science, Vol. 220, Number 4598.

[4]. Numerical recovery of the layered medium from surface data - Master's thesis of P. Li under the supervision of A. G. Ramm, Department of Mathematics, Kansas State University.

[5]. Inverse scattering for geophysical problems - A. G. Ramm, Phys. Lett., 99A (1983) pp 201-204.

CHAPTER III

STOCHASTIC OPTIMIZATION AND SIMULATED ANNEALING

Stochastic optimization refers to the process of optimization of a set of parameters for a system through random guesses of the parameter values, and choosing the case which produces the best result out of all the different guesses made. This is the crude form of stochastic optimization, as here, the different cases bear no correlation to one another, resulting in independent sets of random parameter values for the different cases, and hence there is no consistent improvement as more and more trials are implemented. This is, therefore, an extremely inefficient, but conceptually very simple way of attempting to achieve an optimized set of system parameters. To illustrate this process, let us assume that the system to be optimized is dependent on a set of 'n' parameters $s_1, s_2, ..., s_n$ which take on values in a space $\Omega \subset R^n$ or C^n, where R is the real space and C is the complex space. Let us define a real valued system function $F(s_1, s_2, ..., s_n)$ that depends on these 'n' system parameters. The objective is to find the set of optimum parameter values s_i^* (for i = 1 to n), in Ω such that $F(s_1^*, s_2^*, ..., s_n^*)$ takes the minimum value. In the case of the crude stochastic optimization described above, the task during each trial would be to choose a random set of the s_i values such that $(s_1, s_2, ..., s_n) \in \Omega$, and then compute the corresponding value of the function F. The choices of these n-tuples are independent in each trial and are purely random. This is the concept behind the method of the basic stochastic optimization.

An improvement over the crude and simple stochastic optimization process described above is to have a stochastic procedure where the different trials are not independent, but every trial is based on the state of the system produced by the immediately preceding trial, resulting in a set of system parameters $(s_1, s_2, ..., s_n)$, which move consistently towards an optimized state $(s_1^*, s_2^*, ..., s_n^*)$. Here the set of parameters during the first trial are chosen randomly from Ω just as in the method of the first paragraph, but the parameters for the subsequent trials are obtained by changing the previous set of parameters by small randomly chosen amounts, such that the maximum absolute value of $(s_i^{n+1} - s_i^n)$ is bounded above by a real number **s** called the **step size**. The maximum is taken over all possible i where s_i^{n+1} and s_i^n denote the values of the ith component of the system-parameter vector $(s_1, s_2, ..., s_n)$ during two successive trials with the higher valued superscript representing the later trial. Thus we find that the parameters during the different trials are correlated. Here, the system function F is computed after every update of the n-dimensional parameter vector, and the updated parameters are accepted as the new improved parameters if the new value of $F(s_1, s_2, ..., s_n)$ is less than or equal to its previous value. Otherwise, the new parameter vector is rejected, and the old parameter vector is retained. The process is repeated over and over again, and thus we get consistently improved parameter vectors during the trials. This is a marked difference of this algorithm with the one described in the first paragraph, as the trials there were independent, and hence there was no consistent improvement. But this improved version also has a problem associated with it. As mentioned above, this algorithm accepts only those parameter vectors that lower the system function, and rejects all those updates that increase it. As a result, once the parameters are in the neighborhood of any local minimum of F, there is a good chance that F will get trapped in it when the step size chosen is small, because in

that case almost every update of the parameter vector will take the system function into a region near the local minimum, and hence all those function values would be higher than the existing value, thus all those updates would be rejected. This will result in a solution that will produce a local minimum and not the globally optimized state. One could take a larger step size to come out of a local minimum, but in that case the system will be changed considerably in each step, and hence it will have a lower chance of finding a minimum. To improve upon this problem, we can modify the scheme of accepting the updated parameters, so that the system can come out of a local minimum. This method is called "Simulated Annealing", and it will be described next.

Simulated annealing is a process which is based on the concept of "annealing" as used in the metallurgical processes. Metallurgical annealing is the process of slowly cooling a physical system in order to obtain states with globally minimum energy [2]. In this process, a sample is first heated up sufficiently so as to melt it out completely, and then slowly cooled back to a highly ordered state. This is a statistical mechanical phenomenon as explained below.

A system composed of a very large number of particles (atoms) may be described in terms of the average behavior of the particles constituting the system. In statistical mechanics, to describe the average behavior, each configuration of a system is defined by the set of positions r of the particles weighted by a probability factor of the form

$$e^{\frac{-E(r)}{kT}}$$

where k is called the "Boltzmann constant", E is the energy of the system, and T is the temperature. One property of such a system that is of interest is the

behavior at low temperatures. Usually, the system approaches its ground state as the temperature is lowered. But in practice, low temperature alone is not a sufficient condition for attaining the ground state of the system. Experiments show that the final state of a system at low temperature depends on the rate at which the material is cooled from its original molten state. If the material is cooled at a very slow rate, taking care to keep the material at equilibrium at every stage of the process, then the final state attains an energy which is much lower in energy scale than the configuration attained by a rapid quenching of the substance. In the first case, we often achieve a crystalline solid with the set of particles distributed in a regular pattern, whereas in the second case, we achieve a glassy state with a lot of defects embedded in it. Metallurgical annealing refers to this process of slow cooling of an initially molten substance to take it to a highly ordered state.

This concept of annealing can be applied to the optimization problem mentioned before in this chapter, where we attempt to minimize a system function F that depends on the n-dimensional parameter vectors $F(s_1, s_2, ..., s_n) \in \Omega$. Here the components of the parameter vector will act as the coordinates of the constituting particles of a metallurgical substance, and the starting random parameters are analogous to the random structure of the initially molten metallurgical substance. In the case of simulated annealing, we define a parameter called the effective temperature, which plays the role of temperature **T** in actual annealing. It plays an important role in deciding the acceptance criterion of the updated parameters during the optimization. The updating scheme is similar to the one in the second paragraph of this chapter, for the improved version of the crude stochastic optimization procedure. After an update of the parameter-vector has been made, the corresponding system function $F(s_1, s_2, ..., s_n)$ is computed,

and compared to the previous value. If the new value is less than the old value, then the new parameter vector is accepted as the new parameter-vector unconditionally. If the new value is more than the old value, then the difference between the new and the old values is computed as

$$dF = (F_{new} - F_{old})$$

and the updated parameter is accepted with a probability of

$$p(dF) = exp(\frac{-dF}{kT})$$

This can be simulated on a computer by choosing a random number from a uniformly distributed pseudo-random number generator such that we have equal probability of getting numbers in the interval **(0, 1)**, and then comparing this number with the computed probability of **p(dF)**. If this random number is less than **p(dF)**, the new configuration is accepted, else it is rejected, and the system stays at its old state. We then start over and repeat the procedure.

The primary advantage of this method of optimization is that it has a higher probability of reaching the global minimum than the improved stochastic optimization described in the second paragraph, because simulated annealing does have a chance of coming out of any local minima. When a move increases the function value, the algorithm generates a **finite non-zero** probability of accepting this move, giving the system a chance to get into a 'higher energy' state, which helps the system to come out of a local minimum. This is a marked difference of this algorithm from the algorithms that look for moves that only lower the function value, which can very easily get trapped in any local minimum, as we saw in second paragraph. Simulated annealing, thus, accepts moves which (with certain

probability) even drive the system away from a minimum parameter configuration whereas the scheme in the second paragraph always tries to drive the system towards a strictly lower energy configuration.

Vanderbilt and Louis (1984) compared the efficiency of Simulated Annealing with other conventional non-probabilistic optimization techniques [1]. They found that Simulated Annealing is comparable to these approaches as far as the number of function evaluations and computer time is concerned. But, in case of simple low dimensional optimization problems, almost any of the non-probabilistic approaches has a higher chance of finding the global minimum than the method of Simulated Annealing. Specially for problems with clearly convex cost functions, it is always more efficient to use standard gradient descent methods. But, Simulated Annealing can be very advantageous for optimization problems having very many degrees of freedom, where exhaustive search for a global minimum through deterministic procedures can be impossible or computationally prohibitive.

References:

[1]. Optimization Algorithms: Simulated Annealing and Neural Network - W. Jefferey and R. Rosner, The Astrophysical Journal, **310**, 473-481, 1986 November 1.

[2]. A Tutorial Survey of Theory and Applications of Simulated Annealing - Bruce Hajak, Proceedings of 24th Conference on Decision and Control, December 1985.

[3]. D. Vanderbilt and S. G. Louis, 1984, J. Comp. Phys, 56, pp 259.

CHAPTER IV

LEAST SQUARE OPTIMIZATION
BY
SIMULATED ANNEALING
APPLICATION TO THE INVERSE PROBLEM

In this chapter, we will discuss the application of simulated annealing to least square optimization, with a particular application to the "Inverse Problem" as has been discussed in Chapter II. We have seen in Chapter III that simulated annealing is effectively a method for mutidimensional global optimization, and hence for applying this method to the inverse problem, we have to first frame it (the inverse problem) into an optimization problem. It was discussed in Chapter II how the inverse problem is equivalent to a least square optimization problem. We will discuss now the method adopted here for applying simulated annealing to this case, along with the other methods applied to similar problems.

There are analytic ways of solving least square problems when the number of dimensions is small, but the number of computations grows exponentially with the number of dimensions, making it computationally prohibitive for many dimensions [1]. Besides, the analytic method of solving the least square minimization problem involves inversion of the well known Hilbert matrix, which is a well-posed but ill-conditioned problem for higher dimensions [2]. Hence it is not a very practical method for problems involving a large number of parameters.

Attempts have been made to solve the inverse problem described in Chapter II

by deterministic methods, and one such method has been developed by P. Li and Ramm [1]. The problem discussed in their work is to recover underground layers from surface data. Their method is iterative, and the iteration should be started with some preferred starting point to reduce the chances of attaining a local minimum.

The stochastic optimization algorithm studied here is that of Simulated annealing, and the particular inverse problem of interest here has been explained before. Our objective is to recover underground segments from surface data. For this purpose, a signal is sent below the ground from a transmitter on the ground, and the reflected signal is recorded by receivers on the ground. The data are obtained from these reflected signals. As we have already mentioned (in Chapter II), the data $g(a,b)$ are given by equation (2.5), and hence the integral

$$\int_{box_i} \frac{1}{|\bar{z}-a|\cdot|\bar{z}-b|} \tag{4.1}$$

is computed first for all the boxes, i.e., for $i = 1$ to 25, and stored for later use in the computation of $g(a,b)$ and $\tilde{g}_i$ (see Chapter II). This integral is computed by three dimensional Simpson's integration algorithm for its relative simplicity. In the simulation of the actual seismic data (as has been discussed in Chapter II), we assume a particular weight distribution among the different boxes in the segmented Ω which is the three dimensional region of the Earth's crust considered for this problem, and the values of $g(a,b)$ computed through equation (2.5) for different values of a and b which are the locations of the transmitter and the receiver respectively on the surface of the Earth. These are then taken as the actual seismic data for the configuration assumed in our simulation of the experiment. We then

try to optimize the "cost function" C as expressed by equation (2.3). The simulated annealing algorithm for this optimization is described below.

In the present case, we have computed $g(a,b)$, as described above , for 11 different values of a and 11 different values of b for each of the values of a. These 11 locations of the receiver and the transmitter correspond to the points on the surface of the block uniformly spaced at an interval of 1 unit, starting at the left hand corner and going up to the right hand corner. To start with, we assume that the actual distribution of the weight is unknown, and hence we assign random coefficients $w \in (0,2)$ in each of the boxes. This range of 0 to 2 for the initial values of coefficients is chosen because the maximum coefficient chosen in simulating the actual values of $g(a,b)$ is 1, and we tried to test the algorithm with a starting configuration which does not assign a coefficient which is very different from the actual value. We will see in the next Chapter how the algorithm behaves when this condition is not satisfied. Once the starting coefficients are assigned randomly to each of the boxes in Ω, we compute C by the equation

$$C = \sum_{i=1}^{121} (g_i - \tilde{g}_i)^2 \tag{4.2}$$

where g_i refers to the value of $g(a,b)$ for the ith data out of the 121 total simulated seismic data, and $\tilde{g}_i$ refers to the value of $\tilde{g}$ for the same locations of the receiver and the transmitter. The next step in the algorithm is to add (or subtract) a certain amount from the existing value of the weight of a randomly chosen box. Actually, this amount that is added (or subtracted) has an upper bound of 0.1, which we call **the step size** signifying the fact that in a single step, we cannot change the value of the weight of any box by more than this value. After this change in coeffficient is

made in a box, the new value of C is computed and compared to the previous value. If the new value is less than or equal to the old value, then the updated weight is accepted for the corresponding box, otherwise it is accepted with a Boltzman probability distribution. For this purpose, the difference between the new and the old cost functions ΔC is computed through the following relation:

$$\Delta C = (C_{new} - C_{old}) \tag{4.3}$$

and this difference is positive when the new C is more than the old C. This update is then accepted with a probability of

$$p(\Delta C) = e^{\frac{-\Delta C}{T}} \tag{4.4}$$

where T is the effective **temperature** at that stage of the algorithm. The update is, therefore, accepted if a number chosen from a pseudo-random number generator (which generates nearly uniformly distributed numbers) is less than or equal to the above probability. This is, in brief, the acceptance criterion of the updates for this simulated annealing algorithm. The same procedure is repeated a large number of times at each temperature before the temperature is reduced, so that the configuration gets a chance to equlibrate at any temperature, which is important in the case of annealing, as has been said before. At each temperature, the fraction called **equilibrium ratio**, of the moves that increase the cost function out of the total number of accepted moves from a group of 250 total moves is computed, and if this fraction is close to 0.5, it is assumed that the configuration has attained equilibrium at that temperature, because there are as many accepted moves that increase the cost function, as there are moves that reduce it. So the configuration, on the average, would stay the same even if more and more updates are attempted

at that temperature. This is the point when the temperature is reduced by a factor called the **Cooling Rate**. The temperature is also reduced when we have already made 4000 moves, and the equilibrium ratio is still away from 0.5. But, if the above fraction is away from 0.5, and the total number of moves made at that temperature is less than 4000, then the configuration has not reached equilibrium, and we attempt more updates in groups of 250 moves, with the **equilibrium ratio** computed after each group of 250 moves, to determine if the temperature is to be reduced or not. We chose **equilibrium ratio** to be computed after 250 moves because, there are 25 boxes in total, and hence on the average each box will have a chance of getting updated 10 times in a block of 250 updates. In practice, the temperature is reduced when the value of **equlibrium ratio** is within a certain range, called the **ratio-tolerance**, of 0.5, i.e., the temperature would be reduced if

$$\mid equilibrium\ ratio - 0.5 \mid = ratio - tolerance. \tag{4.5}$$

The above procedure is repeated at each temperature. The program is terminated when the fraction of the total number of moves that are accepted at a temperature falls below a preassigned value called the **acceptance-tolerance**, which is a small number (it is 0.01 for our case), so that the program would stop when most of the updates are not being accepted any more, and the system has nearly reached the frozen state.

This is the algorithm of the annealing algorithm applied to the inverse problem described in Chapter II.

References:

[1]. Numerical recovery of the layered medium from surface data - Master's thesis of P. Li under the supervision of A. G. Ramm, Department of Mathematics, Kansas State University.

CHAPTER V

RESULTS AND CONCLUSION

In this chapter, we will present the results produced by the application of the simulated annealing algorithm described in Chapter 4 on the inverse problem. Initially, in all the simulations, it is assumed that the actual distribution of the weight in the slab of the Earth's crust described before consists of indices of unity in the boxes which are in the 4th row - second column, and 4th row-4th column, and zero everywhere else. This configuration is shown below:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

The data $\{g_i\}_{i=1}^{121}$ are generated for 121 different combinations of the receiver and the transmitter locations on the surface of the Earth, corresponding to 11 different locations of the transmitter and the receiver, and the algorithm is tested on the basis of how closely it can reproduce this configuration starting with a random weight distribution. The reader should be warned that the fits will be poor, as discussed in the Introduction. We are studying the effects of several parametes on this algorithm, and not trying to find the best algorithm.

First of all, the algorithm was tested to see the effect of round-off error on the final results. These round-off errors are produced because of the finite precision of the computers, and a detailed theoretical treatment of this particular topic can be found in any numerical analysis book such as [1]. For our algorithm, we set the **step size** at 0.1, **starting temperature** at 5, **cooling rate** at 0.9, **number of iterations per temperature** at 1000, and **acceptance-tolerance** at 0.1. These values were chosen because a step size of 0.1 is not a very large step size as compared

to the actual maximum weight in a box, neither is it a very small value compared to the same. A starting temperature of 5 indicated that initially almost 98 percent of the total number of attempted moves were being accepted indicating that the configuration at that temperature was analogous to the molten state in an actual annealing procedure. A cooling rate of 0.9 was adopted because it was used in the simulated annealing algorithm designed by S. Kirpatrick, C. D. Gelatt, Jr., and M. P. Vecchi [2], and they found it to be a reasonably slow cooling rate for the annealing algorithm. The acceptance-tolerance was chosen to be 0.1 because it was found that below this value, the execution time goes up considerably without any significant reduction in the final cost function. 1000 iterations at each temperature would give every box the chance to be updated 40 times on the average. With these parameter values, the algorithm was checked in both single and double precision modes.

In the single precision case, it was found that the algorithm produces weight distributions, whose deviations from the actual distribution did not follow a statistical behavior, as it was found that when squared errors C (the cost function for this problem) from different trials were grouped together and the group avergage computed, it did not go down as the square root of the number of groups considered, which would be expected of any statistical phenomenon. In fact it was roughly constant. This suggests the presence of a noise term. The approximate statistical behavior, however, was observed in the values of C obtained in the double precision mode. We also found that the average value of C produced by the single precision execution of the annealing algorithm was about 1.5 times higher than the same obtained by the double precision execution of the same algorithm. This strongly hinted at the significant worsening of the algorithm in the single precision mode because of round-off error affecting the computation of the ratio of

$\frac{\Delta C}{T}$ specially when the temperature gets small because then a small round-off in T will be magnified many times as this quantity appears in the denominator. The result obtained by the execution of the algorithm in the single precision is, thus, meaningless, as what we observe there is just "noise" because of round-off and not the desired quantity. This suggested that the algorithm does not work in single precision (on the VAX) and must be executed in the double precision mode for any further studies. The results obtained in the single and the double precision modes are shown in Tables 1, 2, and 3.

In all the trials above, the number of updates at each temperature was fixed at 1000, as each of the 25 boxes would, on the average, have a chance to get updated 40 times at each temperature which was assumed to be sufficient to let the configuration equilibrate at each temperature. Here, it would be relevant to briefly explain the criterion for deciding if equilibrium had been reached at a temperature. We say that equilibrium has been attained if about 50 percent of the total number of accepted moves at a temperature increases the energy, and the rest decrease it (or keep it the same). This makes sense because at this stage, even if we keep the temperature the same, the configuration will not be improved on the average, because for every move that decreases the cost function, there will be a move that will increase it. We computed the ratio called the **equilibrium ratio** (as defined earlier) of the number of moves decreasing the cost function to the total number of moves accepted at each temperature at the end of 1000 attempted updates, and it was found that this **equilibrium ratio** was much above 50 percent at some of the temperatures, giving us the indication that 1000 iterations per temperature is not always sufficient to attain equilibrium. At some of the temperatures, it was found that as high as 70 percent of the total number of accepted moves were decreasing the cost function when the temperature was lowered, and this was a premature cooling because we

TABLE I

Results showing the non-statistical behavior of the standard error for double precision execution of the annealing algorithm for step size of 0.1

Number of groups	Standard Error σ	Ratio of σ
1	0	
2	7.145E-07	
3	1.127E-06	1.57
4	1.026E-06	0.91
5	1.070E-06	1.04
6	1.084E-06	1.01
7	1.386E-06	1.27
8	1.412E-06	1.02
9	1.332E-06	0.94
10	1.279E-06	0.96
11	1.221E-06	0.95
12	1.351E-06	1.11

TABLE II

Results showing the statistical behavior of the standard error for double precision execution of the annealing algorithm for step size of 0.1

Number of groups	Standard Error σ	Ratio of σ
1	0	
2	1.241E-06	
3	1.459E-06	1.17
4	1.435E-06	0.96
5	1.301E-06	0.90
6	1.273E-06	.97
7	1.192E-06	0.93
8	1.123E-06	0.94
9	1.075E-06	0.95

TABLE III

Results showing the statistical behavior of the standard error for double precision execution of the annealing algorithm
Step size used = 0.05

Number of groups	Standard Error σ	Ratio of σ
1	0	
2	8.788E-07	
3	8.180E-07	0.93
4	7.423E-07	0.90
5	9.929E-07	1.34
6	9.348E-07	0.94
7	9.545E-07	1.02
8	9.096E-07	0.95
9	9.066E-07	0.99
10	8.632E-07	0.95
11	8.355E-07	0.96
12	8.168E-07	0.97

could accept more moves that would lower the cost function on the average even if we stayed at that temperature. **On this basis, it was decided to change the cooling schedule so as to allow the configuration at each temperature to come closer to an equilibrium-ratio of 0.5 before being cooled.** Here the updates were implemented in groups of 250 updates. The new cooling schedule is as follows for each temperature during the annealing process:

1. Initialize the variable **ngroup** to 0. This variable keeps track of the number of groups attempted

2. **ngroup** = **ngroup** + **1**.

3. Make a group of 250 updates.

4. **If** this is the first group of updates at the existing temperature, **then** go to step 2 (this allows the configuration to be closer to equilibrium through the first 250 updates). **If** this is not the first group, **then** compute the **equilibrium ratio**.

5. **If** the total number of updates accepted so far at this temperature is less than 1 percent of the total number of attempted updates, or no moves have been accepted from the last group of updates, then stop the algorithm (as the configuration is assumed to be frozen at this point). **Else if** the **equilibrium ratio** lies between 0.47 and 0.53 (it is assumed that the configuration is very close to equilibrium if the **equilibrium ratio** is within these limits) then reduce the temperature (as the configuration is in equlibrium). **Otherwise** go to step 6.

6. **If** the number of groups attempted so far at the current temperature is less than 16, then go to step 2. **Else** reduce the temperature. (This fixes an upper bound for the number of updates at any temperature at 4000).

So, we observe from the above cooling schedule that the number of total updates

can be different at different temperatures depending on how fast the configuration equilibrates at each temperature. The results obtained by executing the annealing algorithm with this modified cooling schedule showed that the **equilibrium ratio** was within the above mentioned limits at almost each temperature and for some of the temperatures, this limit was reached after only about 500 updates, and therefore, 1000 updates, as before, would be unnecessary at these temperatures. This shows that this schedule automatically determines the number of required updates at each temperaure, thus making it more appropriate for our purpose. All the results that follow are based on this modified cooling schedule.

We see above that the configuration at a temperature is assumed to have attained equilibrium if the **equilibrium ratio** is between 0.47 and 0.53. In fact, we use a parameter called **ratio-tolerance** (see page 24) that determines how close to 0.5 the **equilibrium ratio** should get before the configuration can be assumed to be in equilibrium, and equilibrium is assumed if the **equilibrium ratio** is within **(0.5 - ratio-tolerance)** and **(0.5 + ratio-tolerance)**. So, in the above schedule, **ratio-tolerance** is 0.03. Better results are expected for lower values of this **ratio-tolerance** as that would mean a better approximation to equilibrium, and the results are studied for a lower and a higher values of this parameter, set at 0.015 and 0.05 respectively. The results for these three different values of **ratio-tolerances** are shown below:

Ratio-tolerance	Cost Function	Standard Error	Execution Time
0.015	4.6E-06	8.5E-07	45 min.
0.03	5.8E-06	7.1E-07	17 min.
0.05	7.6E-06	2.2E-07	13 min.

We comment on these results below.

The final weight distribution recovered by this algorithm for these different values of **ratio-tolerances** are shown below. The results here are averages of 30 runs for each value of the **ratio-tolerance**.

Ratio-tolerance = 0.015

0	0	0	0	0
0	0	0	0	0
0.01 ± 0.01	0.04 ± 0.03	0.07 ± 0.04	0.03 ± 0.03	0.01 ± 0.02
0.09 ± 0.08	0.14 ± 0.11	0.18 ± 0.11	0.15 ± 0.13	0.07 ± 0.07
0.29 ± 0.17	0.29 ± 0.26	0.49 ± 0.30	0.38 ± 0.24	0.30 ± 0.21

Ratio-tolerance = 0.03

0	0	0	0	0
0	0	0.01 ± 0.01	0	0
0.01 ± 0.01	0.04 ± 0.04	0.05 ± 0.03	0.03 ± 0.02	0.01 ± 0.02
0.11 ± 0.1	0.14 ± 0.11	0.16 ± 0.13	0.12 ± 0.11	0.08 ± 0.04
0.23 ± 0.13	0.35 ± 0.26	0.48 ± 0.28	0.42 ± 0.29	0.30 ± 0.19

Ratio-tolerance = 0.05

0	0	0	0	0
0	0	0.01	0	0
0.01 ± 0.02	0.03 ± 0.03	0.06 ± 0.05	0.04 ± 0.03	0.03 ± 0.03
0.1 ± 0.08	0.14 ± 0.11	0.13 ± 0.10	0.14 ± 0.10	0.09 ± 0.07
0.30 ± 0.20	0.38 ± 0.24	0.45 ± 0.29	0.36 ± 0.20	0.28 ± 0.19

Comparing the execution times obtained for these different values of ratio-tolerances, we find that the execution time goes up from 13 minutes to about 17 minutes when the ratio-tolerance goes down from 0.05 to 0.03. But the ratio-tolerance of 0.05 produces a cost functionwhich is about 13 times the cost function for a ratio-tolerance of 0.03. Decreasing the ratio-tolerance from 0.03 to 0.015 reduces the cost function by half whereas the execution time goes up from 17 minutes to 45 minutes, which is a considerable increase in the execution time. It should also be noted from the data presented above that the average final cost functions corresponding to **ratio-tolerances** of 0.015 and 0.03 overlap and hence they are statistically indistinguishable, though the smaller **ratio-tolerance** has an execution time which is more than two times that for the bigger **ratio-tolerance**. This suggests that given a choice between the values of 0.015 and 0.03, one should use the larger of these values. But we also observe that the cost functions corresponding to the **ratio-tolerances** of 0.03 and 0.05 do not overlap indicating the superiority of 0.03 as compared to 0.05. We, therefore, decided that a value of 0.03 would be a sensible trade-off between execution time and ratio-tolerance. The subsequent runs are all for a ratio-tolerance of 0.03. We see here that **this parameter is quite important because if chosen inappropriately, it can increase the execution time by a great amount without producing any significantly improved results.**

The present annealing algorithm has been compared with the zero temperature case in which only those moves are accepted which lower the C, and we find that the results from the zero temperature trials are inferior to those of the annealing algorithm both qualitatively and quantitatively. Quantitatively, we find that **the average cost function is about 10 times greater for the case of zero temperature, than that for the annealing algorithm.** The average cost

function in the case of zero temperature trials is of the order of 1.0E-05 whereas that for the annealing algorithm is found to be about 10 times smaller than this. **Qualitatively, we find that the zero temperature case produces an almost uniform average weight distribution among the columns** (as can be seen from the average configuration recovered by the zero temperature case presented below), **whereas the annealing algorithm can eliminate the two bordering columns which is a definite qualitative improvement in the recovery of the configuration.** But there is another aspect of the zero temperature case that is to be noted here, and that is, **the execution time for the annealing algorithm is about ten times more than that for the zero temperature case.** The zero temperature case takes about **2 minutes** to finish execution whereas the annealing algorithm takes about **20 minutes**. Hence, in a given amount of time, one can execute the zero temperature case many more times than the annealing algorithm. The average configuration obtained by executing the zero temperature algorithm 30 times is shown below:

Results obtained by the zero temperature case:

Average of 30 trials:

0	0	0	0	0
0 ± 0.01	0.1 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01
0.04 ± 0.03	0.05 ± 0.03	0.05 ± 0.03	0.05 ± 0.03	0.05 ± 0.03
0.13 ± 0.05	0.12 ± 0.03	0.11 ± 0.03	0.12 ± 0.05	0.14 ± 0.04
0.27 ± 0.07	0.26 ± 0.06	0.25 ± 0.05	0.25 ± 0.06	0.27 ± 0.06

The configuration corresponding to the lowest cost function is shown on the next page.

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0.04	0.06	0	0.02	0.04
0.02	0.04	0.04	0	0.07
0.04	0.17	0.11	0.12	0.2
0.17	0.22	0.23	0.21	0.14

Next, the effects of different step sizes and different starting configurations are studied for this annealing algorithm. The step size is studied first. For this purpose, the other parameters are fixed at the following values:

Starting temperature = 5

Cooling rate = 0.9

Ratio-tolerance = 0.03

Acceptance-tolerance = 0.01

Number of updates per block = 250

Maximum number of updates at a temperature = 4000

With the above set of parameters, the step sizes are taken as 0.1, 0.05, 0.02, 0.01, and 0.005. For each of these step sizes, the annealing algorithm is executed 30 times, and the **standard error** among the cost functions in the final configuration output by the annealing algorithm for each of these 30 trials is computed by grouping the cost functions into groups containing 5 error elements, to get a total of 6 groups to compute the **standard error** from. The cost function for each step size along with the **standard error** in these cost functions are shown below:

Step size	Cost Function	Standard Error
0.005	6.0E-06	3.7E-07
0.01	5.6E-06	6.3E-07
0.02	4.5E-06	7.2E-07
0.05	4.3E-06	6.7E-07
0.10	5.8E-06	7.1E-07

One would guess that a smaller step size would produce a better recovery, and hence yield a lower cost function, but the data above do not clearly show that. This could be due to the fact that, as the step size gets smaller, the value of ΔC (as described in Chapters before) also becomes smaller and hence gets affected more by the round-off error arising because of the finite precision of the machine. The weight distribution recovered by this algorithm for each of these step sizes are shown below:

Step size =0.05

0	0	0	0	0
0	0	0.01 ± 0.01	0	0
0.01 ± 0.02	0.05 ± 0.03	0.06 ± 0.04	0.04 ± 0.04	0.01 ± 0.02
0.10 ± 0.09	0.14 ± 0.10	0.18 ± 0.10	0.13 ± 0.08	0.09 ± 0.07
0.41 ± 0.24	0.27 ± 0.20	0.39 ± 0.18	0.31 ± 0.20	0.36 ± 0.36

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0	0	0
0	0.07	0.07	0.02	0
0.11	0.07	0.32	0.03	0.03
0.22	0.37	0.41	0.55	0.39

Step size = 0.02

0	0	0	0	0
0	0	0.02 ± 0.01	0	0
0	0.02 ± 0.02	0.07 ± 0.05	0.07 ± 0.04	0.02 ± 0.02
0.04 ± 0.06	0.05 ± 0.04	0.11 ± 0.07	0.13 ± 0.09	0.08 ± 0.08
0.64 ± 0.19	0.65 ± 0.27	0.20 ± 0.13	0.20 ± 0.12	0.40 ± 0.20

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0.0	0	0	0
0	0.04	0.15	0.05	0.02
0	0	0.31	0.07	0.19
0.53	0.83	0.04	0.02	0

Step size = 0.01

0	0	0	0	0
0	0	0.02	0	0
0	0.01 ± 0.01	0.06 ± 0.02	0.06 ± 0.03	0.02 ± 0.02
0.03 ± 0.05	0.03 ± 0.02	0.09 ± 0.04	0.12 ± 0.07	0.06 ± 0.04
0.74 ± 0.13	0.76 ± 0.19	0.13 ± 0.06	0.14 ± 0.05	0.56 ± 0.15

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0.01	0	0
0	0.01	0.1	0.05	0
0	0.02	0.05	0.34	0.04
0.76	0.75	0.12	0.11	0.41

Step size = 0.005

0	0	0	0	0
0	0	0.03	0	0
0	0	0.06	0.05 ± 0.01	0.03 ± 0.02
0.03 ± 0.04	0.02 ± 0.01	0.09 ± 0.02	0.11 ± 0.04	0.05 ± 0.03
0.72 ± 0.09	0.83 ± 0.08	0.15 ± 0.02	0.17 ± 0.03	0.53 ± 0.09

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0.02	0	0
0	0	0.1	0.05	0.01
0.07	0	0.09	0.08	0.05
0.55	0.9	0.19	0.21	0.54

Looking at the cost functions for the different step sizes, we observe that it is almost impossible to prefer a step size based on these results because the cost functions for these step sizes overlap, indicating that one cannot be shown to be clearly superior to another. But we have chosen 0.05 to be the step size for this algorithm considering the trade off between execution time and final configuration. Looking at the weight distribution recovered by the algorithm for the different step sizes shown above, we find that we get a qualitatively improved result when the step size is smaller, as is shown by the lower and lower weights for top layers (where the actual weights are zeroes) for smaller and smaller step sizes.

We also plotted the graph of acceptance vs. temperature for all of these step sizes. These graphs are produced on the next page. It is found that as the step size becomes smaller, the acceptance values show a more pronounced deviation from their monotonically decreasing behavior by showing an increase in acceptance at intermediate temperature range before decreasing monotonically again, as can be

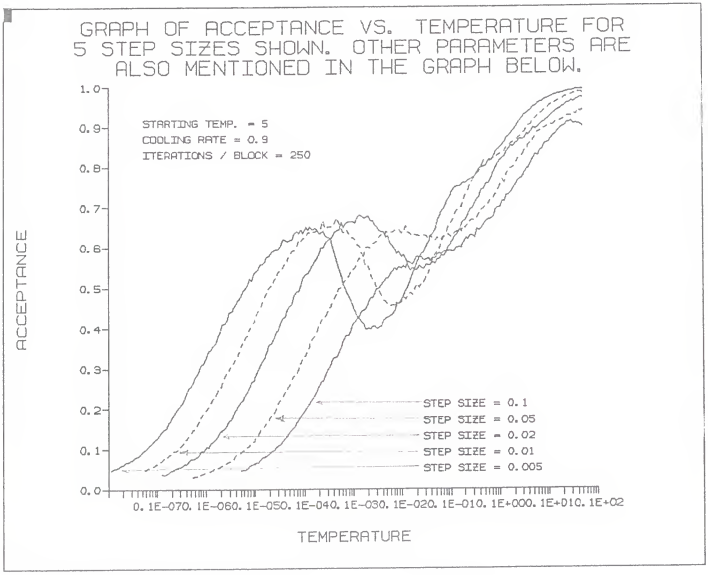
GRAPH OF ACCEPTANCE VS. TEMPERATURE FOR 5 STEP SIZES SHOWN. OTHER PARAMETERS ARE ALSO MENTIONED IN THE GRAPH BELOW.
STARTING TEMP. = 5
COOLING RATE = 0.9
ITERATIONS / BLOCK = 250
STEP SIZE = 0.1
STEP SIZE = 0.05
STEP SIZE = 0.02
STEP SIZE = 0.01
STEP SIZE = 0.005
ACCEPTANCE
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0. 1E-07 0. 1E-06 0. 1E-05 0. 1E-04 0. 1E-03 0. 1E-02 0. 1E-01 0. 1E+00 0. 1E+01 0. 1E+02
TEMPERATURE

Fig - 2

seen in the graphs for step sizes of 0.005, 0.01, and 0.02. This behavior may occur beacuse the average change in the cost function (ΔC) computed at each temperature does not decrease at the same rate as the temerature.

It was then hypothesized that similar acceptance characteristic could be observed if the step size is fixed at some value and the initial starting configuration is varied over a range, because starting with a small step size is quite equivalent to having a larger step size but also a larger deviation from the actual configuration. This would show the correlation between step size and the starting configuration. It was decided to start with a uniform starting configuration with the same value in each of the boxes. The results are shown for a step size of 0.05 (fixed) and the different starting configurations as shown below.

Weight per box	Cost Function	Standard error
0.00	4.002E-06	3.520E-07
0.04	4.331E-06	5.893E-07
0.08	3.792E-06	6.906E-07
0.8	4.897E-06	5.239E-07
1.6	4.087E-06	3.433E-07
4.0	4.571E-06	1.082E-06
8.0	4.745E-06	1.364E-06

As can be seen from the above list of cost functions, the standard error becomes comparable to the data when the weight per box is 4.0 or above (which is a total starting value of 50 times (or more) the actual total value). This means that the present algorithm does not produce reliable results when we start with a total amount of the weight which is more than 50 times the actual total value present inside the volume. It was also found that the algorithm almost never required more than 4000 (the preset upper bound) updates before equilibrating. At most of the

temperatures, it equilibrated well below 400 updates.

The average weight distribution recovered by this algorithm for the different uniform starting configurations mentioned above are shown below:

Initial total weight of 0.00 distributed uniformly among the 25 boxes:

0	0	0	0	0
0	0	0	0	0
0 ± 0.01	0.05 ± 0.03	0.06 ± 0.04	0.05 ± 0.04	0.01 ± 0.01
0.11 ± 0.08	0.16 ± 0.11	0.16 ± 0.11	0.20 ± 0.11	0.10 ± 0.09
0.31 ± 0.19	0.35 ± 0.27	0.34 ± 0.20	0.32 ± 0.18	0.26 ± 0.20

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0	0	0
0.01	0.06	0.09	0.04	0
0.03	0.07	0.23	0.13	0.11
0.4	0.26	0.67	0.36	0.13

Initial total weight of 2.00 distributed uniformly among the 25 boxes:

0	0	0	0	0
0	0	0.01	0	0
0.02 ± 0.02	0.05 ± 0.03	0.06 ± 0.04	0.03 ± 0.03	0.02 ± 0.03
0.09 ± 0.08	0.15 ± 0.09	0.19 ± 0.10	0.17 ± 0.11	0.10 ± 0.09
0.28 ± 0.18	0.35 ± 0.20	0.36 ± 0.23	0.36 ± 0.23	0.25 ± 0.17

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0	0	0
0.04	0.01	0.12	0.02	0
0.03	0.15	0.18	0.14	0.07
0.19	0.48	0.25	0.72	0.18

Initial total weight of 20.00 distributed uniformly among the 25 boxes:

0	0	0	0	0
0	0	0.01 ± 0.01	0	0
0.02 ± 0.03	0.04 ± 0.03	0.06 ± 0.05	0.05 ± 0.05	0.01 ± 0.02
0.08 ± 0.08	0.14 ± 0.10	0.18 ± 0.13	0.12 ± 0.10	0.15 ± 0.11
0.38 ± 0.25	0.35 ± 0.23	0.36 ± 0.23	0.29 ± 0.19	0.28 ± 0.21

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0	0	0
0.03	0.01	0.15	0.06	0
0.11	0.05	0.23	0.1	0
0.18	0.61	0.13	0.55	0.37

Initial total weight of 40.00 distributed uniformly among the 25 boxes:

0	0	0	0	0
0	0	0.01	0	0
0.02 ± 0.02	0.03 ± 0.03	0.07 ± 0.07	0.04 ± 0.03	0.01 ± 0.02
0.10 ± 0.08	0.13 ± 0.10	0.16 ± 0.11	0.17 ± 0.13	0.10 ± 0.08
0.35 ± 0.13	0.35 ± 0.18	0.42 ± 0.20	0.27 ± 0.20	0.32 ± 0.23

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0	0	0
0.05	0.02	0.09	0.08	0.02
0	0.06	0.25	0.15	0.09
0.27	0.61	0.53	0.26	0.06

Initial total weight of 100.00 distributed uniformly among the 25 boxes:

0	0	0	0	0
0	0	0.01 ± 0.01	0	0
0.02 ± 0.03	0.04 ± 0.03	0.07 ± 0.05	0.05 ± 0.04	0.02 ± 0.02
0.07 ± 0.09	0.12 ± 0.09	0.15 ± 0.11	0.15 ± 0.12	0.08 ± 0.07
0.39 ± 0.31	0.42 ± 0.35	0.34 ± 0.21	0.31 ± 0.31	0.35 ± 0.27

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0	0	0
0.01	0.05	0.02	0.03	0
0.01	0.14	0.41	0.19	0.15
0.11	0.62	0.27	0.38	0.05

Initial total weight of 200.00 distributed uniformly among the 25 boxes:

0	0	0	0	0
0	0	0.01	0	0
0.01 ± 0.02	0.04 ± 0.05	0.07 ± 0.04	0.04 ± 0.03	0.01 ± 0.02
0.09 ± 0.10	0.11 ± 0.10	0.16 ± 0.11	0.12 ± 0.10	0.07 ± 0.07
0.35 ± 0.37	0.42 ± 0.39	0.33 ± 0.26	0.3 ± 0.335	0.41 ± 0.35

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0	0.03	0.01	0
0	0.04	0.01	0.04	0
0	0	0.13	0.23	0.17
0.31	0.05	0.27	0.29	0.17

The graphs of acceptence vs. temperature show that the acceptence deviates from its monotonically decreasing behavior (as it did for the lower step sizes before)

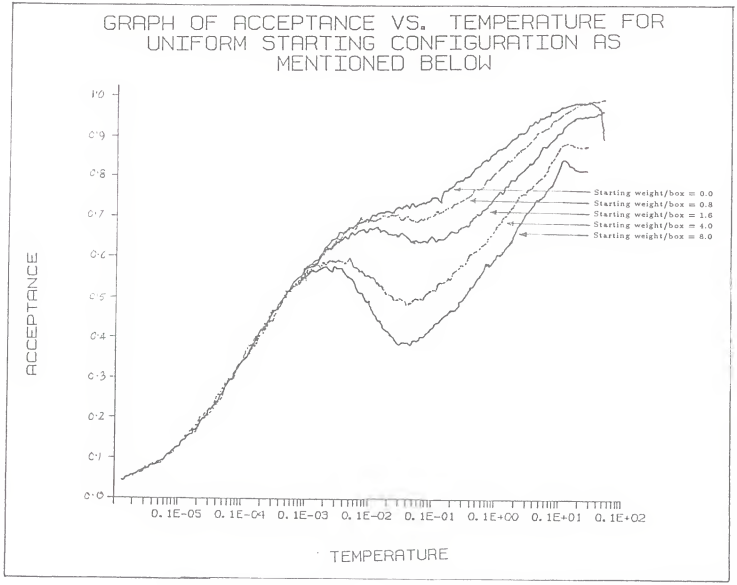
GRAPH OF ACCEPTANCE VS. TEMPERATURE FOR
UNIFORM STARTING CONFIGURATION AS
MENTIONED BELOW
ACCEPTANCE
1·0
0·9
0·8
0·7
0·6
0·5
0·4
0·3
0·2
0·1
0·0
Starting weight/box = 0.0
Starting weight/box = 0.8
Starting weight/box = 1.6
Starting weight/box = 4.0
Starting weight/box = 8.0
0.1E-05 0.1E-04 0.1E-03 0.1E-02 0.1E-01 0.1E+00 0.1E+01 0.1E+02
TEMPERATURE

Fig - 3

4-14

for larger deviations in the total starting weight. This shows that a similar effect can be produced by varying the step size only or by varying the starting configuration only, and hence these two parameters are closely related to each other.

Another interesting aspect observed by comparing the final distribution of weights obtained by the annealing algorithm starting with a configuration skewed to the left and a uniform starting configuration is that the skewed starting configuration also produces a skewed final average configuration, whereas a uniform starting configuration does not show any average skewed behavior in the final configuration. This effect depends on the choice of the step size. The effect is very pronounced for large step sizes, and it diminishes as the step size gets smaller. The reason behind this behavior is not yet fully understood and has not been investigated in this thesis.

Application of the algorithm on other configurations

We have so far been applying the annealing algorithm to test its efficiency to recover an actual weight distribution where the weights were unity only in the fourth layer from top and in the second and fourth columns, with zero everywhere else. Now, we apply the algorithm on other configurations as shown below.

Configuration 1:

0 1 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

Average configuration recovered by the algorithm:

(Average of 30 trials)

0	0.99	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Configuration 2:

0	0	0	0	0
0	1	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Average configuration recovered by the algorithm:

(Average of 30 trials)

0	0.02	0	0	0
0.09 ± 0.03	0.55 ± 0.07	0.04 ± 0.02	0	0
0.15 ± 0.11	0.27 ± 0.12	0	0	0
0 ± 0.02	0 ± 0.01	0	0	0
0	0	0	0	0

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0.72	0.01	0	0
0.32	0.05	0.01	0	0
0	0	0	0	0
0	0	0	0	0

Configuration 3:

0	0	0	0	0
0	0	0	0	0
0	1	0	0	0
0	0	0	0	0
0	0	0	0	0

Average configuration recovered by the algorithm:

(Average of 30 trials)

0	0	0	0	0
0 ± 0.01	0.08 ± 0.02	0.03 ± 0.01	0	0
0 ± 0.08	0.18 ± 0.09	0.04 ± 0.02	0	0
0.19 ± 0.14	0.14 ± 0.09	0.06 ± 0.04	0	0
0.22 ± 0.18	0.17 ± 0.12	0.04 ± 0.04	0	0

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0.1	0.01	0	0
0.09	0.29	0.05	0	0
0.17	0.16	0.04	0	0
0.21	0.03	0.04	0	0

Configuration 4:

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	1	0	0	0
0	0	0	0	0

Average configuration recovered by the algorithm:

(Average of 30 trials)

0	0	0	0	0
0	0	0	0	0
0.05	0.04	0.01	0	0
0.13	0.12	0.10	0.01	0
0.29	0.20	0.12	0.06	0.01

Configuration corresponding to the lowest cost function:

0	0	0	0	0
0	0.01	0	0	0
0	0.02	0.01	0	0
0.09	0.3	0.03	0	0
0.4	0.17	0.14	0	0

It is seen from the results with the above configurations, that the algorithm reproduces the configuration 1 quite accurately.

For configuration 2, the algorithm fails to produce the exact configuration (unlike the previous case) but it produces a higher weight in the proper column than the other columns. In this case the algorithm cannot produce the right box, but definitely identifies the right column. Besides if we look at the result corresponding to the lowest cost function we find that it gives a very good estimate of the exact location of the weight.

For configuration 3, the algorithm fails to determine even the correct column in most of the cases, but it does so in the case having the lowest cost function. Here, the algorithm is only capable of showing that most of the weight is skewed to the left side of the volume, and towards the bottom layers. It is, therefore, seen that

the present algorithm is capable of recovering the weights for the two topmost rows, but does not do so for the layers deeper down. But, it can produce higher weights in the proper columns than in the other ones, even for the weights in the deeper layers.

One interesting aspect to be noted from the results of this set of observations is that the algorithm in all the above three cases gives an approximate estimate of the total amount of weight present in the volume within an accuracy of about 25 percent. This surely depends on the *a priori* assumption that the **weights cannot be negative.**

Further work in this area can be carried out to modify the algorithm so as to recover the correct weights even for the deeper layers more efficiently. The most important modifications are:

1. The step size can be varied during the algorithm, making it gradually smaller during the algorithm, so as to make the adjustments in the weight distribution finer as the configuration comes closer and closer to the frozen state.

2. Besides, the updating scheme of the weight at any temperature can be modified to make more sophisticated weight transfers among the different boxes to drive the configuration towards the desired state more efficiently. For example, the transferring scheme can be modified to incorporate weight redistribution between three boxes, or weight transfer from one box to another, instead of only adding (or subtracting) weights to a box, as has been done in the current algorithm.

References:

[1]. Analysis of Numerical Methods - Eugene Isaacson and Herbert Bishop Keller - John Wiley & Sons, Inc., New York, Library of Congress Catalog Card Number: 66-17630.

[2]. Optimization by Simulated Annealing - S. Kirpatrick, C. D. Gelatt, Jr., M. P. Vecchi, pp 671, 13th May 1983, Science, Vol. 220, Number 4598.

APPENDIX

```
      program generate_data
c
c     program to compute the three dimensional integral  over the
c     a predefined box, with Simpson's rule.
c
      external func
      open(unit=7,status='old',form='unformatted')
c
      write(6,*)' type the x-coordinates of the end points of slab:'
      write(6,*)' (type the lower value first)'
c
      read(5,*)xlimitlow,xlimithi
      write(7)xlimitlow,xlimithi
c
      write(6,*)' type the y-coordinates of the end points of slab:'
      write(6,*)' (type the lower value first) '
c
      read(5,*)ylimitlow,ylimithi
      write(7)ylimitlow,ylimithi
c
      write(6,*)' type the z-coordinates of the end points of slab:'
      write(6,*)' (type the lower value first)'
c
      read(5,*)zlimitlow,zlimithi
      write(7)zlimitlow,zlimithi
c
      write(6,*)' type the number of points for the x-integration:'
      write(6,*)' (value must be an even integer)'
c
      read(5,*)nx
c
      write(6,*)' type the number of points for the y-integration:'
      write(6,*)' (value must be an even integer)'
c
      read(5,*)ny
c
      write(6,*)' type the number of points for the z-integration:'
      write(6,*)' (value must be an even integer)'
c
      read(5,*)nz
      write(7)nx,ny,nz
c
      write(6,*)' type the number of boxes in the x,y,z direction:'
c
      read(5,*)nxbox,nybox,nzbox
      write(7)nxbox,nybox,nzbox
c
c     compute the box size on each axis:
c
      xeps = (xlimithi - xlimitlow)/float(nxbox)
      yeps = (ylimithi - ylimitlow)/float(nybox)
      zeps = (zlimithi - zlimitlow)/float(nzbox)
      write(7)xeps,yeps,zeps
c
c     vary "a" and "b":
c
      do 50 iax = 1,1
      do 48 ibx = 1,11
      weight = 1.
      xa = float(iax - 1)
```

```
      xb = float(ibx - 1)
      ya = 0.
      yb = 0.
      za = 0.
      zb = 0.
      write(7)xa,ya,za,xb,yb,zb
      write(7)weight

      do 300 iz = 1,nzbox
         do 200 iy = 1,nybox
            do 100 ix = 1,nxbox
c
c              compute the end points of the box:
c
               x1 = float(ix -1) * xeps  + xlimitlow
               x2 = x1 + xeps
               y1 = float(iy -1) * yeps  + ylimitlow
               y2 = y1 + yeps
               z1 = float(iz -1) * zeps  + zlimitlow
               z2 = z1 + zeps
               call simp3dim(xa,ya,za,xb,yb,zb,x1,x2,y1,y2,z1,z2,nx,ny,
     +                       nz,value)

               write(7)ix,iy,iz,value
100            continue
200         continue
300      continue

48       continue
50       continue
         stop
         end
c
c*******************************************************************************
c
      subroutine simp3dim(xa,ya,za,xb,yb,zb,x1,x2,y1,y2,z1,z2,nx,ny,
     +                    nz,sum)
c
c     compute the step size for integration:
c
      hx = (x2-x1)/nx
      hy = (y2-y1)/ny
      hz = (z2-z1)/nz

      sum = 0.0

      do 300 ix = 1, nx + 1
         do 200 iy = 1, ny + 1
            if (ix .eq. 1)  go to 1
            if (ix .eq. (nx + 1)) go to 1
            ix1 = ix/2
            ix2 = ix1 * 2
            ix3 = ix - ix2
            if (ix3 .eq. 0) go to 2
            factorx = 2.
            go to 3
 1          factorx = 1.
            go to 3
 2          factorx = 4.
```

```
3             if (iy .eq. 1)  go to 5
              if (iy .eq. (ny + 1)) go to 5
              iy1 = iy/2
              iy2 = iy1 * 2
              iy3 = iy - iy2
              if (iy3 .eq. 0) go to 6
              factory = 2.
              go to 7
5             factory = 1.
              go to 7
6             factory = 4.

7             vx = x1 + float(ix - 1)*hx
              vy = y1 + float(iy - 1)*hy
              suminit = 0.
              suminit = suminit +(func(vx,vy,z1,xa,ya,za,xb,yb,zb)
     +                    + func(vx,vy,z2,xa,ya,za,xb,yb,zb))
     +                    * factorx * factory
              vz = z1 + hz
              suminit = suminit + 4.0
     +                    * func(vx,vy,vz,xa,ya,za,xb,yb,zb)
     +                    * factorx * factory

              sumz = 0.
              do 100 iz = 3, (nz-1), 2
              vz = z1 + float(iz - 1)*hz
              v = func(vx,vy,vz,xa,ya,za,xb,yb,zb)
              sumz = sumz + 2. * v
              vz = z1 + float(iz)*hz
              v = func(vx,vy,vz,xa,ya,za,xb,yb,zb)
              sumz = sumz + 4. * v
100           continue
              sumz = sumz * factorx * factory
              sum = sum + suminit + sumz
200         continue
300   continue

      sum = sum * hx * hy * hz / (3. * 3. * 3.)

c      call lib$show_timer

      return
      end

*******************************************************************************

      function  func(vx,vy,vz,xa,ya,za,xb,yb,zb)
      denom1 = (xa - vx)**2 + (ya - vy)**2 + (za - vz)**2
      denom1 = sqrt(denom1)
      denom2 = (xb - vx)**2 + (yb - vy)**2 + (zb - vz)**2
      denom2 = sqrt(denom2)
      func = 1./(denom1 * denom2)
      return
      end
```

```
      program anneal
c
c        this code reads the values of initial seeds for the random number
c        generator from the file for008.dat,
c
      implicit real*8 (a-h,o-u,w-z)
      real*4 valcomb
      real*4 rannyu
      dimension value(11,11,25,25),suminit(11,11),sumtrial(11,11)
      dimension sumnew(11,11),weight(25,24),weightnew(25,24),iran(4)
      dimension iw(25),valcomb(11,11,5,5),sumt(11,11)
      dimension tryit(3),iran1(4),ixbx(25),izbx(25)
c
c     value = the values of integral for each box read from a file that was
c             computed before.
c     suminit = the integral over all the boxes for a receiver and transmitter
c               location.
c     sumtrial = the integral over all the boxes computed after the the boxes
c                are assigned random weights.
c     sumnew = the integral over all the boxes computed after updating the
c              random weights of the boxes.
c     valcomb = the precomputed three dimensional integral for the kernel
c               computed over each of the boxes by three dimensional
c               Simpson's rule by the FORTRAN program called GENERATE as
c               presented before.
c     weight = random weights assigned to the boxes.
c     weightnew = the updated random weight.
c     iran = the seeds for the random number generator.
c
      data ifive/5/,izero/0/
      data accept_tolerance/0.01/
c
c     open the necessary files:
c
      open(unit=7, file='final_accept.out', status ='new')
c
c       the final configuration is written into unit 7.
c
      open(unit=17,file='val.dat',status='old',form='unformatted')
c
c       unit 17 contains the precomputed values of "valcomb".
c
      open(unit=18,status='old')
c
c       unit 18 contains the seeds for the random number generator.
c
      open(unit=19, file ='config.dat',status ='old')
c
c       unit 19 contains the weights from previous runs.
c
c     the final seeds for the setrn are written in unit 18.
c
      rewind(unit=18)
      read(18,*)iran
      read(5,*)step,t,fac,ilp2
      tstart = t
      write(7,*) 'iran =', iran
c
c     step = the step size for computing the new random weights.
c
```

```
c
c           write out the necessary input parameters for check up:
c
      write(6,*)' iran=', iran
      write(6,1020)
1020  format(3x,'Accept_tolerance')
      write(6,1025)accept_tolerance
1025  format(1x,f14.5,5x,f14.5)
      write(6,1030)
1030  format(1x,'Starting Step',2x,'Starting temp',2x,
     +          'Temp_fac',5x,'kiter')
      write(6,1035)step, t, fac, ilp2
1035  format(1x,f10.5,5x,f10.5,5x,f7.4,3x,i8)
c
c     read the initial values of each big box for each combination of receiver
c     and transmitter.
      do 40 ixb = 1,11
      do 38 ixa = 1,ixb
      do 36 ixcount = 1,5
      do 34 izcount = 1,5
      read(17)valcomb(ixa,ixb,ixcount,izcount)
34    continue
36    continue
38    continue
40    continue
c
      do 45 i = 1,4
45    iran1(i) = iran(i)
      call setrn(iran1)
c
      call second(t1)
      icount = 1
171   continue
      write(6,*)' iran =', iran1
c
c     initialize timer
c
c         read initial weights (random) to the boxes:
c
      rewind(unit=19)
      do  65 ixbox = 1,5
      do  60 izbox = 1,5
60    read(19,*) weight(ixbox,izbox)
65    continue
c
c     compute suminit:
c
         do 75 ixa = 1,11
         do 74 ixb = ixa,11
         suminit(ixa,ixb) = 1.* valcomb(ixa,ixb,2,4) +
     +                      1.* valcomb(ixa,ixb,4,4)
74    continue
75    continue
c
      write(6,*)' finished computing all suminits:'
c
c     initialise the sumtrials:
c
         do 80 ixa = 1,11
         do 79 ixb = 1,11
```

```
79        sumtrial(ixa,ixb) = 0.
80        continue

          do 85 ixb = 1,11
          do 84 ixa = 1,ixb
            do 83 izz = 1,5
            iz = 6 - izz
            do 82 ix = 1,5
82          sumtrial(ixa,ixb) = sumtrial(ixa,ixb) +
     +                          weight(ix,iz) * valcomb(ixa,ixb,ix,iz)
83          continue
84      continue
85      continue
c       compute the initial error:
        errorold = 0.
        do 90 ixb = 1,11
        do 89 ixa = 1,ixb
        error = (suminit(ixa,ixb) - sumtrial(ixa,ixb))
89      errorold = errorold + 2. * error**2
90      continue
        do 95 ixa = 1,11
        errorsub = (suminit(ixa,ixa) - sumtrial(ixa,ixa))
95      errorold = errorold - errorsub**2
c
        t = t/fac
        call second(t11)
        do 145 itemp = 1,999999
c
c       reduce temperature
c
        t = t * fac
        if (t .le. 1.d-24) go to 150
c
        ireject = 0
        iaccept = 0
        iup = 0
c
c       enter the loop
c
        do 135 ilocp2 = 1,ilp2
c
c           add/subtract from a random box:
c
            i = rannyu(0) * 5. + 1
            j = rannyu(0) * 5. + 1
            weightnew(i,j) = weight(i,j) + (rannyu(0) - 0.5) * step
            if ( weightnew(i,j) .ge. 0.) then
                  iflag = 1
                  change = weightnew(i,j)  - weight(i,j)
                  errornew = 0.
                  do 105 ixb = 1,11
                  do 104 ixa = 1,ixb
                  sumnew(ixa,ixb) = sumtrial(ixa,ixb) +change*
     +                    valcomb(ixa,ixb,i,j)
                  err = suminit(ixa,ixb) - sumnew(ixa,ixb)
104               errornew = errornew + 2.* err**2
105               continue
                  do 106 ixa = 1,11
                  err = suminit(ixa,ixa) - sumnew(ixa,ixa)
106               errornew = errornew - err**2
```

```
c
        else

c
c               reject the move:
c
c
                iflag = 4
        end if

c
c       check for acceptance or rejection of the moves:
c
        if ( iflag .eq. 4 ) then
             iflag_reject = 1
        else if (errornew .lt. errorold ) then
                  iflag_reject = 0

        else
                prob = dexp ((errorold - errornew) / t)
                x = rannyu(0)
                 if (x .lt. prob) then
                       iflag_reject = 0
                       iup = iup + 1

                 else
                       iflag_reject = 1
                 end if .
        end if

c
c       take actions according to the iflag_reject:
c
        if ( iflag_reject .eq. 1) then
             ireject = ireject + 1
        else
             iaccept = iaccept + 1
             weight(i,j) = weightnew(i,j)
             errorold = errornew
             do 134 ixa = 1,11
             do 133 ixb = ixa,11
133          sumtrial(ixa,ixb) = sumnew(ixa,ixb)
134          continue
        end if
135     continue
        acceptance = dfloat(iaccept)/dfloat(iaccept + ireject)
1001    format(2e20.9)
c
        if (acceptance .le. accept_tolerance) go to 150
        ratio = dfloat(iup)/dfloat(iaccept)
            print*,' t, ratio, acceptance=', t, ratio, acceptance
145     continue
c
150     continue
c
        do 155 iz = 1,5
        do 154 ix = 1,5
154     iw(ix) = weight(ix,iz) * 100.
        write(6,1000)iz,(iw(i),i=1,5)
155     write(6,*)' '
1000    format(5x,i2,3x,5i4)
        write(6,*)' t=',t

c
        call second(t22)
        t33 = (t22 - t11)/60.
```

```
c     compute the actual error:
c     initialise the sumtrials:
         do 160 ixa = 1,11
         do 159 ixb = 1,11
159      sumtrial(ixa,ixb) = 0.
160      continue

         do 165 ixb = 1,11
         do 164 ixa = 1,ixb
           do 163 izz = 1,5
           iz = 6 - izz
           do 162 ix = 1,5
162        sumtrial(ixa,ixb) = sumtrial(ixa,ixb) +
     +                         weight(ix,iz) * valcomb(ixa,ixb,ix,iz)
163        continue
164   continue
165   continue
c
      erroract = 0.
      do 170 ixb = 1,11
      do 169 ixa = 1,ixb
      error = (suminit(ixa,ixb) - sumtrial(ixa,ixb))
169   erroract = erroract + 2. * error**2
170   continue
      do 175 ixa = 1,11
      errorsub = (suminit(ixa,ixa) - sumtrial(ixa,ixa))
175   erroract = erroract - errorsub**2
      write(6,*)' actual error =',erroract
      errorold = erroract
c
c        write the final configuration for the future computation of the
c        average configuration of all the runs:
c
      do 180 iz = 1, 5
      do 179 ix = 1, 5
179   write(7,*)weight(ix,iz)
180   continue
      write(7,*)' -------------------------------'
c
c
c         write the last random seed
c
      rewind(unit=18)
      call savern(iran1)
      write(18,*)iran1
c
      icount = icount + 1
      if (icount .lt. 4) then
         t = tstart
         go to 171
      end if
         call second(t2)
         write(6,*) 'time (in minutes): ',(t2-t1)/60.
c
      stop
      end
```

STUDY OF SIMULATED ANNEALING FOR LEAST SQUARES OPTIMIZATION

by

PRANAB KUMAR BANERJEE

B.Tech. (Hons.) Indian Institute of Technology, Kharagpur, 1985

AN ABSTRACT OF A MASTER'S THESIS
submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Physics
KANSAS STATE UNIVERSITY
Manhattan, Kansas
1988

ABSTRACT

There are two main classes of optimization techniques - deterministic optimization and stochastic optimization. For many dimensional optimization tasks, the deterministic techniques become computationally very cumbersome, and some of these techniques can only produce a locally extremal state. Stochastic optimization techniques, on the other hand, are computationally more efficient for multidimensional optimization tasks. This thesis studies one such stochastic optimization algorithm called "Simulated Annealing"which has the ability to come out of a locally extremal state even if it happens to reach one. This algorithm has been studied here for least squares optimization with a particular application to an "Inverse Problem".