Scope of Gradient and Genetic Algorithms in Multivariable Function Optimization*

Gholam Ali Shaykhian[a] and **S.K. Sen**[b]

[a]*National Aeronautics and Space Administration (NASA), Engineering Directorate, NE-C1, Kennedy Space Center, FL 32899, United States*
ali.shaykhian@nasa.gov

[b]*Department of Mathematical Sciences, Florida Institute of Technology, 150 West University Boulevard, Melbourne, FL 32901-6975, United States*
sksen@fit.edu

Abstract. Global optimization of a multivariable function — constrained by bounds specified on each variable and also unconstrained — is an important problem with several real world applications. Deterministic methods such as the gradient algorithms as well as the randomized methods such as the genetic algorithms may be employed to solve these problems. In fact, there are optimization problems where a genetic algorithm/an evolutionary approach is preferable at least from the quality (accuracy) of the results point of view. From cost (complexity) point of view, both gradient and genetic approaches are usually polynomial-time; there are no serious differences in this regard, i.e., the computational complexity point of view. However, for certain types of problems, such as those with unacceptably erroneous numerical partial derivatives and those with physically amplified analytical partial derivatives whose numerical evaluation involves undesirable errors and/or is messy, a genetic (stochastic) approach should be a better choice. We have presented here the pros and cons of both the approaches so that the concerned reader/user can decide which approach is most suited for the problem at hand. Also for the function which is known in a tabular form, instead of an analytical form, as is often the case in an experimental environment, we attempt to provide an insight into the approaches focusing our attention toward accuracy. Such an insight will help one to decide which method, out of several available methods, should be employed to obtain the best (least error) output.

* An invited talk at the *Fifth International Conference on Dynamic Systems and Applications* (May 30-June 02, 2007) in Atlanta, Georgia.

Scope of Gradient and Genetic Algorithms in Multivariable Function Optimization[1]

Gholam Ali Shaykhian[a] and **S.K. Sen**[b]

[a]*National Aeronautics and Space Administration (NASA), Engineering Directorate, NE-C1, Kennedy Space Center, FL 32899, United States*
ali.shaykhian@nasa.gov

[b]*Department of Mathematical Sciences, Florida Institute of Technology, 150 West University Boulevard, Melbourne, FL 32901-6975, United States*
sksen@fit.edu

Abstract. Global optimization of a multivariable function — constrained by bounds specified on each variable and also unconstrained — is an important problem with several real world applications. Deterministic methods such as the gradient algorithms as well as the randomized methods such as the genetic algorithms may be employed to solve these problems. In fact, there are optimization problems where a genetic algorithm/an evolutionary approach is preferable at least from the quality (accuracy) of the results point of view. From cost (complexity) point of view, both gradient and genetic approaches are usually polynomial-time; there are no serious differences in this regard, i.e., the computational complexity point of view. However, for certain types of problems, such as those with unacceptably erroneous numerical partial derivatives and those with physically amplified analytical partial derivatives whose numerical evaluation involves undesirable errors and/or is messy, a genetic (stochastic) approach should be a better choice. We have presented here the pros and cons of both the approaches so that the concerned reader/user can decide which approach is most suited for the problem at hand. Also for the function which is known in a tabular form, instead of an analytical form, as is often the case in an experimental environment, we attempt to provide an insight into the approaches focusing our attention toward accuracy. Such an insight will help one to decide which method, out of several available methods, should be employed to obtain the best (least error) output.

1. Introduction

Let $x = [x_1 \quad x_2 \cdots x_n]$ be an n dimensional vector. The problem is to find/compute a vector x that globally minimizes the function $f(x)$ which is given in a tabular form or in an analytical form. The term *optimization* implies either minimization or maximization. Minimizing the function $f(x)$ is the same as maximizing the function $[-f(x)]$. Unless otherwise specified, we will imply by the term *minimization* the global minimization. If the constraints or, equivalently, the bounds on each element/variable x_i are specified, then the problem is called a constrained function optimization problem. Else, it is an unconstrained function optimization problem.

---

[1] An invited talk at the *Fifth International Conference on Dynamic Systems and Applications* (May 30-June 02, 2007) in Atlanta, Georgia.

Although a real world/practical design problem is rarely unconstrained, a study of this (unconstrained) class of problems is important for the following reasons.

(i) The unconstrained minimization algorithms provide a deeper insight required for the study of constrained minimization techniques.
(ii) Some of the robust[2] algorithms for constrained minimization need the use of unconstrained minimization methods.
(iii) The constraints do not have significant effect in certain design problems.
(iv) The unconstrained minimization algorithms can solve certain engineering analysis problems such as the nonlinear displacement response problems involving a structure under a specified load. Here the potential energy is minimized.

Conversion of constrained to unconstrained problem A constrained function optimization problem can be made an unconstrained function optimization problem. For instance, the constraint $a_i \le x_i \le b_i$ is equivalent to $x_i = a_i + (b_i - a_i)\sin^2 \varphi$, where φ is unconstrained. Thus, this equation can be used where x_i appears. However, such procedures can become very complicated. Hence several simpler procedures have been developed [1-8].

Gradual increase in importance of genetic algorithms over deterministic ones Always we need a numerical solution for the real world implementation. An analytical solution is of no use to an engineer until it is translated into numbers. The most important tool for a numerical solution is a computer — rather a digital computer, although an analog computer has its own usage in certain problems such as obtaining a discrete Fourier transform through the fast Fourier transform. However, the accuracy in an analog computer unlike that in a digital computer is very limited, i.e., it is usually not more accurate than 0.001%. This figure translates to four significant digits. This limitation is due to that of the device measuring a physical quantity in an analog computer, which is usually not more accurate than 0.01% [9]. A digital computer may be used to practically any finite precision (word length) and is so dominant that by the term *computer*, we would imply a digital computer and not an analog one. An analog computer may be much faster than a digital computer as in the case of a discrete Fourier transform. But the digital computers over years are progressively improved in speed, memory, as well as band width. Every 18 months the CPU (central processing unit) speed is doubling, every 12 months band width is doubling while every 9 months hard disk space is doubling. Consequently, many problems which were posed earlier and could not be solved because of computing resource limitations are now being solved. We provide below a brief account of past computing years and the gradual increase in importance of randomized algorithms such as the genetic algorithms over deterministic algorithms such as the gradient methods.

Before we proceed, we like to point out that the speed of computing, the storage space as well as the band width go hand in hand. That is, just increasing the computing speed without increasing the memory and band width could result in an operational bottle-neck since a high speed CPU will be bogged down due to too many data retrieval and storage operations if the

[2] A robust algorithm is one that must produce correct output regardless of whether the input actually belongs to the restricted domain or not, i.e., whether it is an inlier or not. In fact, a subjective implication of robust algorithm is the insensitivity to an outlier/noise.

memory size as well as band width is not commensurable, i.e., if these are not relatively large. As stated in the previous paragraph, both CPU and memory space is progressively improving

Pre-high speed computing years (1946-1964) This nineteen year period may be divided into two parts of the first generation (vacuum tubes) computers — early first generation (1946-53) and late first generation (1953-59) and the second generation (transistors) computers (1959-64) [10]. The main memory cycle time was .04-40 ms (milliseconds) during the early first generation while it was .01-.02 ms during the late first generation and .002 - .01 ms during the second generation. An early first generation computer was capable of executing about 10^3 operations per second on an average while a late first generation could execute about 5×10^4 operations per second on an average. Hardly a negligible fraction (compared to modern computers) of practical computing existed during 1946-53. Randomized algorithms were not perceived during these years as a viable alternative to deterministic ones. This is because randomized algorithms were not so much developed as it is today. Also these needed apparently large amount of computation compared to that required by a deterministic one. In reality, however, all randomized algorithms are polynomial-time, i.e., fast. Specifically genetic algorithms (global search techniques) were almost nonexistent during 1946-64. Psychologically we were more comfortable with deterministic algorithms than with nondeterministic ones. As a matter of fact we did not have much faith/confidence about the result/output which remains variant at each run. This is because of the seed to generate required random numbers for the randomized/probabilistic algorithm differs from one run to another — a situation very much unlike any deterministic algorithm such as the gradient methods. A deterministic algorithm will always produce exactly the same output on the same computer, no matter how many times the program (algorithm) is run/executed. Thus gradient algorithms were the only practical acceptable means to optimize a multivariable function and very little of these algorithms were computerized nor were there as sophisticated gradient methods/other deterministic methods as we have today (2007).

High-speed computing years (1964-1975) This eleven years may be divided into two parts of the third generation (monolithic integrated circuits) computers — early third generation (1964-69) and late third generation (1969-75). The main memory cycle time was 0.5-2 μs during the early third generation while it was 0.02-1 μs during the late third generation. An early third generation computer could execute 10^6 (one million) operations per second on an average while a late third generation computer was capable of executing about 20×10^6 (twenty million) operations per second on an average. The enhanced speed allowed the scientists/engineers to explore more compute intensive problems which were hitherto discouraged due to processing/memory speed limitations. Also, they developed newer and newer algorithms suitable for computation for real world problems. Randomized algorithms such as the genetic algorithms and evolutionary approaches started gaining popularity increasingly. These algorithms started gaining momentum and being considered as possible candidates for practical computations along with the deterministic ones. But these were yet to become sufficiently appealing for extensive computation in lieu of deterministic ones and were yet to be widely accepted means of global function optimization.

Super-high-speed computing years (1975-1990) The term *supercomputer* has a time-dependant non-rigid definition since today's supercomputer tends to become tomorrow's normal computer. Further, there is no generally accepted definition for fourth generation (very large scale

integrated circuits and possibly with vector processors) as well as fifth generation (depicting artificial intelligence, which is mainly due to the software simulation of the natural intelligence) computers. It is thus not useful to carry the concept of computer generation beyond the third generation. We consider computers introduced since 1975 as modern computers and refer to the third generation computers as those of the past. However, for the purpose of speed relative to that of the past computers, a modern computer was loosely termed as a supercomputer if its speed exceeds 100 million operations per second (one hundred million floating point operations per second, i.e., one hundred megaflops). Such a technological improvement gave a significant impetus to scientists/researchers to explore much more compute intensive algorithms such as the randomized ones and perceive the scope/utility of these algorithms over the deterministic algorithms such as the gradient methods for global optimization. A gradient method could get stuck at a local minimum unless appropriate measures are taken to get out of this situation while a genetic/evolutionary algorithm has much in general much less probability to

Ultra-high-speed computing years (1990-onwards) Compared to the foregoing speeds, it would be reasonable to term a processing speed exceeding 1000 million (i.e., one billion) flops as an ultra-high speed. The ultra-high frequency band is generally accepted as 3000-300 megahertz. Electrical signals propagates no faster than the speed of light. A random access memory used to 10^9 cycles per second (one gigahertz) will deliver information at 10^{-10} (i.e., 0.1 nanosecond) speed if its diameter is 3 centimeters since in 10^{-10} seconds, light travels 3 centimeters [9]. It is physics rather than technology and architecture sets up the limits/barriers to increase the computational speed arbitrarily. The physical barriers are the (i) speed of light, (ii) the thermal efficiency, and the quantum barriers. Per mass of hydrogen atom (1.67×10^{-24} gm), maximum 2.505×10^{23} bits/sec can be theoretically processed/transmitted. Since the estimated number of protons in the universe is 10^{73}, if the whole universe is dedicated to information processing, then no more than 7.9×10^{103} bits per year can be processed [9]. The ultra-high speeds along with ultra-large memory and ultra-large band-width have permitted the scientists to encroach into the realm of hitherto unexplored NP-hard problems such as a large traveling salesman problem (TSP) of immense practical importance in a meaningful way. While a deterministic algorithm for the TSP is combinatorial/exponential-time needing evaluation of (n−1)! paths to obtain the exact minimum cost path, a genetic (heuristic) algorithm which is always polynomial-time would need relatively very little computation to provide us a low cost path, which though may/may not be the exact minimum cost path, that is accepted and used by the traveling salesman. Maybe in future a better (lower cost) path will be found by the algorithm possibly with increased computing power (processing speed, storage, and band width). The purpose is to impress on the fact that randomized algorithms will be the only tool to explore the vast world of NP-hard problems [9]. The deterministic algorithms will have no entry to this world as these could take billions of centuries to produce the required output. Even the estimated age of the universe is a numerical zero compared to this computation time. In the present context, we are involved in only polynomial time deterministic algorithms such as the gradient methods as well as the randomized algorithms such as the genetic algorithms for function optimization which is a polynomial-time problem. Even in such polynomial-time problems, genetic algorithms appear to be the only options for most real-world problems.

Computational complexity The optimization of the function $f(x)$ considered here is a polynomial-time problem and consequently the concerned gradient (deterministic) and genetic

(randomized) approaches are all polynomial-time (i.e., fast). In this respect, all these algorithms are attractive and without any significant edge of one category over the other.

Accuracy, flexibility, and simplicity We are essentially concerned with practical application of function optimization. We have considered typical problems including test ones and found that the genetic algorithms are significantly better than the gradient algorithms in terms of accuracy, flexibility as well as simplicity. Often partial derivatives computation accurately in a gradient method turns out to be a bottle-neck.

Global versus local optimization The gradient methods usually give a local minimum. This is not the goal of our problem. We need to determine the global minimum. So we need to devise a way possibly divide the domain into an appropriate number of sub-domains, each having only one minimum and then apply a gradient method for each sub-domain. On the other hand, genetic algorithms have the tendency to search the global minimum and hardly get stuck at a local minimum.

In section 2, we discuss gradient methods which have specific scopes. Also, in this section, we present genetic algorithms with their applicability to solve various function optimization. Besides, we have also included Matlab codes for some of the algorithms. In section 3, we include typical examples including test ones to illustrate the gradient as well as genetic algorithms along with gradient method implemented by Matlab. We have stressed on accuracy (quality of result) and complexity (cost of the result). Also included are some relevant graphs (up to 3 dimensions) for the purpose of visualization of the nature/character of the function. We demonstrate in this section that genetic algorithms are the winners. Section 4 comprises conclusions.

2. Gradient and Genetic Algorithms

Gradient methods The Davidon-Powell variable metric method and the Fletcher-Reeves conjugate gradient method are among the most efficient general-purpose gradient methods for function optimization while the Powell-Smith method can be used with advantage when the derivative computation presents difficulty [8, 12-20]. Immense attention was given to develop these methods during 1960s and 1970s by the concerned scientists/researchers and thus 1960-1975 may be called the golden period in the development of gradient algorithms for function optimization. During this period, genetic algorithms were not practically known. Gradient methods were the sole dominant algorithms for the optimization on a main-frame digital computer. The Matlab command **fminsearch** has implemented the popular Nelder-Mead downhill simplex method (direct search) proposed by J.A. Nelder are R. Mead in 1965 [20] in its software. While sometimes a combination of Nelder-Mead method and a genetic algorithm [21] can be used profitably, this combined method lacks in simplicity (no partial derivative computations) and generality (no divergence) inherent in a genetic algorithm alone.

We omit here the description of the Davidon-Powell variable metric, the Fletcher-Reeves conjugate gradient, the Powell-Smith, and the Nelder-Mead methods used here to conserve space as these are available in the foregoing literature [8, 12-21] as well as in the internet. However, we provide a brief description of the multi-dimensional bisection based genetic algorithm [22] as this method is not widely and possibly readily available.

Genetic Algorithm A genetic algorithm (GA) inspired by Darwin's theory of evolution and employed to solve optimization problems — unconstrained or constrained — uses an evolutionary process. It is a search algorithm mimicking mechanics of natural selection and natural genetics. A GA is used to find a true or an approximate solution to an optimization problem occurring in engineering, computer science, economics, mathematics, or any other area.

In 1975, John Holland, University of Michigan, developed a GA by describing how to apply the principles of natural evolution to solve optimization problems [23]. His initial goals were two-fold, viz., improving the understanding of natural adaptation process, and designing artificial systems which may have properties similar to natural systems. Genetic algorithms constitute a branch of evolutionary algorithms (EA). EA (inspired by evolutionary biology) mechanisms are similar to biological evolution involving reproduction, mutation, recombination, natural selection, and survival of the fittest.

A GA was motivated from biological processes such as crossovers, and mutations involving chromosomes — gene-carrying body in the nucleus of a cell, and DNA — the main constituents of the chromosomes of all living things. A genetic approach is to generate successive sets of generations (solutions), making each new generation to inherit properties from the best available chromosomes of the precedent. In this process, the weak chromosomes are replaced (survival of the fittest) with new chromosomes which have gone through crossover and mutation operations. The weakest chromosomes (12.5%) are crossover (recombination) with the strong (12.5%) chromosomes.

The GA for a general search problem starts with creation of a population of individuals, represented by chromosomes. Individuals are a set of bit/character (non-bit) strings analogous to what we see in our own DNA. The individuals in the population then go through a process of evolution. Those individuals that are fitter/stronger, while competing for resources in the environment, are more likely to survive and propagate their genetic material (genome).

A GA encodes/represents 'the potential solutions' as a set of strings of numbers/characters. Two solutions are mated to form new solutions (off-springs) through crossover/mutation.

The GA requirements are (i) a genetic representation of the solution domain and (ii) an objective function (a fitness function) to evaluate the solution domain. The first step in applying a GA is to encode appearance, behavior, physical qualities of an individual chromosome. Encoding involves changing the values of x_i and y_i into a string consisting of 1's and 0's (binary expansion of the numbers). The length of each binary string depends greatly on the accuracy that is needed. Bit string chromosomes are quite handy because they can represent everything. The bit string structure shared by all the chromosomes is called the genetic representation.

We use a string vector consisting of 1s and 0s as a chromosome to represent real values of the variable x_i. The string of genes (a gene can take different values or alleles. For example, a gene for eye color may have alleles black, brown, etc.) that completely specifies a potential solution is known as a chromosome.

The domain of each variable x_i has length [dh –dl] where dh and dl are respectively the upper and the lower bounds of the domain in which the value of any variable x_i exists. Here, the precision requirement is denoted as xp decimal digits. The precision requirement implies that the interval (range) [dh –dl] should be divided into at least [dh –dl]*10xp equal sized subintervals.

The initial population is formed by selecting individuals (chromosomes) where their values are evenly distributed between dl and dh. The population size is determined based on the number of independent variables and accuracy.

Recombination Recombination or sexual reproduction, is a key operator for natural evolution. Technically, it takes two genotypes (The word genotype is used to describe the set of genes for a particular individual) to produce a new genotype by mixing the genes found in the originals. In biology, the most common form of recombination is crossover. In a crossover, two chromosomes are cut at one point and the halves are spliced to create new chromosomes. The crossover operation depends on how good the individual (chromosome) is at competing in its environment.

Mutation Mutation, sometimes called a background operator, changes a bit/character in a chromosome in a random way and is used usually sparingly. The reproductive operators — recombination and mutation — work at the level of chromosomes. Reproduction may take place due to one or multi-point crossovers and occasionally by (bit-wise) mutation.

Some GAs use a simple fitness function model to select stochastically individuals to undergo genetic operations such as crossover (genetically different offspring) or asexual reproduction (genetically unaltered offspring). Other GAs use a model in which certain randomly selected individuals in a subgroup compete and the fittest is selected. Different breeding techniques such as tournament selection, roulette wheel selection, and selection methods based on a fitness rate are applied to breed a new generation.

GA for an n-variable function: parameters, operators, and perturbations In view of enormous computing power (over one billion flops), we divide the whole n-dimensional region into k coarse n-dimensional sub-regions. We scan each coarse n-dimensional sub-region using a GA and then identify those of these sub-regions which are suspected to have a global minimum. This sub-region based search is a simpler form of the n-dimensional bisection based GA [22]. It is important that for each of the n variables, the real interval in which the required value of the variable exists should be as narrow (small) as possible. That is, our initial n dimensional search domain should be as small as reasonably possible. This is to avoid too much unnecessary computation and possibly to obviate missing a potential global minimum.

To achieve this, we first identify all the parameters and the genetic operators. We then decide on a plan how to perturb the parameters so that the distance between two solutions, viz., (||best GOS – worst GOS|| in each subspace) comes down as much as possible, where GOS denotes a/the global optimal solution. These perturbations will give us deeper insight into the character/nature and the possible location of the GOS. The parameters are (i) the n dimensional search space, (ii) the size of a member (chromosome) of the population, i.e., size of a solution vector having n elements, and (iii) the size of the population, i.e., the number of candidate solution vectors. The genetic operators which we advocate to perturb here are (i) crossover and (ii) mutation. There are many possible ways of implementing/perturbing these operators and deciding their probabilities based on the outcome/result. The sole motive of any perturbation in the probability of crossover, the probability of mutation, or the population size is to increase the fitness value, i.e., to proceed toward the actual GOS. In fact, the perturbations will also improve enormously our confidence in that the true GOS has not eluded and escaped from our search.

The parameter (ii), viz., the size of a member depends on the number of significant digits/decimal places accuracy for the solution and the given dimension n, Once we decide on the accuracy required/desired, we have no scope to perturb this parameter. For example, if the interval in one dimension/variable x is $[a, b]$ and if we need 5 decimal places accuracy then the domain of the variable x has length $L = (b - a)$ and the interval should be divided into at least $L \times 10^5$ equal size subintervals. If $b = 4$ and $a = -3$ then we need 20 bits for the binary vector (chromosome) since $524288 = 2^{19} \leq 7 \times 10^5 \leq 2^{20} = 1048576$. The mapping from a binary string $(b_{19}b_{18} \ldots b_1)$ to a real (decimal) number x from the interval $[a, b]$ is $x = a + [x'(b - a)/(2^{20} - 1)]$, where $x' = (\sum_{i=0}^{19} b_i \times 2^i)_{10}$. Now if we have n variables then our binary vector (string) will be at least n times as large to represent a member (chromosome) of the population.

We are now left with two other parameters (i) and (iii), viz., the search space and the population size. Interestingly both these parameters are related. For the given search space, we can perturb the population size, possibly increasing step by step the size starting from an arbitrary small value to a large value so that the difference between the worst GOS and the best GOS tends to vanish. Or, we can keep the population size fixed (constant and not relatively large) throughout and divide the search/solution space into subspaces (sub-regions). We then apply the GA for each subspace and identify that subspace which would contain the required GOS assuming the existence of only one GOS in the given original search space. The rest of the subspaces are removed. Thus our search space is significantly reduced. In case there are two or more doubtful subspaces where the GOS is likely, then we keep these subspaces while removing other subspaces. This later situation could arise if our population size is not sufficiently large. Although the genetic operators and their probabilities could contribute to some extent to our doubt, the main contributor is the population size. However, our main motive is to quickly and confidently reduce the search space and consequently we will keep population size unaltered throughout for any subspace and its further subdivision to track down the GOS. We will play, if necessary, with the genetic operators by varying their implementations and changing their probabilities to see if there is an improvement.

Preprocessing procedure for parameter values We first identify the parameters whose values we will determine by actually running the GA for the given problem a number of times. It is not out of place to mention that due to enormous processing power/speed[3] available to us, where most of this power is unutilized, focusing too much on relative time complexity (minor differences in time of computation) is wasteful. Executing a GA a number of times and using the results still keeps the complexity polynomial-time and is desirable.

Search space, population, and its member sizes Two parameters, viz., the population size and the size of the search space are vital. The size (number of bits) of a member (chromosome) of the population is predetermined based on the accuracy needed. However, if too much of accuracy is required, then this

[3] Every 18 months processor speed is doubling, Every 12 months band-width is doubling while every 9 months hard disk space is doubling. Currently, in many commercially available computers, we may execute over a billion floating-point operations per second (flops). On the extreme as of now (2007) the processing speed has touched 36 billion flops. The storage (CPU registers, cache, main executable memory, hard disk) sizes and their retrieval speed also have proportionately increased and are increasing. Since for higher processing power, higher storage space is a must to avoid any bottle neck in overall processing speed., we do have terabyte auxiliary storage space currently. The next goal is to achieve peta flops (peta $=10^{15}$) speed. Is there a limit beyond which the speed cannot be increased? Indeed there is a limit set by the speed of light barrier, thermal efficiency barrier, as well as quantum barrier do limit the computational power [9]. The number of protons in the universe is estimated to be around 10^{73}. If the whole universe is dedicated to information processing, then no more than 7.8996×10^{103} bits per year can be processed [9]. However, we are still too too far away from these extreme speed which appears to be a universal maximum in silicon technology.

might either not be possible due to precision limitation of the computer or would increase the search density too much and consequently the time complexity. Further, real world implementation of these highly numerically accurate quantities may not be possible as any measuring device cannot, in general, give an accuracy more than .005% [5]. Hence, it is enough if our accuracy of a quantity to be used in the real world environment is just four significant digits. Thus we take the member size such that it gives at most four significant digit accuracy and at least that much accuracy (usually 2 to 4 significant digits) required in the given problem. Thus the member size is practically fixed unless the concerned quantity is not an intermediate one. For an intermediate quantity, however, one should have more than four significant digit accuracy so that the terminal quantity to be used in the physical world has four significant digit accuracy.

Three or less variable functions: Generating graphs to reduce given search space For a given multi dimensional constrained/unconstrained optimization problem involving real numbers, we assume that for the given problem, we are not having any knowledge about the character of the problem to start with. For one, two, and three dimensional problems (given in an analytical form, not in a tabular form) which may be considered small but nontrivial and may crop up in many practical applications, one may get the graph using the Matlab commands and get considerable information about the nature of the problem, e.g., about how frequently/closely the function is fluctuating. It may be noted that a graph is just a crude form/representation of the actual required numerical values. Thus the graph will at best tell us about search space and may help us reducing the search space for global optimization to an extent.

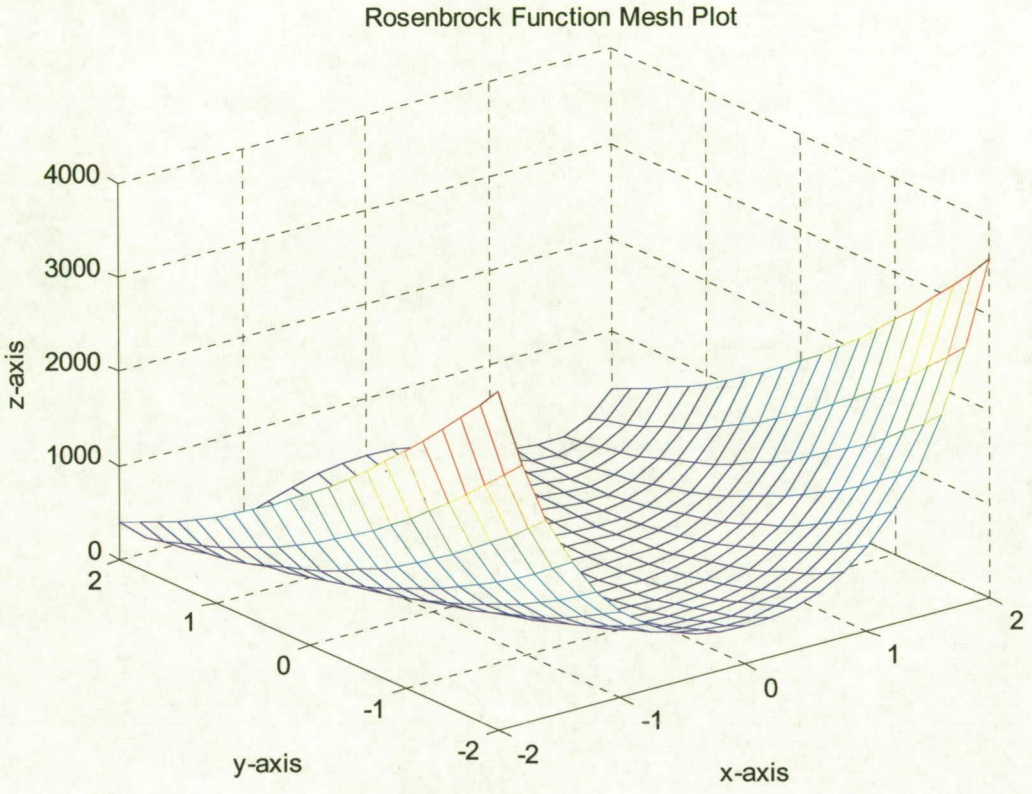
Four or more variable functions: bisecting space to reduce search space For four or more variable functions such a generation of graphs is impossible. We have two dimensional paper that can be used to represent two dimensional graphs. Since we, the common human beings, are capable of visualizing three dimensional objects, a three dimensional graph drawn (using projective geometry) on a two dimensional paper can be visualized.

3. Numerical Experiments

We have considered several test functions, viz., (i) Ackley, (ii) Dixon and Price, (iii) Hartmann, (iv) Griewank, (v) Levy, (vi) Michalewicz, (vii) Perm, (viii) Powell, (ix) Power sum, (x) Rastrigin, (xi) Rosenbrock, (xii) Schwefel, (xiii) Shubert, (xiv) Trid, and (xv) Zakharov functions. However, to conserve space, we present here the computation of a GOS for only Rosenbrock and Griewank functions using the gradient and genetic algorithms. This computation demonstrates that for most real world applications, a GA/an evolutionary approach is more desirable over the deterministic gradient algorithms from the accuracy point of view. This is because a GA needs to compute only functions and not derivatives which could often involve more error in computation. The gradient algorithms include the popular Matlab multivariable function minimization **fminsearch** that uses Nelder-Mead deterministic downhill simplex method. The probability of a gradient algorithm getting stuck at a local minimum giving an impression that it could be a global minimum is higher than that of a GA. Further, the GOS in a gradient method usually depends on the choice of an initial approximation for a multivariable function while the GOS in a GA usually depends on the search domain and the population size. Specifically, the accuracy of GOS in a GA also depends on the length of an element (chromosome) in the population.

Example 1 Rosenbrock function

$$Minimize: f(x) = \sum_{i=1}^{n/2} 100(x_{2i} - x^2{}_{2i-1})^2 + (1 - x_{2i-1})^2 \quad i = 1(1)n, \; lowerbound \le x_i \le upperbound$$



Comparisons (2 independent variables)

Rosenbrock Function

	Y	***x1***	***x2***
Genetic Algorithm	0	1	1
Gradient Algorithm	0.192964	1.437185	2.061217
MatLab fminsearch()	0	1	1

Griewank function

$$f(x) = \sum_{i=1}^{n} \frac{x_i^{\,2}}{4000} - \prod_{i=1}^{n} \cos(x_i / \sqrt{i}) + 1.$$

Comparisons (2 independent variables)

Griewank Function

	y	*x1*	*x2*
Genetic Algorithm	0.028448	9.409260	13.306537
Gradient Algorithm	1.049438	3.255481	15.355896
MatLab fminsearch()	0.496300	0.049900	0.099200

MatLab Method MatLab Algorithm with Rosenbrock function

y = 8.1777e-010 x = [1.0000 1.0000]t

Gradient Method Gradient method (Davidon-Fletcher-Powell Algorithm) with Rosenbrock function

y = 0.192964 x = $[1.437185 \quad 2.061217]^t$

Number of Generations	Genetic Algorithm 2^n *2 Subspace, 8 generations (Rosenbrock Function with 2 variables)					
	Range =[-2.0 -1.5]			**Range =[-1.5 -1.0]**		
	y	**x1**	**x2**	**y**	**x1**	**x2**
1	1424.274202	-1.503906	-1.503906	415.034095	-1.003906	-1.019531
2	1457.047282	-1.515623	-1.511717	415.034095	-1.003906	-1.019531
8	1424.261985	-1.503901	-1.503904	408.720515	-1.003904	-1.003905
	Range =[-1.0 -0.5]			**Range =[-0.5 0.0]**		
	y	**x1**	**x2**	**y**	**x1**	**x2**
1	59.692025	-0.503906	-0.503906	1.115313	-0.046875	-0.011719
2	62.761261	-0.515624	-0.511719	1.013954	-0.003906	-0.007813
8	59.691734	-0.503905	-0.503905	1.021884	-0.007812	-0.007812

	Range =[0.0 0.5]			Range =[0.5 1.0]		
	y	x1	x2	y	x1	x2
1	0.990113	0.011719	0.011719	0.012785	0.972656	0.957031
2	0.990113	0.011719	0.011719	0.005824	0.976563	0.960937
8	0.486389	0.312500	0.085938	0.000396	0.980468	0.960936
	Range =[1.0 1.5]			Range =[1.5 2.0]		
	y	x1	x2	y	x1	x2
1	0.000000	1.000000	1.000000	8.160156	1.500000	1.968750
2	0.000000	1.000000	1.000000	8.160156	1.500000	1.968750
3	0.001072	1.031250	1.062500	15.513900	1.503906	1.871094
7	0.000250	1.015625	1.031250	8.386331	1.503906	1.976560
8	0.000253	1.015625	1.031189	8.386331	1.503906	1.976560

MatLab Method MatLab Algorithm with Rosenbrock function

y = 8.1777e-010 x = [1.0000 1.0000]^t

Gradient Method Gradient method (Davidon-Fletcher-Powell Algorithm) with Rosenbrock function

y = 0.192964 x = [1.437185 2.061217]^t

Number of Generations	Genetic Algorithm Rosenbrock Function with 2 variables		
	Y	x1	x2
1	0.453125	1.250000	1.500000
2	0.169468	1.156250	1.375000
4	0.001072	0.968751	0.937501
5	0.040039	1.125000	1.281250
12	0.005434	0.937501	0.875000
13	0.001072	0.968752	0.937503
14	0.068836	0.938481	0.906251
15	0.005432	0.937502	0.875004
39	0.544158	0.281291	0.062507
40	0.660106	0.250006	0.031260

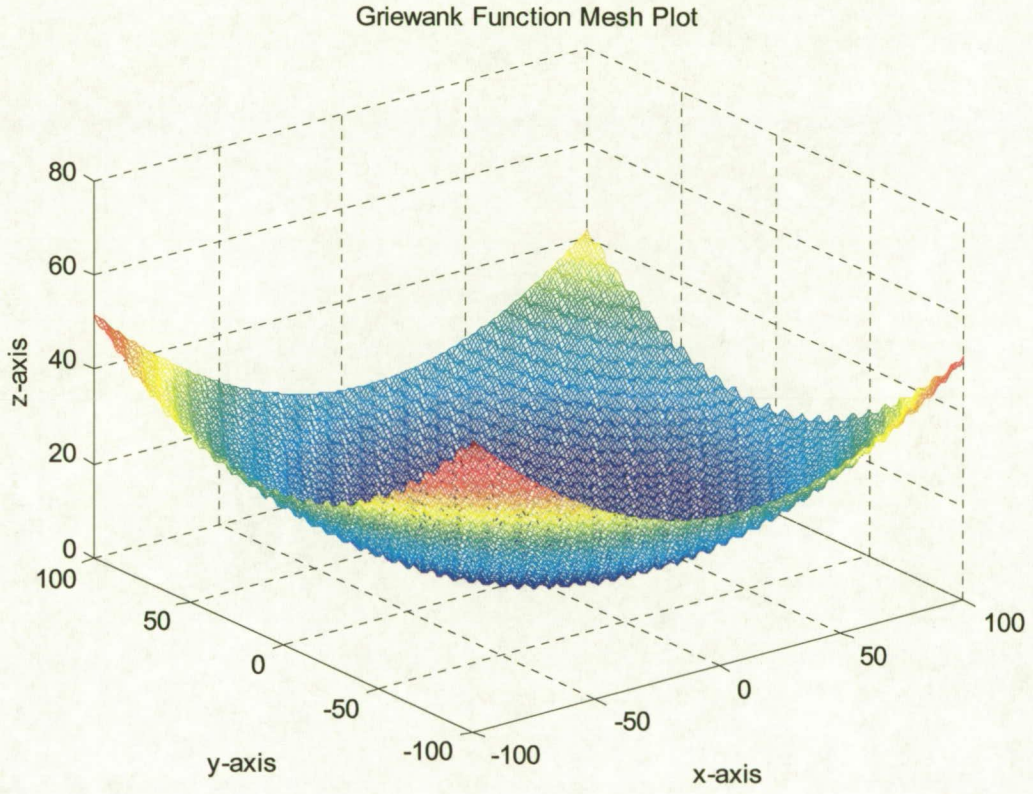
Griewank function

$$Minimize: f(x) = \frac{1}{4000}\sum_{i=1}^{n}(x_i - 100)^2 - \prod_{i=1}^{n}\cos(\frac{x_i}{\sqrt{i}}) + 1 \quad i = 1...n$$

$$lowerbound \le x_i \le upperbound$$

Gradient for n=2:

$$gradient = \begin{bmatrix} \frac{1}{200}x_1 - \frac{1}{20} - \sin(x_1) * \cos(\frac{x_2}{2} * \sqrt{2}) \\ \frac{1}{200}x_2 - \frac{1}{20} - \frac{1}{2}\cos(x_1) * \sin(\frac{x_2}{2} * \sqrt{2} * \sqrt{2}) \end{bmatrix}$$



Griewank Function (2 variables)

MatLab Method: MatLab Algorithm with Griewank Function (2 variables)

y = 0.4963 x = $[0.0499 \quad 0.0992]^t$

Gradient Method: Gradient method (Davidon-Fletcher-Powell Algorithm) with griewank function

y = 1.049438 x = $[3.255481 \quad 15.355896]^t$

Number of Generations	Genetic Algorithm /2^n*2 subspace, 8 generations (Griewank Function with 2 variables)					
	Range [-100 -75]			**Range [-75 -50]**		
	Y	**x1**	**x2**	**y**	**x1**	**x2**
1	38.243549	-75.195313	-75.195313	19.741632	-51.367188	-50.195313
2	38.243549	-75.195313	-75.195313	18.985257	-50.195313	-52.539063
8	37.715221	-77.734375	-75.195030	18.985252	-50.195257	-52.538912
	Range [-50 -25]			**Range [-25 0]**		
	Y	**x1**	**x2**	**y**	**x1**	**x2**
1	6.634640	-25.585938	-25.781250	1.170814	-0.976563	-0.976563
2	6.634640	-25.585938	-25.781250	1.170814	-0.976563	-0.976563
3	6.787624	-25.781250	-25.585938	0.804812	-0.195294	-0.976545
4	6.427955	-25.195276	-26.171839	0.748442	-0.585919	-0.390607
5	6.427955	-25.195276	-26.171839	0.748442	-0.585919	-0.390607
6	6.484920	-25.195276	-25.781214	0.642016	-0.195276	-0.585919
7	6.447919	-25.195276	-25.976527	0.642016	-0.195276	-0.585919
8	6.447919	-25.195276	-25.976527	0.585960	-0.195283	-0.390607
	Range [0 25]			**Range [25 50]**		
	Y	**x1**	**x2**	**y**	**x1**	**x2**
1	0.204940	3.125000	4.296875	1.674764	25.390625	25.195313
2	0.134078	9.765625	4.492188	1.674764	25.390625	25.195313
3	0.067623	6.445313	9.179688	1.311425	25.000000	26.953124
4	0.118047	9.179688	4.296875	1.311425	25.000000	26.953124
7	0.069626	9.179688	13.085938	1.269845	25.195309	26.367084
8	0.087676	9.375000	4.687500	1.283780	25.195309	26.757809
	Range [50 75]			**Range [75 100]**		
	Y	**x1**	**x2**	**y**	**x1**	**x2**
1	8.832051	50.585938	52.539063	22.983011	75.000000	75.000000
2	8.786172	50.000000	52.148387	22.983011	75.000000	75.000000
3	8.803204	50.195312	52.148434	22.784103	76.953116	75.000000
4	8.750069	50.195307	52.343750	22.663461	77.343745	75.390433
5	8.693107	50.000000	53.125000	22.385830	77.929681	75.292873
6	8.680684	50.000000	52.734371	22.338780	78.125000	75.195312
7	8.680684	50.000000	52.734371	22.338780	78.125000	75.195312
8	8.726663	50.000000	53.320303	22.365357	77.929490	75.195312

Griewank Function (2 variables)

MatLab Method MatLab Algorithm with Griewank Function (2 variables)

y = 0.4963 x = $[0.0499 \quad 0.0992]^t$.

Gradient Method Gradient method (Davidon-Fletcher-Powell Algorithm) with Griewank function

y = 1.049438 x = $[3.255481 \quad 15.355896]^t$.

Number of Generations	Genetic Algorithm Griewank Function (2 variables)		
	y	x1	x2
0	1.175222	14.062500	18.750000
1	1.175222	14.062500	18.750000
2	0.095894	6.250000	9.375000
35	0.077939	12.500311	9.375164
36	0.077915	12.500207	9.375041
64	0.077064	12.524940	9.375417
65	0.029676	9.375275	13.281601
108	0.028621	9.402112	13.282249
109	0.028448	9.409260	13.306537
110	0.028531	9.406088	13.281738
127	0.028494	9.406204	13.288153
128	0.028520	9.406580	13.281864

Griewank Function with 7 variables

Comparisons (7 independent variables)

Griewank Function

	y	*x1*	*x2*	*x3*	*x4*	*x5*	*x6*	*x7*
Genetic Algorithm	0.250326	9.656299	8.437529	11.343793	11.909241	6.750028	8.250042	8.062541
Gradient Algorithm	1.425241	10.442063	2.537317	9.198368	-0.439168	10.050415	9.125494	9.895116
MatLab fminsearch()	1.715300	0.051500	0.102400	0.152700	0.202300	0.251500	0.300000	0.348000

MatLab Method MatLab Algorithm with Griewank function (7 variables)

y = 1.7153 x = $[\,0.0515 \quad 0.1024 \quad 0.1527 \quad 0.2023 \quad 0.2515 \quad 0.3000 \quad 0.3480\,]^t$.

Gradient Method Davidon-Fletcher-Powell Algorithm with Griewank function (7 variables)

y = 1.425241

x = [10.442063 2.537317 9.198368 -0.439168 10.050415 9.125494 9.895116]t

Number of Generations	Genetic Algorithm (Griewank Function with 7 variables)							
	Y	x1	x2	x3	x4	X5	x6	x7
0	0.979566	10.031250	10.031250	10.031250	10.031250	10.031250	9.000000	10.031250
1	1.053514	7.031257	7.781250	7.781250	10.312500	7.781250	10.500000	7.781250
15	0.852372	8.531280	8.906270	8.906257	11.531265	7.218786	7.031271	9.093758
58	0.782795	8.906308	8.437557	11.156316	10.781293	8.343795	7.687555	10.781294
59	0.250326	9.656299	8.437529	11.343793	11.909241	6.750028	8.250042	8.062541
60	0.749432	8.531307	8.531271	11.812572	11.906310	8.718823	8.343805	9.843824
128	0.784087	9.187621	9.283004	9.750172	11.906321	9.187594	9.844045	9.468889

Rosenbrock Function with 8 variables

Comparisons (8 independent variables)
Rosenbrock Function

	y	x1	x2	x3	x4	x5	x6	x7	X8
Genetic Algorithm	1.227185	1.015625	1.015625	0.953125	1.015625	1.015625	1.015625	1.015625	1.015625
Gradient Algorithm	y = NaN								
MatLab fminsearch()	3.974200	2.046300	4.190900	0.002300	0.001600	2.247400	5.053500	0.429700	0.182400

MatLab Method MatLab Algorithm with Rosenbrock function

x = [2.0463 4.1909 0.0023 0.0016 2.2474 5.0535 0.4297 0.1824]^t, y = 3.9742

Gradient Method Davidon-Fletcher-Powell Algorithm

x = [-1.144971 0.966277 0.022287 1.140769 -0.822340 1.160610 -0.909875 0.500168]^t,
y=188.628060
y = NaN
x = NaN NaN NaN NaN NaN NaN NaN NaN

Number of Generations	Genetic Algorithm (Rosenbrock Function with 8 variables)								
	y	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**
0	1.227185	1.015625	1.015625	0.953125	1.015625	1.015625	1.015625	1.015625	1.015625
1	10.984105	0.437501	0.218750	0.218750	0.218750	0.218750	0.218750	0.218750	0.218750
2	13.074403	-0.953125	0.953126	0.343750	0.234377	0.343751	0.343750	0.687500	0.343750
16	18.165923	0.671877	0.453126	0.609375	0.203126	0.484377	0.000002	-0.499998	-0.015621
124	2.895730	0.281265	0.046887	0.203137	0.031264	0.063491	0.000051	0.140640	0.031267
128	3.073391	0.173841	0.046888	0.171898	0.016618	0.140635	0.031262	0.046890	0.000010

Run-2 Rosenbrock Function with 8 variables

Comparisons (8 independent variables)
Rosenbrock Function

	y	*x1*	*x2*	*x3*	*x4*	*x5*	*x6*	*x7*	*x8*
Genetic Algorithm	1.636273	0.437515	0.203137	0.312522	0.125015	0.515638	0.281267	0.421889	0.218765
Gradient Algorithm	y =	NaN							
MatLab fminsearch()	3.974200	2.046300	4.190900	0.002300	0.001600	2.247400	5.053500	0.429700	0.182400

MatLab Method MatLab Algorithm with Rosenbrock function

x = [2.0463 4.1909 0.0023 0.0016 2.2474 5.0535 0.4297 0.1824]^t, y = 3.974

Gradient Method Davidon-Fletcher-Powell Algorithm

y=188.628060
x = [-1.144971 0.966277 0.022287 1.140769 -0.822340 1.160610 -0.909875 0.500168]t,
y = NaN
x = NaN NaN NaN NaN NaN NaN NaN NaN

Number of Generations	Genetic Algorithm (Rosenbrock Function with 8 variables)								
	y	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**
0	4.589105	-0.015625	-0.015625	-0.015625	-0.015625	-0.015625	0.062500	-0.015625	-0.015625
1	55.671459	0.828125	1.281250	-0.218749	-0.156250	-0.843750	0.828125	0.718750	0.828125
2	44.921494	-0.359374	0.328125	0.328125	0.328125	0.390626	0.328125	-1.390625	1.437500
24	18.908493	0.156250	0.390628	0.796878	0.796877	0.187501	0.109377	0.078128	0.031252
112	3.239470	0.304698	0.078133	0.343760	0.140636	0.204120	0.140881	0.203136	0.039076
125	2.621539	0.296886	0.125012	0.265641	0.046881	0.218760	0.078143	0.265637	0.031259
126	1.636273	0.437515	0.203137	0.312522	0.125015	0.515638	0.281267	0.421889	0.218765
127	2.505399	0.437509	0.218764	0.234387	0.031504	0.265637	0.093762	0.484388	0.156263
128	3.563958	0.312506	0.080092	0.109389	0.046886	0.054702	0.015642	0.454124	0.109391

Table 1 – Genetic Algorithm N-Subspace

```
function [] = genetic_algorithm(function_handle,x,master_range)

 ga_history=[; ]; keep_min_value=[];
 valriable_size=size(x);
 subinterval_matrix=[]; range=[];

 createRangeMatrix()

 iterationCounts = inputNumberOfIterations();
 sub_intervals=1;
 filename1= strcat('ga_data_',num2str(sub_intervals));
 filename =strcat(filename1,'.dat');
 fid = fopen(filename,'w');
 fprintf (fid,' Genetic Algorithm\n');
 fprintf (fid,' Y              x1              x2       ....\n');
 for sub_intervals=1:2^valriable_size(2)*2
  range(1,1) = subinterval_matrix(sub_intervals,1);
  range(1,2) = subinterval_matrix(sub_intervals,2);
  [x y] = ga_minimize(function_handle,x,range,fid,iterationCounts,ga_history);
 end
 disp('*****************************************');
 % ga_history(1..2)= range
 % ga_history(3) = y
 % ga_history(4..n)=x
 ga_history
 m=min(ga_history,3)

 fclose(fid);

% utility function

disp('*****************************************');

function iterationCounts = inputNumberOfIterations()
% This function returns number of iterations required.
 iterationCounts =input('How many iterations? ');
 disp(' ');
 if(iterationCounts<1) iterationCounts=1; end;
end;

function [] = createRangeMatrix()
% Define subintervals based on the number of variables
% subspaceincrement = (abs(range(1))+abs(range(2)))/ (K1(2)*2^K1(2));

  subspacerange = (abs(master_range(1))+ ...
                  abs(master_range(2)))/ (2^valriable_size(2)*2);

  L= master_range(1);
  for i=1:2^valriable_size(2)*2
   subinterval_matrix(i,1) = L;
   L=L+ subspacerange;
```

```
  end

  L= master_range(1)+subspacerange;
  for i=1:2^valriable_size(2)*2
    subinterval_matrix(i,2) = L;
    L= L + subspacerange;
  end
end
end  % genetic_algorithm(function_handle)
```

Table 2 – Genetic Algorithm Computations

```
%
% Dr. Sen and Gholam Ali Shaykhian
% Procedure:
% Step-1 Randomly select n chromosomes
% Step-2 Calculate the fitness function for each chromosome
% Step-3 Prepare the mating pool:
%       Sort the fitness functions, replace the weak 12.5% chromosomes
%       with the strong 12.5% chromosome.
% Step-4 Perform Crossover:
%       (a) Randomly select a number between 1 to size(chromosome)
%       (b) Perform crossover, chromosome-1 is crossovered with
%       chromosome-n-1, chromosome-2 is crossovered with chromosome-n-2,
%       chromosome-3 is crossovered with chromosome-n-3, and .....
% Step-5 Calculate the fitness function
% Step-6 Randomly select a chromosome, then rondomly mutate one bit in that
%       chromosome, (if the random bit is '1' change to '0', and vice  versa.
% Step-7 Increment generation count (iterationCounts)
% Step-8 Is iterationCounts equal to number of generations, then goto
%       Step-9 else goto Step-2
% Step-9 Show report
%
function [x y] = ga_minimize(function_handle, ...
                             x, ...
                             intervalRange, ...
                             fid, ...
                             iterationCounts, ...
                             ga_history)
%
% Genetic Algorithm, the first required task is to code the parameter
% set. The coding is generated from a string of 1's and 0's; the string is
% to represent the independent variables [x1,x2, x3, ..., xn].
% The values of the variable x1, x2, x3, .... are represented as a
% binary vector; a chromosome. The length of the vector depends on the
% required precision.
% In this example, we use six places after the decimal point. The domain
% of the variable xi is determined by evaluating the parameter
% intervalRange. For example, if x1 has a length of 15.1 (-3,12.1); the
% coding uses the range by dividing 15.1 into at least 15.1*1000000 equal
% sizes. This means that 24 bits are required for each binary vector
% (chromosome) in population:
%           8388608 =2^23<15100000<=2^24=16777216.
```

```
%
% initialize vector and matrix and scalar variables
temp_history=[; ]; crossOverMatrix=[ ; ];
B1=getRequiredPrecision();
K1=size(x); chromosomeCount =0; row_history=1;

createCrossOverMatrix();
keepHistory(1); % keep initial best fit

for CNT=1:iterationCounts % multiple generations
 row_history=1; % rest the index of temp_history
 CrossOver();
 keepHistory(CNT+1)
end
y=min(temp_history(1,:));
disp('*****************************************************');
disp(sprintf('\n\n\t\t\t\t\t\t DONE !!!!'));
%
% ========================================================================
%
%                  Utility functions
%
% ========================================================================


function [] = keepHistory(history_index)
   %
% keep the minimum of each generation
% Finds the indices of the minimum values of temp_history(), and returns
% them in output vector I. If there are several identical minimum values,
% the index of the first one found is returned.
%
 [C,I] = min(temp_history(:,1));
 R=[];
 R(1)=intervalRange(1,1);
 R(2)=intervalRange(1,2);
 R(3)= C;

 tm =2;
 for km=4:K1(2)+3 % number of variables
    R(km)=temp_history(I,tm);
    tm = tm +1;
 end
 ga_history(history_index,:)= R;
 sR=size(R);
 sR
 R
 for iR=1:sR(2)
   fprintf(fid,' %12.6f',R(iR));
 end
 fprintf(fid,'\n');
end

function [] = CrossOver()
%
% Cross over and reproduction-Individual strings are copied according to
```

```
% their objective function. Survival of fittest-The greater fitness an
% element has, the greater is its chance of being used for reproduction. We
% will duplicate 12.5% of strong chromosomes for reproduction. The weak
% 12.5% of chromosomes are replaced with the strongest chromosomes before
% crossover operations.
% Crossover: For each pair of chromosomes, a random number between 1 to
% the size of chromosome is selected. Let n to be the size of the
% chromosome, and d to be the random number selected. All
% genes from d to n are swapped (crossover). The chromosome i is
% paired with chromosome population-i.
%
 crossOverMatrix = sortrows(crossOverMatrix);
 performCrossOver();

 % Select a chromosome at random for mutation
 p = randperm(chromosomeCount); d2=p(1);
 performMutation(d2)

 % recalculate the best fit after cross over

  for i=1:chromosomeCount
    paramX=crossOverMatrix(i,:);
    % calculate the initial fitness values
    crossOverMatrix(i,1)= calculateBestFits(paramX);
  end
end



function [] = performCrossOver()
%
% Mating pool--The mating pool is constructed by replacing the lowest 12.5%
% fitness values (x1, x2, x3,...,xn) with the population strong 12.5%
% values. After replacement, all chromosome are randomly shuffled.
%
  count=chromosomeCount/8;  % crossover 12.5%
  mm=chromosomeCount;
  binaryValue1= dec2bin(0, B1);
  binaryValue2= dec2bin(0, B1);
  binaryValue3= dec2bin(0, B1);
  binaryValue4= dec2bin(0, B1);
  n=B1-6; %crossover at least 6 bits
  p = randperm(n);
  d=p(1);

  nn=size(binaryValue1);

  for i=1:count
     for j=2:K1(2)+1
       binaryValue1 = convertValueToBinary(crossOverMatrix(i,j));
       binaryValue2 = convertValueToBinary(crossOverMatrix(mm,j));

       binaryValue3= strcat(binaryValue1(1:d),binaryValue2(d+1:nn(2)));
       % binaryValue4= strcat(binaryValue1(11:nn(2)),binaryValue2(1:10));
       % change the weak chromosome
       crossOverMatrix(mm,j)= convertChromosomeToDecimal(binaryValue3);
```

```
      mm= mm-1;
    end
  end
  shuffleCrossOverMatrix();
end


function [] = performMutation(d2)
%
% Conduct mutation at random selection for each generation
% A gene within a chromosome is mutated at random position
%
 gene = randperm(B1);
 genePosition=gene(1); % gene position to mutate

 for j=2:K1(2)+1
   mbinaryValue = convertValueToBinary(crossOverMatrix(d2,j));
   if mbinaryValue(:,genePosition) =='0'
     mbinaryValue(:,genePosition)='1';
   else
     mbinaryValue(:,genePosition)='0';
   end
   crossOverMatrix(d2,j)= convertChromosomeToDecimal(mbinaryValue);
 end
end


function [] = createCrossOverMatrix()
%
% Set up the initial population
%
  rangeValue=intervalRange(1);
  chromosomeCount = getNumberOfChromosome();
  incrementValue = (abs(intervalRange(1))+ ...
                abs(intervalRange(2)))/ chromosomeCount;

  for i=1:chromosomeCount
    for j=2:K1(2)+1 %number of variables
     crossOverMatrix(i,j)=rangeValue;
    end
    paramX=  crossOverMatrix(i,:);
     % calculate the initial fits
    crossOverMatrix(i,1)= calculateBestFits(paramX);
    rangeValue=rangeValue+incrementValue;
  end
  shuffleCrossOverMatrix();
end

function [] = shuffleCrossOverMatrix()
%
%   shuffle the chromosomes in population randomly
%
  p = randperm(chromosomeCount);
  for i=1:chromosomeCount
   T =crossOverMatrix(i,:);
```

```
    crossOverMatrix(i,:) = crossOverMatrix(p(1,i),:);
    crossOverMatrix(p(1,i),:)=T;
  end
end

function y = calculateBestFits(paramX)
%
% The function handle provides a means of calling a function indirectly.
% The function handle is created by the calling program and is passed to
% ga_minimize() as parameter.
%
  pos=K1(2)+1;
  y=function_handle(paramX(2:pos));
  paramX(1,1)=y;

%   for varCNT=1:pos
%     fprintf(fid,' %12.4f',paramX(1,varCNT));
%   end
%   fprintf(fid,'\n');
  temp_history(row_history,:)=paramX;
  row_history= row_history+ 1;
end

function binaryValue = convertValueToBinary(val)
clear binaryValue;
%
% Genetic algorithm coding: The first required task is to code the
% parameter set (x1, x2, x3, ...xn). The coding is generated from a string
% of 1's and 0's; the string is to represent the parameter xi.
% The values of each variable xi is represented as a binary vector; a
% chromosome. The length of the vector depends on the required precision.
% This example uses six places after the decimal point precision. The
% domain of the variable xi is used to calculate the length of xi; for
% example, if the low intervalRange is -3, and high intervalRange is 12.1,
% the length is computed as 15.1; range [-3..12.1]; the same procedure is
% applied to all xi.
% The mapping from real value to binary value is as follow:
% (1) Determine the required precision
% (2) Let B1= number of bits needed for required precision
% (3) let intervalRange(1) and intervalRange(2) represent the domain of xi
% (4) Let rValue = real value of a chromosome
% (5) Convert rValue to chromosome (binary string)
%
  g =getRequiredPrecision();
  N1=2^g;
  rValue = (intervalRange(1)-val)*N1/(intervalRange(1)-intervalRange(2));
  binaryValue = dec2bin(rValue, g);
end

function precision = getRequiredPrecision()
%
% The precision requirement of six digits implies that each xi is  divided
% into at least 15.1*1000000 equal size ranges. To determine the string size
% (chromosome-number of genes), 8388608 =2^23<15100000<=2^24=16777216,
% means that 24 bits are required, a binary vector for each chromosome is
% constructed.
```

```
%
 percision = 0;
 requiredPrecision=dec2bin(1000000*(abs(intervalRange(1))+ ...
                          abs(intervalRange(2))));
 pL = size(requiredPrecision);
 precision = pL(1,2);
end

function realValue = convertChromosomeToDecimal(chromosome)
%
% For a 24 bits binary string (<b23b22 ...b0> ), the mapping from binary to real value is as follow:
% (1) Convert the 24-bit-binary to real value
% (2) Let intervalRange(1) and intervalRange(2) to be the low and high  value of the range of xi respectively
% (3) Let B1 = the number of bits (genes) in each chromosome
% (4) Let N1 = convert the binary string (chromosome) to decimal
% (5) Compute real value correspond to a chromosome use the equation:
%     realValue = intervalRange(1)+N1*(intervalRange(2)-  intervalRange(1))/((2^(B1(2))-intervalRange(1)
%

 N1 = bin2dec(chromosome);
    realValue = intervalRange(1)+N1*(intervalRange(2)- ...
            intervalRange(1))/((2^(B1)-intervalRange(1)));
end

function chromosomeCount = getNumberOfChromosome()
%
% This function returns the population size.
%
 if (K1(2)>5)
    chromosomeCount = 256;
  else
    chromosomeCount = 128;
  end;
end

end % ga_minimize(function_handle,x,intervalRange)
```

Table 3 – Rosenbrock Function (2 independent variables)

```
function [y, grad] = rosenbrock2(x)
%
% Rosenbrock banana function
%
% Initialize vector variables
%
grad=[]; xx1=[]; xx2=[]; xx1= x(1); xx2=x(2);

% declare symbolic variables- The symbolic variables are used by Jacobian
% return the gradient functions this is done by setting a variable equal
% to a symbolic expression, and then apply the syms command to the variable
%
syms x1 x2
```

```
% fitness function
y =100*(x2-x1^2)^2+(1-x1)^2;

% The gradient function is computed using the function jacobian in
% conjucction with symbols representing the independent variables in the
% fitness function. The jacobian takes two parameters; namely variable y
% and  a vector whose components are the variables of the function y.
%
gradf =jacobian(y,[x1,x2]);

% The inline constructs an inline function object corresponding to
% gradient; the vectorize vectorizes the formula for the gradient function.
%
gf1 = inline(vectorize(gradf(1))); gf2 = inline(vectorize(gradf(2)));

grad(1,1) = gf1(xx1,xx2); grad(1,2) = gf2(xx1,xx2);

y =100*(x(2)-x(1)^2)^2+(1-x(1))^2;
end
```

Table 4 – Rosenbrock Function (8 independent variables)

```
function [y, grad] = rosenbrock8(x)
%
% Rosenbrock banana function
%
% Initialize vector variables
%
grad=[]; xx1=[]; xx2=[];
xx1= x(1); xx2=x(2); xx3= x(3); xx4=x(4);
xx5= x(5); xx6=x(6); xx7= x(7); xx8=x(8);
% declare symbolic variables- The symbolic variables are used by Jacobian
% return the gradient functions this is done by setting a variable equal
% to a symbolic expression, and then apply the syms command to the variable
%
syms x1 x2 x3 x4 x5 x6 x7 x8

% fitness function
   y = 100*(x2-x1^2)^2+(1-x1)^2 + ...
      100*(x4-x3^2)^2+(1-x3)^2 + ...
      100*(x6-x5^2)^2+(1-x5)^2 + ...
      100*(x8-x7^2)^2+(1-x7)^2;
%
% The gradient function is computed using the function jacobian in
% conjucction with symbols representing the independent variables in the
% fitness function. The jacobian takes two parameters; namely variable y
% and  a vector whose components are the variables of the function y.
%
%
% gradient = [
%          -400*(x2-x1^2)*x1-2+2*x1,
%          200*x2-200*x1^2,
%          -400*(x4-x3^2)*x3-2+2*x3,
```

```
%           200*x4-200*x3^2,
%           -400*(x6-x5^2)*x5-2+2*x5,
%           200*x6-200*x5^2,
%           -400*(x8-x7^2)*x7-2+2*x7,
%           200*x8-200*x7^2]
%

gradf =jacobian(y,[x1,x2,x3,x4,x5,x6,x7,x8]);

% The inline constructs an inline function object corresponding to
% gradient; the vectorize vectorizes the formula for the gradient function.
%
gf1 = inline(vectorize(gradf(1))); gf2 = inline(vectorize(gradf(2)));
gf3 = inline(vectorize(gradf(3))); gf4 = inline(vectorize(gradf(4)));
gf5 = inline(vectorize(gradf(5))); gf6 = inline(vectorize(gradf(6)));
gf7 = inline(vectorize(gradf(7))); gf8 = inline(vectorize(gradf(8)));

grad(1,1) = gf1(xx1,xx2); grad(1,2) = gf2(xx1,xx2); grad(1,3) = gf3(xx3,xx4); grad(1,4) = gf4(xx3,xx4);
grad(1,5) = gf5(xx5,xx6); grad(1,6) = gf6(xx5,xx6); grad(1,7) = gf7(xx7,xx8); grad(1,8) = gf8(xx7,xx8);

   y =100*(x(2)-x(1)^2)^2+(1-x(1))^2 + ...
     100*(x(4)-x(3)^2)^2+(1-x(3))^2 + ...
     100*(x(6)-x(5)^2)^2+(1-x(5))^2 + ...
     100*(x(8)-x(7)^2)^2+(1-x(7))^2 ;
end
```

Table 5 – Gradient Algorithm (Davidon-Fletcher-P

```
% Davidon-Fletcher-Powell Gradient Algorithm (7 variables function)
% Dr. Sen / Gholam Ali Shaykhian
% March 2007
function [x y history] = dfp_minimize (function_handle,lambda_handle, x_init,fid)
% ------------------------------------------------
% Gradient Descent parameters
% ------------------------------------------------
max_iteration = 100; epsilon = 0.01;
lambda=0; x = x_init; K1=size(x);
history = [];
% keep data points for history and possible graphs (if dimensions allowed)
keepX=zeros(max_iteration,size(x,1));
keepY=zeros(max_iteration,1);

% ------------------------------------------------
% Step-1: set dk=-Hkgk as the search direction from the current point xk,
% where gk=gradient(f(xk), H0 = I and x0 is given at the start, viz k=0
% ------------------------------------------------
Hk = eye(size(x,1));
% s1=size(x)   s2=size(Hk)

val0 = lambda;
history = [history; x];
```

```
for mK=1:max_iteration
    % Evaluate y = f(x)
   [y, grad] = function_handle(x);

    fprintf(fid,'y = %12.6f   x =',y);
    for i=1:K1(2), fprintf(fid,' %12.6f',x(1,i)); end; fprintf(fid,'\n');

   d=-Hk*grad'; lambda = lambda_handle(x,d,val0);
   % ------------------------------------------------
   % Step-2 Perform a linear search to find lambda (>0), where lambda is
   % the value that minimizes f(x+lambda*dk)

   % Step-3 Set Sk=lambda * dk
   Sk = lambda * d;

   if (max(abs(Sk)) < epsilon)
      break;
   end;

   % Step-4 Set X(k+1)= X(k) =Sk
   x=x+Sk';

   % Step-5 Evaluate f(x(k+1)) and g(k+1)
   % Evaluate y = f(x)
   [y, current_grad] = function_handle(x);
   % Step-6 Ck= g(k+1)- g(k)
   Ck= (current_grad - grad)';

   % Step-7 Set Hk+1 = Hk+Ak+Bk
   Ak = (Sk * Sk')/(Sk'*Ck);

   Bk= -(Hk*Ck*Ck')/(Ck'*Hk*Ck);

   Hk = Hk+Ak+Bk;
   history = [history; x]
end
    % Evaluate y = f(x)
   [y, grad] = function_handle(x);

 fprintf ('    finished at iteration # %d\n', mK);
 for dim = 1:size(x,1),
   fprintf ('    f(x%d) = %d\n', dim,x(dim)); % display each dimension of final_xValues vector
 end
end
```

Table 6 – 2-varaiables Lambda Rosenbrock Calculation

```
function lambda = rosenbrockLambda2(x,d,val0)
 % Perform a linear search to find lambda, where lamda is the value which
 % minimizes the function.
 % y =100*(x(2)-x(1)^2)^2+(1-x(1))^2;
 f=@(L)100*((x(2)+d(2)*L)-(x(1)+d(1)*L)^2)^2 +(1-(x(1)+d(1)*L))^2;
```

```
 % fminbnd finds the minimum of a function of one variable within a fixed
 % interval.
 lambda=fminbnd(f,val0,val0+2);
end
```

Table 7 – 8-varaiables Lambda Rosenbrock Calculation

```
function lambda = rosenbrockLambda8(x,d,val0)
 % Perform a linear search to find lambda, where lamda is the value which
 % minimizes the function.
%    y =100*(x(2)-x(1)^2)^2+(1-x(1))^2 + ...
%       100*(x(4)-x(3)^2)^2+(1-x(3))^2 + ...
%       100*(x(6)-x(5)^2)^2+(1-x(5))^2 + ...
%       100*(x(8)-x(7)^2)^2+(1-x(7))^2 ;

 f=@(L)100*((x(2)+d(2)*L)-(x(1)+d(1)*L)^2)^2 +(1-(x(1)+d(1)*L))^2;
 % fminbnd finds the minimum of a function of one variable within a fixed
 % interval.
  f=@(L)100*((x(2)+d(2)*L)-(x(1)+d(1)*L)^2)^2 + (1-(x(1)+d(1)*L))^2 + ...
    100*((x(4)+d(4)*L)-(x(3)+d(3)*L)^2)^2 + (1-(x(3)+d(3)*L))^2 + ...
    100*((x(6)+d(6)*L)-(x(5)+d(5)*L)^2)^2 + (1-(x(5)+d(5)*L))^2 + ...
    100*((x(8)+d(8)*L)-(x(7)+d(7)*L)^2)^2 + (1-(x(7)+d(7)*L))^2;

  lambda=fminbnd(f,val0,val0+2);
end
```

Table 8 – 2-varaiables Lambda Griewank Calculation

```
function lambda = griewankLambda2(x,d,val0)
 % Perform a linear search to find lambda, where lamda is the value which
 % minimizes the function.

 % fminbnd finds the minimum of a function of one variable within a fixed
 % interval.
 % Griewank's Function
 %
 % f(x) = (1/400)*sum(Xi-10)^2-product(cos(Xi/sqrt(i)))+1  for i:1:n
 % let i=1:7
 % y= (1/400)*((x(1)-10)^2+(x(2)-10)^2+(x(3)-10)^2+(x(4)-10)^2+...
 %    (x(5)-10)^2+(x(6)-10)^2+(x(7)-10)^2)-...
 %    (cos(x(1)/1^0.5)*cos(x(2)/2^0.5)*cos(x(3)/3^0.5)*...
 %    cos(x(4)/4^0.5)*cos(x(5)/5^0.5)*cos(x(6)/6^0.5)*cos(x(7)/7^0.5))+1;

 f=@(L)(1/400)*((x(1)+d(1)*L-10)^2+(x(2)+d(1)*L-10)^2 - ...
    cos(x(1)+d(1)*L/1^0.5)*cos(x(2)+d(1)*L/2^0.5));

 lambda=fminbnd(f,val0,val0+2);
end
```

Table 9 – 7-varaiables Lambda Griewank Calculation

```
function lambda = griewankLamdda7(x,d,val0)
 % Perform a linear search to find lambda, where lamda is the value which
 % minimizes the function.

 % fminbnd finds the minimum of a function of one variable within a fixed
 % interval.
 % Griewank's Function
 %
 % f(x) = (1/400)*sum(Xi-10)^2-product(cos(Xi/sqrt(i)))+1  for i:1:n
 % let i=1:7
 % y= (1/400)*((x(1)-10)^2+(x(2)-10)^2+(x(3)-10)^2+(x(4)-10)^2+...
 %    (x(5)-10)^2+(x(6)-10)^2+(x(7)-10)^2)-...
 %    (cos(x(1)/1^0.5)*cos(x(2)/2^0.5)*cos(x(3)/3^0.5)*...
 %    cos(x(4)/4^0.5)*cos(x(5)/5^0.5)*cos(x(6)/6^0.5)*cos(x(7)/7^0.5))+1;

 f=@(L)(1/400)*((x(1)+d(1)*L-10)^2+(x(2)+d(1)*L-10)^2+(x(3)+...
    d(1)*L-10)^2+(x(4)+d(1)*L-10)^2+(x(5)+d(1)*L-10)^2+(x(6)+...
    d(1)*L-10)^2+(x(7)+d(1)*L-10)^2)-cos(x(1)+d(1)*L/1^0.5)*...
    cos(x(2)+d(1)*L/2^0.5)*cos(x(3)+d(1)*L/3^0.5)*cos(x(4)+...
    d(1)*L/4^0.5)*cos(x(5)+d(1)*L/5^0.5)*cos(x(6)+d(1)*L/6^0.5)*...
    cos(x(7)+d(1)*L/7^0.5)+1;

 lambda=fminbnd(f,val0,val0+2);
end
```

Table 10 – 2- Griewank Function

```
%
% Griewank's Function
% lower boundary <=Xi<=upper boundary
%
% f(x) = (1/400)*sum(Xi-10)^2-product(cos(Xi/sqrt(i)))+1  for i:1:n
%
function [y, grad] = griewank2(x)

% Initialize vector variables
%
grad=[]; xx1=[]; xx2=[];
xx1= x(1); xx2=x(2);

% declare symbolic variables- The symbolic variables are used by Jacobian
% return the gradient functions this is done by setting a variable equal
% to a symbolic expression, and then apply the syms command to the variable
%
syms x1 x2

% fitness function
y = (1/400)*(x1-10)^2+(1/400)*(x2-10)^2--cos(x1)*cos(x2/2^0.5)+1;

% The gradient function is computed using the function jacobian in
% conjucction with symbols representing the independent variables in the
% fitness function. The jacobian takes two parameters; namely variable y
```

```
% and  a vector whose components are the variables of the function y.
%
gradf =jacobian(y,[x1,x2]);

% The inline constructs an inline function object corresponding to
% gradient; the vectorize vectorizes the formula for the gradient function.
%
gf1 = inline(vectorize(gradf(1)));
gf2 = inline(vectorize(gradf(2)));

grad(1,1) = gf1(xx1,xx2);
grad(1,2) = gf2(xx1,xx2);

y = (1/400)*(x(1)-10)^2+(1/400)*(x(2)-10)^2-cos(x(1))*cos(x(2)/2^0.5)+1;

end
```

Table 11 – 7- Griewank Function

```
%
% Griewank's Function
% f(x) = (1/400)*sum(Xi-10)^2-product(cos(Xi/sqrt(i)))+1  for i:1:n
%
function [y, grad] = griewank7(x)
% % let i=1:7
% y= (1/400)*((x(1)-10)^2+(x(2)-10)^2+(x(3)-10)^2+(x(4)-10)^2+...
%    (x(5)-10)^2+(x(6)-10)^2+(x(7)-10)^2)-...
%    (cos(x(1)/1^0.5)*cos(x(2)/2^0.5)*cos(x(3)/3^0.5)*...
%    cos(x(4)/4^0.5)*cos(x(5)/5^0.5)*cos(x(6)/6^0.5)*cos(x(7)/7^0.5))+1;
%
% Initialize vector variables
% Gradient = [
% 1/200*x1-1/20+sin(x1)*cos(1/2*x2*2^(1/2))*cos(1/3*x3*3^(1/2))*
% cos(1/2*x4)*cos(1/5*x5*5^(1/2))*cos(1/6*x6*6^(1/2))*
% cos(1/7*x7*7^(1/2)),
% 1/200*x2-1/20+1/2*cos(x1)*sin(1/2*x2*2^(1/2))*2^(1/2)*cos(1/3*x3*
% 3^(1/2))*cos(1/2*x4)*cos(1/5*x5*5^(1/2))*cos(1/6*x6*6^(1/2))*
% cos(1/7*x7*7^(1/2)),
% 1/200*x3-1/20+1/3*cos(x1)*cos(1/2*x2*2^(1/2))*sin(1/3*x3*
% 3^(1/2))*3^(1/2)*cos(1/2*x4)*cos(1/5*x5*5^(1/2))*cos(1/6*x6*6^(1/2))*
% cos(1/7*x7*7^(1/2)),
% 1/200*x4-1/20+1/2*cos(x1)*cos(1/2*x2*2^(1/2))*cos(1/3*x3*
% 3^(1/2))*sin(1/2*x4)*cos(1/5*x5*5^(1/2))*cos(1/6*x6*6^(1/2))*
% cos(1/7*x7*7^(1/2)),
% 1/200*x5-1/20+1/5*cos(x1)*cos(1/2*x2*2^(1/2))*cos(1/3*x3*
% 3^(1/2))*cos(1/2*x4)*sin(1/5*x5*5^(1/2))*5^(1/2)*cos(1/6*
% x6*6^(1/2))*cos(1/7*x7*7^(1/2)),
% 1/200*x6-1/20+1/6*cos(x1)*cos(1/2*x2*2^(1/2))*cos(1/3*x3*
% 3^(1/2))*cos(1/2*x4)*cos(1/5*x5*5^(1/2))*sin(1/6*x6*6^(1/2))*6^(1/2)*
% * cos(1/7*x7*7^(1/2)),
% 1/200*x7-1/20+1/7*cos(x1)*cos(1/2*x2*2^(1/2))*cos(1/3*x3*
% 3^(1/2))*cos(1/2*x4)*cos(1/5*x5*5^(1/2))*cos(1/6*x6*6^(1/2))*
```

```
% sin(1/7*x7*7^(1/2))*7^(1/2)]


grad=[]; xx1=[]; xx2=[]; xx3=[];  xx4=[]; xx5=[]; xx6=[]; xx7=[];
xx1= x(1); xx2=x(2); xx3= x(3); xx4=x(4); xx5= x(5); xx6=x(6); xx7=x(7);

% declare symbolic variables- The symbolic variables are used by Jacobian
% return the gradient functions this is done by setting a variable equal
% to a symbolic expression, and then apply the syms command to the variable
%
syms x1 x2 x3 x4 x5 x6 x7

% fitness function
y =100*(x2-x1^2)^2+(1-x1)^2;

y= (1/400)*((x1-10)^2+(x2-10)^2+(x3-10)^2+(x4-10)^2+...
  (x5-10)^2+(x6-10)^2+(x7-10)^2)-...
  (cos(x1/1^0.5)*cos(x2/2^0.5)*cos(x3/3^0.5)*...
  cos(x4/4^0.5)*cos(x5/5^0.5)*cos(x6/6^0.5)*cos(x7/7^0.5))+1;

% The gradient function is computed using the function jacobian in
% conjucction with symbols representing the independent variables in the
% fitness function. The jacobian takes two parameters; namely variable y
% and  a vector whose components are the variables of the function y.
%
gradf =jacobian(y,[x1,x2,x3,x4,x5,x6,x7]);

% The inline constructs an inline function object corresponding to
% gradient; the vectorize vectorizes the formula for the gradient function.
%
gf1 = inline(vectorize(gradf(1))); gf2 = inline(vectorize(gradf(2)));
gf3 = inline(vectorize(gradf(3))); gf4 = inline(vectorize(gradf(4)));
gf5 = inline(vectorize(gradf(5))); gf6 = inline(vectorize(gradf(6)));
gf7 = inline(vectorize(gradf(7)));


grad(1,1) = gf1(xx1,xx2,xx3,xx4,xx5,xx6,xx7); grad(1,2) = gf2(xx1,xx2,xx3,xx4,xx5,xx6,xx7);
grad(1,3) = gf3(xx1,xx2,xx3,xx4,xx5,xx6,xx7); grad(1,4) = gf4(xx1,xx2,xx3,xx4,xx5,xx6,xx7);
grad(1,5) = gf5(xx1,xx2,xx3,xx4,xx5,xx6,xx7); grad(1,6) = gf6(xx1,xx2,xx3,xx4,xx5,xx6,xx7);
grad(1,7) = gf7(xx1,xx2,xx3,xx4,xx5,xx6,xx7);

y= (1/400)*((x(1)-10)^2+(x(2)-10)^2+(x(3)-10)^2+(x(4)-10)^2+...
  (x(5)-10)^2+(x(6)-10)^2+(x(7)-10)^2)-...
  (cos(x(1)/1^0.5)*cos(x(2)/2^0.5)*cos(x(3)/3^0.5)*...
  cos(x(4)/4^0.5)*cos(x(5)/5^0.5)*cos(x(6)/6^0.5)*cos(x(7)/7^0.5))+1;
end
```

Table 12 – 7- MatLab Function (fminsearch)

```
% [x y history] = minimizeTest(function_handle,x_init)
% The minimize test function receives as input parameter a function handle
% which will point to a user define function, and initial values for vector
% x. This function returns the final values for vector x, y value and a
```

```
% history matrix which includes calculated points along the path of
% searching for minimum value.
%
% The MatLab fminsearch() standard function minimizes a function of several
% variables. The input parameters to this function is a user define
% function; specified at run time, by its function handle, initial value of
% vector x and data structure options- MatLab uses default values (when
% user values are not specified) for the parameter options- The data value
% for the options data structure is defined by using optimset() function,
% as follow:
%
% options = optimset('param1',value1,'param2',value2,...)
% MatLab set any unspecified parameters to [], indicating their
% corresponding default values. Fields that can be set by optimset() and is
% used by fminsearch() are as follow:
%
% Display   Level of display. 'off' displays no output; 'iter' displays
%           output at each iteration; 'final' displays just the final
%           output; 'notify' (default) dislays output only if the function
%           does not converge.
% Followings are exteracted from MatLab user guide to clarify the usage of
% fminsearch() standard function.
% FunValCheck    Check whether objective function values are valid. 'on'
%              displays a warning when the objective function returns a
%              value that is complex, Inf or NaN. 'off' (the default)
%              displays no warning.
% MaxFunEvals   Maximum number of function evaluations allowed
% MaxIter       Maximum number of iterations allowed
% OutputFcn     Specify a user-defined function that the optimization
%              function calls at each iteration.
% TolFun        Termination tolerance on the function value
% TolX          Termination tolerance on x
%

function
   history = [];
   options = optimset('OutputFcn', @myoutput);
   [x y] = fminsearch(function_handle, x_init,options);
   function
      stop = [];
      if state == 'iter'
        history = [history; x];
      end
   end
end
```

4. Conclusions

Increasing dominance of genetic/other randomized algorithms over gradient/other deterministic algorithms We have attempted to provide a glimpse of increasing importance of randomized algorithms (such as the genetic algorithms/evolutionary approaches) over the deterministic

methods (such as the gradient algorithms) with the steadily increasing capability of computing resources such as the processing speed, band width, and storage space. Today (2007) a situation where a processing speed of over a billion FLOPS are widely available on a desk top computer has cropped up where randomized algorithms are more attractive than the deterministic ones for large real-world optimization problems both in terms of accuracy (quality of result) as well as inherent simplicity (ease of human comprehension). With further improvement in the capability, these algorithms will have more dominance over the deterministic ones which in past decades (before 1980s) were practically the only means to tackle small not-so-involved function optimization problems. It can be seen that if the computing power would have remained static (of the order of 10000 FLOPS) during the past four decades, then definitely randomized algorithms could not have gained so much of importance as these have today. So far as the time/computational complexity (cost of computation) is concerned, it is often not a serious issue with most real-world problems since all randomized algorithms as well as all deterministic ones for function optimization are polynomial-time [9, 11]. The function optimization problem is inherently polynomial-time (not NP-hard). However, when the search hyperspace is large (say, 7 or over 7 variables) and the function involving, say some combination of transcendental/special functions, is computationally large, then to track down the global optimal solution could involve significant computation though not often a major complexity issue. In such problems, the error involved in gradient algorithms that usually need some kind of knowledge of the derivative of the function could be significant to affect the global optimum value considerably. A genetic algorithm, or for that matter any randomized algorithm, that usually involves only the function computation will, on the other hand, tend to produce more accurate global minimum. Our numerical experiments with various typical problems in section 3 depict this fact both numerically as well as visually.

Genetic/other randomized algorithms are the only ways for NP-hard problems such as TSPs A traveling salesman problem (TSP) has immense importance in numerous practical applications. Although our subject matter is not connected to an NP-hard problem such as a TSP, we like to stress that no deterministic algorithms are usable because of intractability [9, 11]. The only alternatives are the randomized algorithms, viz., genetic, ant, and other evolutionary approaches. These algorithms are always polynomial-time and hence are always tractable.

References

[1] Rao, S.S. (1978). Optimization: Theory and Applications, Wiley Eastern, New Delhi.
[2] Box, M.J., Daview, D., and Swann, W.H. (1969). Nonlinear Optimization, Oliver and Boyd, Edinburgh.
[3] Fox, R.L. (1971). Optimization Methods for Engineering Design, Addison-Wesley, Reading, Massachusetts.
[4] Leitmann, G. (1962). Optimization Techniques with Applications to Aerospace Systems, Academic Press, New York.
[5] Murray, W. (1972). Numerical Methods for Unconstrained Optimization, Academic Press, London.
[6] Walsh, G.R. (1975). Methods of Optimization, Wiley, New York.

[7] Wilde, D.J. (1964). Optimum Seeking Methods, Prentice-Hall, Englewood Cliffs, New Jersey.
[8] Wolfe, M.A. (1978). Numerical Methods for Unconstrained Optimization: An Introduction, Van Nostrand Reinhold, New York.
[9] Lakshmikantham, V. and Sen, S.K. (2005). Computational Error and Complexity in Science and Engineering, Elsevier, Amsterdam.
[10] Ralston, A. and Reilly, E.D. Jr. eds. (1983). Encyclopedia of Computer Science and Engineering (2^{nd} ed.), Van Nostrand Reinhold, New York.
[11] Krishnamurthy, E.V. and Sen, S.K. (2004). Introductory Theory of Computer Science, Affiliated East-West Press, New Delhi.
[12] Krishnamurthy E.V. and Sen, S.K. (2001). Numerical Algorithms: Computations in Science and Engineering, Affiliated East-West Press, New Delhi.
[13] Davidon, W.C. (1959). Variable metric method for minimization, AEC Research and Development Report ANL-5990 (Rev.), Argonne National Laboratory, Argonne, Illinois.
[14] Fletcher R. (1965). Function minimization without evaluating derivatives: A review, *The Computer J.*, **8**, 33-41.
[15] Fletcher, R. and Powell, M.J.D. (1963). A rapidly convergent descent method for minimization, *The Computer J.*, **6**, 163-68.
[16] Fletcher, R. and Reeves, C.M. (1964). Function minimization by conjugate gradients, *The Computer J.*, **7**, 149-54.
[17] Fox, R.L. (1971). Optimization methods for Engineering Design, Addison-Wesley, Reading, Massachusetts.
[18] Murray, W. (1972). Numerical Methods for Unconstrained Optimization, Academic Press, London.
[19] Powell, M.J.D. (1970). A survey of numerical methods for unconstrained optimization, *SIAM Rev.*, **12**, 79-97.
[20] Nelder, J.A. and Mead, R., (1965). A simplex method for function minimization, *Computer J.*, **7**, 308-13.
[21] Durand, N. and Alliott, J-M. (1999). A combined Nelder-Mead simplex and genetic algorithm, http://recherche.enac.fr/opti/papers/articles/gecco99.pdf
[22] Sen, S.K. and Shaykhian, G.A. (2006).Genetic algorithm for optimization: Preprocessing with n dimensional bisection and error estimation, Proceedings of Neural, Parallel and Scientific Computations 3, 243-52.
[23] Holland, J.H. (1992). Adaptation in Natural and Artificial Systems, MIT Press/Bradford Book Edition, Massachusetts.