Nature-Inspired Metaheuristic Algorithms

Book · July 2010

CITATIONS
3,461

READS
35,537

1 author:

Xin-She Yang
Middlesex University, UK
563 PUBLICATIONS **52,183** CITATIONS

SEE PROFILE

Nature-Inspired Metaheuristic Algorithms Second Edition

Xin-She Yang

University of Cambridge, United Kingdom

Luniver Press

Published in 2010 by Luniver Press
Frome, BA11 6TT, United Kingdom
www.luniver.com

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from
the British Library

ISBN-13: 978-1-905986-28-6
ISBN-10: 1-905986-28-9

CONTENTS

Preface to the Second Edition

Since the publication of the first edition of this book in 2008, significant developments have been made in metaheuristics, and new nature-inspired metaheuristic algorithms emerge, including cuckoo search and bat algorithms. Many readers have taken time to write to me personally, providing valuable feedback, asking for more details of algorithm implementation, or simply expressing interests in applying these new algorithms in their applications.

In this revised edition, we strive to review the latest developments in metaheuristic algorithms, to incorporate readers' suggestions, and to provide a more detailed description to algorithms. Firstly, we have added detailed descriptions of how to incorporate constraints in the actual implementation. Secondly, we have added three chapters on differential evolution, cuckoo search and bat algorithms, while some existing chapters such as ant algorithms and bee algorithms are combined into one due to their similarity. Thirdly, we also explained artificial neural networks and support vector machines in the framework of optimization and metaheuristics. Finally, we have been trying in this book to provide a consistent and unified approach to metaheuristic algorithms, from a brief history in the first chapter to the unified approach in the last chapter.

Furthermore, we have provided more Matlab programs. At the same time, we also omit some of the implementation such as genetic algorithms, as we know that there are many good software packages (both commercial and open course). This allows us to focus more on the implementation of new algorithms. Some of the programs also have a version for constrained optimization, and readers can modify them for their own applications.

Even with the good intention to cover most popular metaheuristic algorithms, the choice of algorithms is a difficult task, as we do not have the space to cover every algorithm. The omission of an algorithm does not mean that it is not popular. In fact, some algorithms are very powerful and routinely used in many applications. Good examples are Tabu search and combinatorial algorithms, and interested readers can refer to the references provided at the end of the book. The effort in writing this little book becomes worth while if this book could in some way encourage readers' interests in metaheuristics.

Xin-She Yang

August 2010

Chapter 1

INTRODUCTION

It is no exaggeration to say that optimization is everywhere, from engineering design to business planning and from the routing of the Internet to holiday planning. In almost all these activities, we are trying to achieve certain objectives or to optimize something such as profit, quality and time. As resources, time and money are always limited in real-world applications, we have to find solutions to optimally use these valuable resources under various constraints. Mathematical optimization or programming is the study of such planning and design problems using mathematical tools. Nowadays, computer simulations become an indispensable tool for solving such optimization problems with various efficient search algorithms.

1.1 OPTIMIZATION

Mathematically speaking, it is possible to write most optimization problems in the generic form

$$\underset{\boldsymbol{x}\in\Re^n}{\text{minimize}} \quad f_i(\boldsymbol{x}), \qquad (i=1,2,...,M), \tag{1.1}$$

$$\text{subject to} \quad h_j(\boldsymbol{x})=0, \quad (j=1,2,...,J), \tag{1.2}$$

$$g_k(\boldsymbol{x}) \le 0, \quad (k=1,2,...,K), \tag{1.3}$$

where $f_i(\boldsymbol{x}), h_j(\boldsymbol{x})$ and $g_k(\boldsymbol{x})$ are functions of the design vector

$$\boldsymbol{x}=(x_1,x_2,...,x_n)^T. \tag{1.4}$$

Here the components x_i of $\boldsymbol{x}$ are called design or decision variables, and they can be real continuous, discrete or the mixed of these two.

The functions $f_i(\boldsymbol{x})$ where $i=1,2,...,M$ are called the objective functions or simply cost functions, and in the case of $M=1$, there is only a single objective. The space spanned by the decision variables is called the design space or search space $\Re^n$, while the space formed by the objective function values is called the solution space or response space. The equalities for h_j and inequalities for g_k are called constraints. It is worth pointing

out that we can also write the inequalities in the other way ≥ 0, and we can also formulate the objectives as a maximization problem.

In a rare but extreme case where there is no objective at all, there are only constraints. Such a problem is called a feasibility problem because any feasible solution is an optimal solution.

If we try to classify optimization problems according to the number of objectives, then there are two categories: single objective $M = 1$ and multiobjective $M > 1$. Multiobjective optimization is also referred to as multicriteria or even multi-attributes optimization in the literature. In real-world problems, most optimization tasks are multiobjective. Though the algorithms we will discuss in this book are equally applicable to multiobjective optimization with some modifications, we will mainly place the emphasis on single objective optimization problems.

Similarly, we can also classify optimization in terms of number of constraints $J + K$. If there is no constraint at all $J = K = 0$, then it is called an unconstrained optimization problem. If $K = 0$ and $J \geq 1$, it is called an equality-constrained problem, while $J = 0$ and $K \geq 1$ becomes an inequality-constrained problem. It is worth pointing out that in some formulations in the optimization literature, equalities are not explicitly included, and only inequalities are included. This is because an equality can be written as two inequalities. For example $h(\boldsymbol{x}) = 0$ is equivalent to $h(\boldsymbol{x}) \leq 0$ and $h(\boldsymbol{x}) \geq 0$.

We can also use the actual function forms for classification. The objective functions can be either linear or nonlinear. If the constraints h_j and g_k are all linear, then it becomes a linearly constrained problem. If both the constraints and the objective functions are all linear, it becomes a linear programming problem. Here 'programming' has nothing to do with computing programming, it means planning and/or optimization. However, generally speaking, all f_i, h_j and g_k are nonlinear, we have to deal with a nonlinear optimization problem.

1.2 SEARCH FOR OPTIMALITY

After an optimization problem is formulated correctly, the main task is to find the optimal solutions by some solution procedure using the right mathematical techniques.

Figuratively speaking, searching for the optimal solution is like treasure hunting. Imagine we are trying to hunt for a hidden treasure in a hilly landscape within a time limit. In one extreme, suppose we are blindfold without any guidance, the search process is essentially a pure random search, which is usually not efficient as we can expect. In another extreme, if we are told the treasure is placed at the highest peak of a known region, we will then directly climb up to the steepest cliff and try to reach to the highest peak, and this scenario corresponds to the classical hill-climbing

techniques. In most cases, our search is between these extremes. We are not blind-fold, and we do not know where to look for. It is a silly idea to search every single square inch of an extremely large hilly region so as to find the treasure.

The most likely scenario is that we will do a random walk, while looking for some hints; we look at some place almost randomly, then move to another plausible place, then another and so on. Such random walk is a main characteristic of modern search algorithms. Obviously, we can either do the treasure-hunting alone, so the whole path is a trajectory-based search, and simulated annealing is such a kind. Alternatively, we can ask a group of people to do the hunting and share the information (and any treasure found), and this scenario uses the so-called swarm intelligence and corresponds to the particle swarm optimization, as we will discuss later in detail. If the treasure is really important and if the area is extremely large, the search process will take a very long time. If there is no time limit and if any region is accessible (for example, no islands in a lake), it is theoretically possible to find the ultimate treasure (the global optimal solution).

Obviously, we can refine our search strategy a little bit further. Some hunters are better than others. We can only keep the better hunters and recruit new ones, this is something similar to the genetic algorithms or evolutionary algorithms where the search agents are improving. In fact, as we will see in almost all modern metaheuristic algorithms, we try to use the best solutions or agents, and randomize (or replace) the not-so-good ones, while evaluating each individual's competence (fitness) in combination with the system history (use of memory). With such a balance, we intend to design better and efficient optimization algorithms.

Classification of optimization algorithm can be carried out in many ways. A simple way is to look at the nature of the algorithm, and this divides the algorithms into two categories: deterministic algorithms, and stochastic algorithms. Deterministic algorithms follow a rigorous procedure, and its path and values of both design variables and the functions are repeatable. For example, hill-climbing is a deterministic algorithm, and for the same starting point, they will follow the same path whether you run the program today or tomorrow. On the other hand, stochastic algorithms always have some randomness. Genetic algorithms are a good example, the strings or solutions in the population will be different each time you run a program since the algorithms use some pseudo-random numbers, though the final results may be no big difference, but the paths of each individual are not exactly repeatable.

Furthermore, there is a third type of algorithm which is a mixture, or a hybrid, of deterministic and stochastic algorithms. For example, hill-climbing with a random restart is a good example. The basic idea is to use the deterministic algorithm, but start with different initial points. This has certain advantages over a simple hill-climbing technique, which may be

stuck in a local peak. However, since there is a random component in this hybrid algorithm, we often classify it as a type of stochastic algorithm in the optimization literature.

1.3 NATURE-INSPIRED METAHEURISTICS

Most conventional or classic algorithms are deterministic. For example, the simplex method in linear programming is deterministic. Some deterministic optimization algorithms used the gradient information, they are called gradient-based algorithms. For example, the well-known Newton-Raphson algorithm is gradient-based, as it uses the function values and their derivatives, and it works extremely well for smooth unimodal problems. However, if there is some discontinuity in the objective function, it does not work well. In this case, a non-gradient algorithm is preferred. Non-gradient-based or gradient-free algorithms do not use any derivative, but only the function values. Hooke-Jeeves pattern search and Nelder-Mead downhill simplex are examples of gradient-free algorithms.

For stochastic algorithms, in general we have two types: heuristic and metaheuristic, though their difference is small. Loosely speaking, *heuristic* means 'to find' or 'to discover by trial and error'. Quality solutions to a tough optimization problem can be found in a reasonable amount of time, but there is no guarantee that optimal solutions are reached. It hopes that these algorithms work most of the time, but not all the time. This is good when we do not necessarily want the best solutions but rather good solutions which are easily reachable.

Further development over the heuristic algorithms is the so-called metaheuristic algorithms. Here *meta-* means 'beyond' or 'higher level', and they generally perform better than simple heuristics. In addition, all metaheuristic algorithms use certain tradeoff of randomization and local search. It is worth pointing out that no agreed definitions of heuristics and metaheuristics exist in the literature; some use 'heuristics' and 'metaheuristics' interchangeably. However, the recent trend tends to name all stochastic algorithms with randomization and local search as metaheuristic. Here we will also use this convention. Randomization provides a good way to move away from local search to the search on the global scale. Therefore, almost all metaheuristic algorithms intend to be suitable for global optimization.

Heuristics is a way by trial and error to produce acceptable solutions to a complex problem in a reasonably practical time. The complexity of the problem of interest makes it impossible to search every possible solution or combination, the aim is to find good feasible solution in an acceptable timescale. There is no guarantee that the best solutions can be found, and we even do not know whether an algorithm will work and why if it does work. The idea is to have an efficient but practical algorithm that will work most the time and is able to produce good quality solutions. Among

the found quality solutions, it is expected some of them are nearly optimal, though there is no guarantee for such optimality.

Two major components of any metaheuristic algorithms are: intensification and diversification, or exploitation and exploration. Diversification means to generate diverse solutions so as to explore the search space on the global scale, while intensification means to focus on the search in a local region by exploiting the information that a current good solution is found in this region. This is in combination with with the selection of the best solutions. The selection of the best ensures that the solutions will converge to the optimality, while the diversification via randomization avoids the solutions being trapped at local optima and, at the same time, increases the diversity of the solutions. The good combination of these two major components will usually ensure that the global optimality is achievable.

Metaheuristic algorithms can be classified in many ways. One way is to classify them as: population-based and trajectory-based. For example, genetic algorithms are population-based as they use a set of strings, so is the particle swarm optimization (PSO) which uses multiple agents or particles.

On the other hand, simulated annealing uses a single agent or solution which moves through the design space or search space in a piecewise style. A better move or solution is always accepted, while a not-so-good move can be accepted with a certain probability. The steps or moves trace a trajectory in the search space, with a non-zero probability that this trajectory can reach the global optimum.

Before we introduce all popular meteheuristic algorithms in detail, let us look at their history briefly.

1.4 A BRIEF HISTORY OF METAHEURISTICS

Throughout history, especially at the early periods of human history, we humans' approach to problem-solving has always been heuristic or metaheuristic – by trial and error. Many important discoveries were done by 'thinking outside the box', and often by accident; that is heuristics. Archimedes's Eureka moment was a heuristic triumph. In fact, our daily learning experience (at least as a child) is dominantly heuristic.

Despite its ubiquitous nature, metaheuristics as a scientific method to problem solving is indeed a modern phenomenon, though it is difficult to pinpoint when the metaheuristic method was first used. Alan Turing was probably the first to use heuristic algorithms during the second World War when he was breaking German Enigma ciphers at Bletchley Park. Turing called his search method *heuristic search*, as it could be expected it worked most of time, but there was no guarantee to find the correct solution, but it was a tremendous success. In 1945, Turing was recruited to the National Physical Laboratory (NPL), UK where he set out his design for

the Automatic Computing Engine (ACE). In an NPL report on *Intelligent machinery* in 1948, he outlined his innovative ideas of machine intelligence and learning, neural networks and evolutionary algorithms.

The 1960s and 1970s were the two important decades for the development of evolutionary algorithms. First, John Holland and his collaborators at the University of Michigan developed the genetic algorithms in 1960s and 1970s. As early as 1962, Holland studied the adaptive system and was the first to use crossover and recombination manipulations for modeling such system. His seminal book summarizing the development of genetic algorithms was published in 1975. In the same year, De Jong finished his important dissertation showing the potential and power of genetic algorithms for a wide range of objective functions, either noisy, multimodal or even discontinuous.

In essence, a genetic algorithm (GA) is a search method based on the abstraction of Darwinian evolution and natural selection of biological systems and representing them in the mathematical operators: crossover or recombination, mutation, fitness, and selection of the fittest. Ever since, genetic algorithms become so successful in solving a wide range of optimization problems, there have several thousands of research articles and hundreds of books written. Some statistics show that a vast majority of Fortune 500 companies are now using them routinely to solve tough combinatorial optimization problems such as planning, data-fitting, and scheduling.

During the same period, Ingo Rechenberg and Hans-Paul Schwefel both then at the Technical University of Berlin developed a search technique for solving optimization problem in aerospace engineering, called evolutionary strategy, in 1963. Later, Peter Bienert joined them and began to construct an automatic experimenter using simple rules of mutation and selection. There was no crossover in this technique, only mutation was used to produce an offspring and an improved solution was kept at each generation. This was essentially a simple trajectory-style hill-climbing algorithm with randomization. As early as 1960, Lawrence J. Fogel intended to use simulated evolution as a learning process as a tool to study artificial intelligence. Then, in 1966, L. J. Fogel, together A. J. Owen and M. J. Walsh, developed the evolutionary programming technique by representing solutions as finite-state machines and randomly mutating one of these machines. The above innovative ideas and methods have evolved into a much wider discipline, called *evolutionary algorithms* and/or *evolutionary computation*.

Although our focus in this book is metaheuristic algorithms, other algorithms can be thought as a heuristic optimization technique. These includes artificial neural networks, support vector machines and many other machine learning techniques. Indeed, they all intend to minimize their learning errors and prediction (capability) errors via iterative trials and errors.

Artificial neural networks are now routinely used in many applications. In 1943, W. McCulloch and W. Pitts proposed the artificial neurons as simple information processing units. The concept of a neural network was probably first proposed by Alan Turing in his 1948 NPL report concerning 'intelligent machinery'. Significant developments were carried out from the 1940s and 1950s to the 1990s with more than 60 years of history.

The support vector machine as a classification technique can date back to the earlier work by V. Vapnik in 1963 on linear classifiers, and the nonlinear classification with kernel techniques were developed by V. Vapnik and his collaborators in the 1990s. A systematical summary in Vapnik's book on the Nature of Statistical Learning Theory was published in 1995.

The two decades of 1980s and 1990s were the most exciting time for metaheuristic algorithms. The next big step is the development of simulated annealing (SA) in 1983, an optimization technique, pioneered by S. Kirkpatrick, C. D. Gellat and M. P. Vecchi, inspired by the annealing process of metals. It is a trajectory-based search algorithm starting with an initial guess solution at a high temperature, and gradually cooling down the system. A move or new solution is accepted if it is better; otherwise, it is accepted with a probability, which makes it possible for the system to escape any local optima. It is then expected that if the system is cooled down slowly enough, the global optimal solution can be reached.

The actual first usage of memory in modern metaheuristics is probably due to Fred Glover's Tabu search in 1986, though his seminal book on Tabu search was published later in 1997.

In 1992, Marco Dorigo finished his PhD thesis on optimization and natural algorithms, in which he described his innovative work on ant colony optimization (ACO). This search technique was inspired by the swarm intelligence of social ants using pheromone as a chemical messenger. Then, in 1992, John R. Koza of Stanford University published a treatise on genetic programming which laid the foundation of a whole new area of machine learning, revolutionizing computer programming. As early as in 1988, Koza applied his first patent on genetic programming. The basic idea is to use the genetic principle to breed computer programs so as to gradually produce the best programs for a given type of problem.

Slightly later in 1995, another significant progress is the development of the particle swarm optimization (PSO) by American social psychologist James Kennedy, and engineer Russell C. Eberhart. Loosely speaking, PSO is an optimization algorithm inspired by swarm intelligence of fish and birds and by even human behavior. The multiple agents, called particles, swarm around the search space starting from some initial random guess. The swarm communicates the current best and shares the global best so as to focus on the quality solutions. Since its development, there have been about 20 different variants of particle swarm optimization techniques, and have been applied to almost all areas of tough optimization problems. There is

some strong evidence that PSO is better than traditional search algorithms and even better than genetic algorithms for many types of problems, though this is far from conclusive.

In around 1996 and later in 1997, R. Storn and K. Price developed their vector-based evolutionary algorithm, called differential evolution (DE), and this algorithm proves more efficient than genetic algorithms in many applications.

In 1997, the publication of the 'no free lunch theorems for optimization' by D. H. Wolpert and W. G. Macready sent out a shock way to the optimization community. Researchers have been always trying to find better algorithms, or even universally robust algorithms, for optimization, especially for tough NP-hard optimization problems. However, these theorems state that if algorithm A performs better than algorithm B for some optimization functions, then B will outperform A for other functions. That is to say, if averaged over all possible function space, both algorithms A and B will perform on average equally well. Alternatively, there is no universally better algorithms exist. That is disappointing, right? Then, people realized that we do not need the average over all possible functions for a given optimization problem. What we want is to find the best solutions, which has nothing to do with average over all possible function space. In addition, we can accept the fact that there is no universal or magical tool, but we do know from our experience that some algorithms indeed outperform others for given types of optimization problems. So the research now focuses on finding the best and most efficient algorithm(s) for a given problem. The objective is to design better algorithms for most types of problems, not for all the problems. Therefore, the search is still on.

At the turn of the 21st century, things became even more exciting. First, Zong Woo Geem *et al.* in 2001 developed the harmony search (HS) algorithm, which has been widely applied in solving various optimization problems such as water distribution, transport modelling and scheduling. In 2004, S. Nakrani and C. Tovey proposed the honey bee algorithm and its application for optimizing Internet hosting centers, which followed by the development of a novel bee algorithm by D. T. Pham *et al.* in 2005 and the artificial bee colony (ABC) by D. Karaboga in 2005. In 2008, the author of this book developed the firefly algorithm (FA)[1]. Quite a few research articles on the firefly algorithm then followed, and this algorithm has attracted a wide range of interests. In 2009, Xin-She Yang at Cambridge University, UK, and Suash Deb at Raman College of Engineering, India, introduced an efficient cuckoo search (CS) algorithm, and it has been demonstrated that CS is far more effective than most existing metaheuristic algorithms

[1] X. S. Yang, *Nature-Inspired Meteheuristic Algorithms*, Luniver Press, (2008)

including particle swarm optimization[2]. In 2010, the author of this book developed a bat-inspired algorithm for continuous optimization, and its efficiency is quite promising.

As we can see, more and more metaheuristic algorithms are being developed. Such a diverse range of algorithms necessitates a systematic summary of various metaheuristic algorithms, and this book is such an attempt to introduce all the latest nature-inspired metaheuristics with diverse applications.

We will discuss all major modern metaheuristic algorithms in the rest of this book, including simulated annealing (SA), genetic algorithms (GA), ant colony optimization (ACO), bee algorithms (BA), differential evolution (DE), particle swarm optimization (PSO), harmony search (HS), the firefly algorithm (FA), cuckoo search (CS) and bat-inspired algorithm (BA), and others.

REFERENCES

1. C. M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford, 1995.
2. B. J. Copeland, *The Essential Turing*, Oxford University Press, 2004.
3. B. J. Copeland, *Alan Turing's Automatic Computing Engine*, Oxford University Press, 2005.
4. K. De Jong, *Analysis of the Behaviour of a Class of Genetic Adaptive Systems*, PhD thesis, University of Michigan, Ann Anbor, 1975.
5. M. Dorigo, *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, Italy, 1992.
6. L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence Through Simulated Evolution*, Wiley, 1966.
7. Z. W. Geem, J. H. Kim and G. V. Loganathan, A new heuristic optimization: Harmony search, *Simulation*, **76**(2), 60-68 (2001).
8. F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, 1997.
9. J. Holland, *Adaptation in Natural and Artificial systems*, University of Michigan Press, Ann Anbor, 1975.
10. P. Judea, *Heuristics*, Addison-Wesley, 1984.
11. D. Karaboga, An idea based on honey bee swarm for numerical optimization, Technical Report, Erciyes University, 2005.
12. J. Kennedy and R. Eberhart, Particle swarm optimization, in: *Proc. of the IEEE Int. Conf. on Neural Networks*, Piscataway, NJ, pp. 1942-1948 (1995).

[2]Novel cuckoo search 'beats' particle swarm optimization, *Science Daily*, news article (28 May 2010), www.sciencedaily.com

13. S. Kirkpatrick, C. D. Gellat, and M. P. Vecchi, Optimization by simulated annealing, *Science*, **220**, 671-680 (1983).

14. J. R. Koza, *Genetic Programming: One the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

15. S. Nakrani and C. Tovey, On honey bees and dynamic server allocation in Internet hostubg centers, *Adaptive Behavior*, **12**, 223-240 (2004).

16. D. T. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim and M. Zaidi, The bees algorithm, Technical Note, Manufacturing Engineering Center, Cardiff University, 2005.

17. A. Schrijver, On the history of combinatorial optimization (till 1960), in: *Handbook of Discrete Optimization* (Eds K. Aardal, G. L. Nemhauser, R. Weismantel), Elsevier, Amsterdam, pp.1-68 (2005).

18. H. T. Siegelmann and E. D. Sontag, Turing computability with neural nets, *Appl. Math. Lett.*, **4**, 77-80 (1991).

19. R. Storn and K. Price, Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces, *Journal of Global Optimization*, **11**, 341-359 (1997).

20. A. M. Turing, *Intelligent Machinery*, National Physical Laboratory, technical report, 1948.

21. V. Vapnik, *The Nature of Statistical Learning Theory*, Springer-Verlag, New York, 1995.

22. V. Vapnik, S. Golowich, A. Smola, Support vector method for function approxiation, regression estimation, and signal processing, in: *Advances in Neural Information Processing System 9* (Eds. M. Mozer, M. Jordan and T. Petsche), MIT Press, Cambridge MA, 1997.

23. D. H. Wolpert and W. G. Macready, No free lunch theorems for optimization, *IEEE Transaction on Evolutionary Computation*, **1**, 67-82 (1997).

24. X. S. Yang, *Nature-Inspired Metaheuristic Algorithms*, Luniver Press, (2008).

25. X. S. Yang, Firefly algorithms for multimodal optimization, *Proc. 5th Symposium on Stochastic Algorithms, Foundations and Applications, SAGA 2009*, Eds. O. Watanabe and T. Zeugmann, Lecture Notes in Computer Science, **5792**, 169-178 (2009).

26. X. S. Yang and S. Deb, Cuckoo search via Lévy flights, in: *Proc. of World Congress on Nature & Biologically Inspired Computing* (NaBic 2009), IEEE Publications, USA, pp. 210-214 (2009).

27. X. S. Yang and S. Deb, Engineering optimization by cuckoo search, *Int. J. Math. Modelling & Num. Optimization*, **1**, 330-343 (2010).

28. X. S. Yang, A new metaheuristic bat-inspired algorithm, in: *Nature Inspired Cooperative Strategies for Optimization* (NICSO 2010) (Eds. J. R. Gonzalez et al.), Springer, SCI **284**, 65-74 (2010).

29. History of optimization, http://hse-econ.fi/kitti/opthist.html

30. Turing Archive for the History of Computing, www.alanturing.net/

Chapter 2

RANDOM WALKS AND LÉVY FLIGHTS

From the brief analysis of the main characteristics of metaheuristic algorithms in the first chapter, we know that randomization plays an important role in both exploration and exploitation, or diversification and intensification. The essence of such randomization is the random walk. In this chapter, we will briefly review the fundamentals of random walks, Lévy flights and Markov chains. These concepts may provide some hints and insights into how and why metaheuristic algorithms behave.

2.1 RANDOM VARIABLES

Loosely speaking, a random variable can be considered as an expression whose value is the realization or outcome of events associated with a random process such as the noise level on the street. The values of random variables are real, though for some variables such as the number of cars on a road can only take discrete values, and such random variables are called discrete random variables. If a random variable such as noise at a particular location can take any real values in an interval, it is called continuous. If a random variable can have both continuous and discrete values, it is called a mixed type. Mathematically speaking, a random variable is a function which maps events to real numbers. The domain of this mapping is called the sample space.

For each random variable, a probability density function can be used to express its probability distribution. For example, the number of phone calls per minute, and the number of users of a web server per day all obey the Poisson distribution

$$p(n;\lambda) = \frac{\lambda^n e^{-\lambda}}{n!}, \qquad (n = 0, 1, 2, ...), \tag{2.1}$$

where $\lambda > 0$ is a parameter which is the mean or expectation of the occurrence of the event during a unit interval.

Different random variables will have different distributions. Gaussian distribution or normal distribution is by far the most popular distributions, because many physical variables including light intensity, and er-

rors/uncertainty in measurements, and many other processes obey the normal distribution

$$p(x;\mu,\sigma^2)=\frac{1}{\sigma\sqrt{2\pi}}\exp[-\frac{(x-\mu)^2}{2\sigma^2}],\qquad -\infty<x<\infty, \tag{2.2}$$

where μ is the mean and $\sigma>0$ is the standard deviation. This normal distribution is often denoted by $\mathrm{N}(\mu,\sigma^2)$. In the special case when $\mu=0$ and $\sigma=1$, it is called a standard normal distribution, denoted by $\mathrm{N}(0,1)$.

In the context of metaheuristics, another important distribution is the so-called Lévy distribution, which is a distribution of the sum of N identically and independently distribution random variables whose Fourier transform takes the following form

$$F_N(k)=\exp[-N|k|^{\beta}]. \tag{2.3}$$

The inverse to get the actual distribution $L(s)$ is not straightforward, as the integral

$$L(s)=\frac{1}{\pi}\int_0^{\infty}\cos(\tau s)e^{-\alpha\,\tau^{\beta}}d\tau,\qquad (0<\beta\le 2), \tag{2.4}$$

does not have analytical forms, except for a few special cases. Here $L(s)$ is called the Lévy distribution with an index β. For most applications, we can set $\alpha=1$ for simplicity. Two special cases are $\beta=1$ and $\beta=2$. When $\beta=1$, the above integral becomes the Cauchy distribution. When $\beta=2$, it becomes the normal distribution. In this case, Lévy flights become the standard Brownian motion.

Mathematically speaking, we can express the integral (2.4) as an asymptotic series, and its leading-order approximation for the flight length results in a power-law distribution

$$L(s)\sim|s|^{-1-\beta}, \tag{2.5}$$

which is heavy-tailed. The variance of such a power-law distribution is infinite for $0<\beta<2$. The moments diverge (or are infinite) for $0<\beta<2$, which is a stumbling block for mathematical analysis.

2.2 RANDOM WALKS

A random walk is a random process which consists of taking a series of consecutive random steps. Mathematically speaking, let S_N denotes the sum of each consecutive random step X_i, then S_N forms a random walk

$$S_N=\sum_{i=1}^{N}X_i=X_1+...+X_N, \tag{2.6}$$

where X_i is a random step drawn from a random distribution. This relationship can also be written as a recursive formula

$$S_N = \sum_{i=1}^{N-1} + X_N = S_{N-1} + X_N, \tag{2.7}$$

which means the next state S_N will only depend the current existing state S_{N-1} and the motion or transition X_N from the existing state to the next state. This is typically the main property of a Markov chain to be introduced later.

Here the step size or length in a random walk can be fixed or varying. Random walks have many applications in physics, economics, statistics, computer sciences, environmental science and engineering.

Consider a scenario, a drunkard walks on a street, at each step, he can randomly go forward or backward, this forms a random walk in one-dimensional. If this drunkard walks on a football pitch, he can walk in any direction randomly, this becomes a 2D random walk. Mathematically speaking, a random walk is given by the following equation

$$S_{t+1} = S_t + w_t, \tag{2.8}$$

where S_t is the current location or state at t, and w_t is a step or random variable with a known distribution.

If each step or jump is carried out in the n-dimensional space, the random walk discussed earlier

$$S_N = \sum_{i=1}^{N} X_i, \tag{2.9}$$

becomes a random walk in higher dimensions. In addition, there is no reason why each step length should be fixed. In fact, the step size can also vary according to a known distribution. If the step length obeys the Gaussian distribution, the random walk becomes the Brownian motion (see Fig. 2.1).

In theory, as the number of steps N increases, the central limit theorem implies that the random walk (2.9) should approaches a Gaussian distribution. As the mean of particle locations shown in Fig. 2.1 is obviously zero, their variance will increase linearly with t. In general, in the d-dimensional space, the variance of Brownian random walks can be written as

$$\sigma^2(t) = |v_0|^2 t^2 + (2dD)t, \tag{2.10}$$

where v_0 is the drift velocity of the system. Here $D = s^2/(2\tau)$ is the effective diffusion coefficient which is related to the step length s over a short time interval τ during each jump.

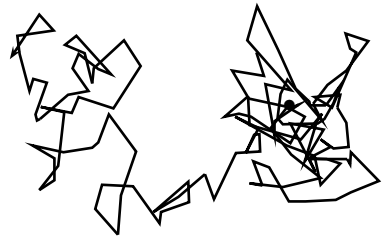

Figure 2.1: Brownian motion in 2D: random walk with a Gaussian step-size distribution and the path of 50 steps starting at the origin $(0, 0)$ (marked with $\bullet$).

Therefore, the Brownian motion $B(t)$ essentially obeys a Gaussian distribution with zero mean and time-dependent variance. That is, $B(t) \sim N(0, \sigma^2(t))$ where $\sim$ means the random variable obeys the distribution on the right-hand side; that is, samples should be drawn from the distribution. A diffusion process can be viewed as a series of Brownian motion, and the motion obeys the Gaussian distribution. For this reason, standard diffusion is often referred to as the Gaussian diffusion. If the motion at each step is not Gaussian, then the diffusion is called non-Gaussian diffusion.

If the step length obeys other distribution, we have to deal with more generalized random walk. A very special case is when the step length obeys the Lévy distribution, such a random walk is called a Lévy flight or Lévy walk.

2.3 LÉVY DISTRIBUTION AND LÉVY FLIGHTS

Broadly speaking, Lévy flights are a random walk whose step length is drawn from the Lévy distribution, often in terms of a simple power-law formula $L(s) \sim |s|^{-1-\beta}$ where $0 < \beta \leq 2$ is an index. Mathematically speaking, a simple version of Lévy distribution can be defined as

$$L(s, \gamma, \mu) = \begin{cases} \sqrt{\frac{\gamma}{2\pi}} \exp[-\frac{\gamma}{2(s-\mu)}] \frac{1}{(s-\mu)^{3/2}}, & 0 < \mu < s < \infty \\ 0 & \text{otherwise}, \end{cases} \tag{2.11}$$

where $\mu > 0$ is a minimum step and γ is a scale parameter. Clearly, as $s \to \infty$, we have

$$L(s, \gamma, \mu) \approx \sqrt{\frac{\gamma}{2\pi}} \frac{1}{s^{3/2}}. \tag{2.12}$$

This is a special case of the generalized Lévy distribution.

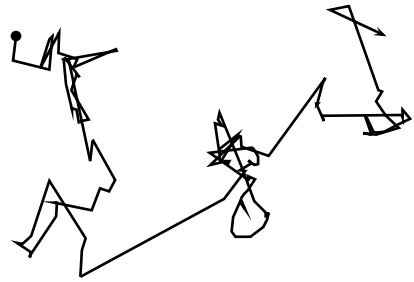

Figure 2.2: Lévy flights in consecutive 50 steps starting at the origin $(0, 0)$ (marked with $\bullet$).

In general, Lévy distribution should be defined in terms of Fourier transform

$$F(k) = \exp[-\alpha|k|^\beta], \qquad 0 < \beta \le 2, \tag{2.13}$$

where α is a scale parameter. The inverse of this integral is not easy, as it does not have analytical form, except for a few special cases.

For the case of $\beta = 2$, we have

$$F(k) = \exp[-\alpha k^2], \tag{2.14}$$

whose inverse Fourier transform corresponds to a Gaussian distribution. Another special case is $\beta = 1$, and we have

$$F(k) = \exp[-\alpha|k|], \tag{2.15}$$

which corresponds to a Cauchy distribution

$$p(x, \gamma, \mu) = \frac{1}{\pi} \frac{\gamma}{\gamma^2 + (x - \mu)^2}, \tag{2.16}$$

where μ is the location parameter, while γ controls the scale of this distribution.

For the general case, the inverse integral

$$L(s) = \frac{1}{\pi} \int_0^\infty \cos(ks) \exp[-\alpha|k|^\beta] dk, \tag{2.17}$$

can be estimated only when s is large. We have

$$L(s) \to \frac{\alpha\, \beta\, \Gamma(\beta) \sin(\pi\beta/2)}{\pi |s|^{1+\beta}}, \qquad s \to \infty. \tag{2.18}$$

Here $\Gamma(z)$ is the Gamma function

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt. \tag{2.19}$$

In the case when $z = n$ is an integer, we have $\Gamma(n) = (n-1)!$.

Lévy flights are more efficient than Brownian random walks in exploring unknown, large-scale search space. There are many reasons to explain this efficiency, and one of them is due to the fact that the variance of Lévy flights

$$\sigma^2(t) \sim t^{3-\beta}, \qquad 1 \le \beta \le 2, \tag{2.20}$$

increases much faster than the linear relationship (i.e., $\sigma^2(t) \sim t$) of Brownian random walks.

Fig. 2.2 shows the path of Lévy flights of 50 steps starting from $(0, 0)$ with $\beta = 1$. It is worth pointing out that a power-law distribution is often linked to some scale-free characteristics, and Lévy flights can thus show self-similarity and fractal behavior in the flight patterns.

From the implementation point of view, the generation of random numbers with Lévy flights consists of two steps: the choice of a random direction and the generation of steps which obey the chosen Lévy distribution. The generation of a direction should be drawn from a uniform distribution, while the generation of steps is quite tricky. There are a few ways of achieving this, but one of the most efficient and yet straightforward ways is to use the so-called Mantegna algorithm for a symmetric Lévy stable distribution. Here 'symmetric' means that the steps can be positive and negative.

A random variable U and its probability distribution can be called stable if a linear combination of its two identical copies (or U_1 and U_2) obeys the same distribution. That is, $aU_1 + bU_2$ has the same distribution as $cU + d$ where $a, b > 0$ and $c, d \in \Re$. If $d = 0$, it is called strictly stable. Gaussian, Cauchy and Lévy distributions are all stable distributions.

In Mantegna's algorithm, the step length s can be calculated by

$$s = \frac{u}{|v|^{1/\beta}}, \tag{2.21}$$

where u and v are drawn from normal distributions. That is

$$u \sim N(0, \sigma_u^2), \qquad v \sim N(0, \sigma_v^2), \tag{2.22}$$

where

$$\sigma_u = \left\{ \frac{\Gamma(1+\beta)\sin(\pi\beta/2)}{\Gamma[(1+\beta)/2]\ \beta\ 2^{(\beta-1)/2}} \right\}^{1/\beta}, \qquad \sigma_v = 1. \tag{2.23}$$

This distribution (for s) obeys the expected Lévy distribution for $|s| \ge |s_0|$ where s_0 is the smallest step. In principle, $|s_0| \gg 0$, but in reality s_0 can be taken as a sensible value such as $s_0 = 0.1$ to 1.

Studies show that Lévy flights can maximize the efficiency of resource searches in uncertain environments. In fact, Lévy flights have been observed among foraging patterns of albatrosses and fruit flies, and spider monkeys. Even humans such as the Ju/'hoansi hunter-gatherers can trace paths of Lévy-flight patterns. In addition, Lévy flights have many applications. Many physical phenomena such as the diffusion of fluorescent molecules, cooling behavior and noise could show Lévy-flight characteristics under the right conditions.

2.4 OPTIMIZATION AS MARKOV CHAINS

In every aspect, a simple random walk we discussed earlier can be considered as a Markov chain. Briefly speaking, a random variable ζ is a Markov process if the transition probability, from state $\zeta_t = S_i$ at time t to another state $\zeta_{t+1} = S_j$, depends only on the current state ζ_i. That is

$$P(i,j) \equiv P(\zeta_{t+1} = S_j | \zeta_0 = S_p, ..., \zeta_t = S_i)$$
$$= P(\zeta_{t+1} = S_j | \zeta_t = S_i), \tag{2.24}$$

which is independent of the states before t. In addition, the sequence of random variables $(\zeta_0, \zeta_1, ..., \zeta_n)$ generated by a Markov process is subsequently called a Markov chain. The transition probability $P(i,j) \equiv P(i \to j) = P_{ij}$ is also referred to as the transition kernel of the Markov chain.

If we rewrite the random walk relationship (2.7) with a random move governed by w_t which depends on the transition probability P, we have

$$S_{t+1} = S_t + w_t, \tag{2.25}$$

which indeed has the properties of a Markov chain. Therefore, a random walk is a Markov chain.

In order to solve an optimization problem, we can search the solution by performing a random walk starting from a good initial but random guess solution. However, simple or blind random walks are not efficient. To be computationally efficient and effective in searching for new solutions, we have to keep the best solutions found so far, and to increase the mobility of the random walk so as to explore the search space more effectively. Most importantly, we have to find a way to control the walk in such a way that it can move towards the optimal solutions more quickly, rather than wander away from the potential best solutions. These are the challenges for most metaheuristic algorithms.

Further research along the route of Markov chains is that the development of the Markov chain Monte Carlo (MCMC) method, which is a class of sample-generating methods. It attempts to directly draw samples from some highly complex multi-dimensional distribution using a Markov

chain with known transition probability. Since the 1990s, the Markov chain Monte Carlo has become a powerful tool for Bayesian statistical analysis, Monte Carlo simulations, and potentially optimization with high nonlinearity.

An important link between MCMC and optimization is that some heuristic and metaheuristic search algorithms such as simulated annealing to be introduced later use a trajectory-based approach. They start with some initial (random) state, and propose a new state (solution) randomly. Then, the move is accepted or not, depending on some probability. There is strongly similar to a Markov chain. In fact, the standard simulated annealing is a random walk.

Mathematically speaking, a great leap in understanding metaheuristic algorithms is to view a Markov chain Monte carlo as an optimization procedure. If we want to find the minimum of an objective function $f(\theta)$ at $\theta = \theta_*$ so that $f_* = f(\theta_*) \leq f(\theta)$, we can convert it to a target distribution for a Markov chain

$$\pi(\theta) = e^{-\beta f(\theta)}, \tag{2.26}$$

where $\beta > 0$ is a parameter which acts as a normalized factor. β value should be chosen so that the probability is close to 1 when $\theta \to \theta_*$. At $\theta = \theta_*$, $\pi(\theta)$ should reach a maximum $\pi_* = \pi(\theta_*) \geq \pi(\theta)$. This requires that the formulation of $L(\theta)$ should be non-negative, which means that some objective functions can be shifted by a large constant $A > 0$ such as $f \leftarrow f + A$ if necessary.

By constructing a Markov chain Monte Carlo, we can formulate a generic framework as outlined by Ghate and Smith in 2008, as shown in Figure 2.3. In this framework, simulated annealing and its many variants are simply a special case with

$$P_t = \begin{cases} \exp[-\frac{\Delta f}{T_t}] & \text{if } f_{t+1} > f_t \\ 1 & \text{if } f_{t+1} \leq f_t \end{cases},$$

In this case, only the difference Δf between the function values is important.

Algorithms such as simulated annealing, to be discussed in the next chapter, use a single Markov chain, which may not be very efficient. In practice, it is usually advantageous to use multiple Markov chains in parallel to increase the overall efficiency. In fact, the algorithms such as particle swarm optimization can be viewed as multiple interacting Markov chains, though such theoretical analysis remains almost intractable. The theory of interacting Markov chains is complicated and yet still under development, however, any progress in such areas will play a central role in the understanding how population- and trajectory-based metaheuristic algorithms perform under various conditions. However, even though we do not fully understand why metaheuristic algorithms work, this does not hinder us to

Markov Chain Algorithm for Optimization

Start with $\zeta_0 \in S$, *at* $t = 0$
while *(criterion)*
Propose a new solution Y_{t+1};
Generate a random number $0 \le P_t \le 1$;

$$\zeta_{t+1} = \begin{cases} Y_{t+1} & \text{with probability } P_t \\ \zeta_t & \text{with probability } 1 - P_t \end{cases} \tag{2.27}$$

end

Figure 2.3: Optimization as a Markov chain.

use these algorithms efficiently. On the contrary, such mysteries can drive and motivate us to pursue further research and development in metaheuristics.

REFERENCES

1. W. J. Bell, *Searching Behaviour: The Behavioural Ecology of Finding Resources*, Chapman & Hall, London, (1991).
2. C. Blum and A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptural comparison, *ACM Comput. Surv.*, **35**, 268-308 (2003).
3. G. S. Fishman, *Monte Carlo: Concepts, Algorithms and Applications*, Springer, New York, (1995).
4. D. Gamerman, *Markov Chain Monte Carlo*, Chapman & Hall/CRC, (1997).
5. L. Gerencser, S. D. Hill, Z. Vago, and Z. Vincze, Discrete optimization, SPSA, and Markov chain Monte Carlo methods, *Proc. 2004 Am. Contr. Conf.*, 3814-3819 (2004).
6. C. J. Geyer, Practical Markov Chain Monte Carlo, *Statistical Science*, **7**, 473-511 (1992).
7. A. Ghate and R. Smith, Adaptive search with stochastic acceptance probabilities for global optimization, *Operations Research Lett.*, **36**, 285-290 (2008).
8. W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, *Markov Chain Monte Carlo in Practice*, Chapman & Hall/CRC, (1996).
9. M. Gutowski, Lévy flights as an underlying mechanism for global optimization algorithms, *ArXiv Mathematical Physics e-Prints*, June, (2001).
10. W. K. Hastings, Monte Carlo sampling methods using Markov chains and their applications, *Biometrika*, **57**, 97-109 (1970).
11. S. Kirkpatrick, C. D. Gellat and M. P. Vecchi, Optimization by simulated annealing, *Science*, **220**, 670-680 (1983).

12. R. N. Mantegna, Fast, accurate algorithm for numerical simulation of Levy stable stochastic processes, *Physical Review E*, **49**, 4677-4683 (1994).

13. E. Marinari and G. Parisi, Simulated tempering: a new Monte Carlo scheme, *Europhysics Lett.*, **19**, 451-458 (1992).

14. J. P. Nolan, Stable distributions: models for heavy-tailed data, American University, (2009).

15. N. Metropolis, and S. Ulam, The Monte Carlo method, *J. Amer. Stat. Assoc.*, **44**, 335-341 (1949).

16. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, Equation of state calculations by fast computing machines, *J. Chem. Phys.*, **21**, 1087-1092 (1953).

17. I. Pavlyukevich, Lévy flights, non-local search and simulated annealing, *J. Computational Physics*, **226**, 1830-1844 (2007).

18. G. Ramos-Fernandez, J. L. Mateos, O. Miramontes, G. Cocho, H. Larralde, B. Ayala-Orozco, Lévy walk patterns in the foraging movements of spider monkeys (*Ateles geoffroyi*),*Behav. Ecol. Sociobiol.*, **55**, 223-230 (2004).

19. A. M. Reynolds and M. A. Frye, Free-flight odor tracking in Drosophila is consistent with an optimal intermittent scale-free search, *PLoS One*, **2**, e354 (2007).

20. A. M. Reynolds and C. J. Rhodes, The Lévy flight paradigm: random search patterns and mechanisms, *Ecology*, **90**, 877-887 (2009).

21. I. M. Sobol, *A Primer for the Monte Carlo Method*, CRC Press, (1994).

22. M. E. Tipping M. E., Bayesian inference: An introduction to principles and and practice in machine learning, in: *Advanced Lectures on Machine Learning*, O. Bousquet, U. von Luxburg and G. Ratsch (Eds), pp.41-62 (2004).

23. G. M. Viswanathan, S. V. Buldyrev, S. Havlin, M. G. E. da Luz, E. P. Raposo, and H. E. Stanley, Lévy flight search patterns of wandering albatrosses, *Nature*, **381**, 413-415 (1996).

24. E. Weisstein, http://mathworld.wolfram.com

Chapter 3

SIMULATED ANNEALING

One of the earliest and yet most popular metaheuristic algorithms is simulated annealing (SA), which is a trajectory-based, random search technique for global optimization. It mimics the annealing process in material processing when a metal cools and freezes into a crystalline state with the minimum energy and larger crystal size so as to reduce the defects in metallic structures. The annealing process involves the careful control of temperature and its cooling rate, often called annealing schedule.

3.1 ANNEALING AND BOLTZMANN DISTRIBUTION

Since the first development of simulated annealing by Kirkpatrick, Gelatt and Vecchi in 1983, SA has been applied in almost every area of optimization. Unlike the gradient-based methods and other deterministic search methods which have the disadvantage of being trapped into local minima, the main advantage of simulated annealing is its ability to avoid being trapped in local minima. In fact, it has been proved that simulated annealing will converge to its global optimality if enough randomness is used in combination with very slow cooling. Essentially, simulated annealing is a search algorithm via a Markov chain, which converges under appropriate conditions.

Metaphorically speaking, this is equivalent to dropping some bouncing balls over a landscape, and as the balls bounce and lose energy, they settle down to some local minima. If the balls are allowed to bounce enough times and lose energy slowly enough, some of the balls will eventually fall into the globally lowest locations, hence the global minimum will be reached.

The basic idea of the simulated annealing algorithm is to use random search in terms of a Markov chain, which not only accepts changes that improve the objective function, but also keeps some changes that are not ideal. In a minimization problem, for example, any better moves or changes that decrease the value of the objective function f will be accepted; however, some changes that increase f will also be accepted with a probability p. This probability p, also called the transition probability, is determined

by

$$p = e^{-\frac{\Delta E}{k_B T}}, \tag{3.1}$$

where k_B is the Boltzmann's constant, and for simplicity, we can use k to denote k_B because $k = 1$ is often used. T is the temperature for controlling the annealing process. ΔE is the change of the energy level. This transition probability is based on the Boltzmann distribution in statistical mechanics.

The simplest way to link ΔE with the change of the objective function Δf is to use

$$\Delta E = \gamma \Delta f, \tag{3.2}$$

where γ is a real constant. For simplicity without losing generality, we can use $k_B = 1$ and $\gamma = 1$. Thus, the probability p simply becomes

$$p(\Delta f, T) = e^{-\Delta f/T}. \tag{3.3}$$

Whether or not we accept a change, we usually use a random number r as a threshold. Thus, if $p > r$, or

$$p = e^{-\frac{\Delta f}{T}} > r, \tag{3.4}$$

the move is accepted.

3.2 PARAMETERS

Here the choice of the right initial temperature is crucially important. For a given change Δf, if T is too high ($T \to \infty$), then $p \to 1$, which means almost all the changes will be accepted. If T is too low ($T \to 0$), then any $\Delta f > 0$ (worse solution) will rarely be accepted as $p \to 0$ and thus the diversity of the solution is limited, but any improvement Δf will almost always be accepted. In fact, the special case $T \to 0$ corresponds to the gradient-based method because only better solutions are accepted, and the system is essentially climbing up or descending along a hill. Therefore, if T is too high, the system is at a high energy state on the topological landscape, and the minima are not easily reached. If T is too low, the system may be trapped in a local minimum (not necessarily the global minimum), and there is not enough energy for the system to jump out the local minimum to explore other minima including the global minimum. So a proper initial temperature should be calculated.

Another important issue is how to control the annealing or cooling process so that the system cools down gradually from a higher temperature to ultimately freeze to a global minimum state. There are many ways of controlling the cooling rate or the decrease of the temperature.

Two commonly used annealing schedules (or cooling schedules) are: linear and geometric. For a linear cooling schedule, we have

$$T = T_0 - \beta t, \tag{3.5}$$

or $T \to T - \delta T$, where T_0 is the initial temperature, and t is the pseudo time for iterations. β is the cooling rate, and it should be chosen in such a way that $T \to 0$ when $t \to t_f$ (or the maximum number N of iterations), this usually gives $\beta = (T_0 - T_f)/t_f$.

On the other hand, a geometric cooling schedule essentially decreases the temperature by a cooling factor $0 < \alpha < 1$ so that T is replaced by αT or

$$T(t) = T_0 \alpha^t, \qquad t = 1, 2, ..., t_f. \tag{3.6}$$

The advantage of the second method is that $T \to 0$ when $t \to \infty$, and thus there is no need to specify the maximum number of iterations. For this reason, we will use this geometric cooling schedule. The cooling process should be slow enough to allow the system to stabilize easily. In practise, $\alpha = 0.7 \sim 0.99$ is commonly used.

In addition, for a given temperature, multiple evaluations of the objective function are needed. If too few evaluations, there is a danger that the system will not stabilize and subsequently will not converge to its global optimality. If too many evaluations, it is time-consuming, and the system will usually converge too slowly, as the number of iterations to achieve stability might be exponential to the problem size.

Therefore, there is a fine balance between the number of evaluations and solution quality. We can either do many evaluations at a few temperature levels or do few evaluations at many temperature levels. There are two major ways to set the number of iterations: fixed or varied. The first uses a fixed number of iterations at each temperature, while the second intends to increase the number of iterations at lower temperatures so that the local minima can be fully explored.

3.3 SA ALGORITHM

The simulated annealing algorithm can be summarized as the pseudo code shown in Fig. 3.1.

In order to find a suitable starting temperature T_0, we can use any information about the objective function. If we know the maximum change $\max(\Delta f)$ of the objective function, we can use this to estimate an initial temperature T_0 for a given probability p_0. That is

$$T_0 \approx -\frac{\max(\Delta f)}{\ln p_0}.$$

If we do not know the possible maximum change of the objective function, we can use a heuristic approach. We can start evaluations from a very

Simulated Annealing Algorithm

Objective function $f(\boldsymbol{x})$, $\boldsymbol{x} = (x_1, ..., x_p)^T$
Initialize initial temperature T_0 *and initial guess* $\boldsymbol{x}^{(0)}$
Set final temperature T_f *and max number of iterations* N
Define cooling schedule $T \mapsto \alpha T$, $(0 < \alpha < 1)$
while ($T > T_f$ *and* $n < N$)
  Move randomly to new locations: $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \boldsymbol{\epsilon}$ *(random walk)*
  Calculate $\Delta f = f_{n+1}(\boldsymbol{x}_{n+1}) - f_n(\boldsymbol{x}_n)$
  Accept the new solution if better
  if *not improved*
    Generate a random number r
    Accept if $p = \exp[-\Delta f/T] > r$
  end if
  Update the best $\boldsymbol{x}_*$ *and* f_*
  $n = n + 1$
end while

Figure 3.1: Simulated annealing algorithm.

high temperature (so that almost all changes are accepted) and reduce the temperature quickly until about 50% or 60% of the worse moves are accepted, and then use this temperature as the new initial temperature T_0 for proper and relatively slow cooling.

For the final temperature, it should be zero in theory so that no worse move can be accepted. However, if $T_f \to 0$, more unnecessary evaluations are needed. In practice, we simply choose a very small value, say, $T_f = 10^{-10} \sim 10^{-5}$, depending on the required quality of the solutions and time constraints.

3.4 UNCONSTRAINED OPTIMIZATION

Based on the guidelines of choosing the important parameters such as the cooling rate, initial and final temperatures, and the balanced number of iterations, we can implement the simulated annealing using both Matlab and Octave.

For Rosenbrock's banana function

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2,$$

we know that its global minimum $f_* = 0$ occurs at $(1, 1)$ (see Fig. 3.2). This is a standard test function and quite tough for most algorithms. However, by modifying the program given later in the next chapter, we can find this

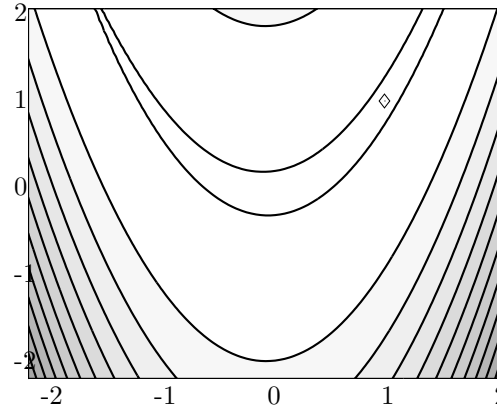

Figure 3.2: Rosenbrock's function with the global minimum $f_* = 0$ at $(1, 1)$.

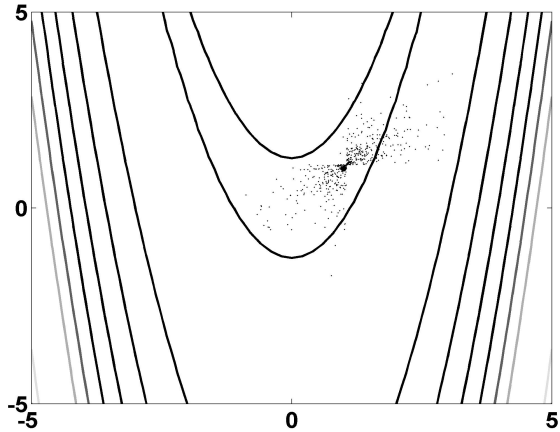

Figure 3.3: 500 evaluations during the annealing iterations. The final global best is marked with •.

global minimum easily and the last 500 evaluations during annealing are shown in Fig. 3.3.

This banana function is still relatively simple as it has a curved narrow valley. We should validate SA against a wide range of test functions, especially those that are strongly multimodal and highly nonlinear. It is straightforward to extend the above program to deal with highly nonlinear multimodal functions.

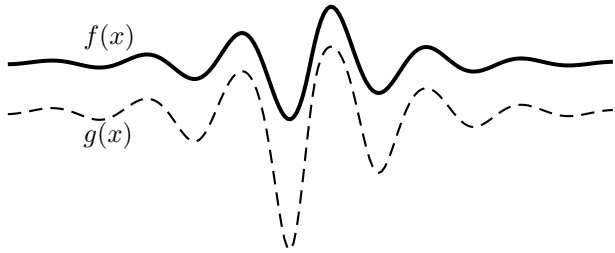

Figure 3.4: The basic idea of stochastic tunneling by transforming $f(x)$ to $g(x)$, suppressing some modes and preserving the locations of minima.

3.5 STOCHASTIC TUNNELING

To ensure the global convergence of simulated annealing, a proper cooling schedule must be used. In the case when the functional landscape is complex, simulated annealing may become increasingly difficult to escape the local minima if the temperature is too low. Raising the temperature, as that in the so-called simulated tempering, may solve the problem, but the convergence is typically slow, and the computing time also increases.

Stochastic tunneling uses the tunneling idea to transform the objective function landscape into a different but more convenient one (e.g., Wenzel and Hamacher, 1999). The essence is to construct a nonlinear transformation so that some modes of $f(x)$ are suppressed and other modes are amplified, while preserving the loci of minima of $f(x)$.

The standard form of such a tunneling transformation is

$$g(x) = 1 - \exp[-\gamma(f(x) - f_0)], \tag{3.7}$$

where f_0 is the current lowest value of $f(x)$ found so far. $\gamma > 0$ is a scaling parameter, and g is the transformed new landscape. From this simple transformation, we can see that $g \to 0$ when $f - f_0 \to 0$, that is when f_0 is approaching the true global minimum. On the other hand, if $f \gg f_0$, then $g \to 1$, which means that all the modes well above the current minimum f_0 are suppressed. For a simple one-dimensional function, it is easy to see that such properties indeed preserve the loci of the function (see Fig. 3.4).

As the loci of the minima are preserved, then all the modes that above the current lowest value f_0 are suppressed to some degree, while the modes below f_0 are expanded or amplified, which makes it easy for the system to escape local modes. Simulations and studies suggest that it can significantly improve the convergence for functions with complex landscape and modes.

Up to now we have not actually provided a detailed program to show how the SA algorithm can be implemented in practice. However, before we can actually do this, we need to find a practical way to deal with con-

straints, as most real-world optimization problems are constrained. In the next chapter, we will discuss in detail the ways of incorporating nonlinear constraints.

REFERENCES

1. Cerny V., A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm, *Journal of Optimization Theory and Applications*, **45**, 41-51 (1985).
2. Hamacher K. and Wenzel W., The scaling behaviour of stochastic minimization algorithms in a perfect funnel landscape, *Phys. rev. E.*, **59**, 938-941 (1999).
3. Kirkpatrick S., Gelatt C. D., and Vecchi M. P., Optimization by simulated annealing, *Science*, **220**, No. 4598, 671-680 (1983).
4. Metropolis N., Rosenbluth A. W., Rosenbluth M. N., Teller A. H., and Teller E., Equations of state calculations by fast computing machines, *Journal of Chemical Physics*, **21**, 1087-1092 (1953).
5. Wenzel W. and Hamacher K., A stochastic tunneling approach for global optimization, *Phys. Rev. Lett.*, **82**, 3003-3007 (1999).
6. Yang X. S., Biology-derived algorithms in engineering optimization (Chapter 32), in *Handbook of Bioinspired Algorithms*, edited by Olariu S. and Zomaya A., Chapman & Hall / CRC, (2005).


Nature-Inspired Metaheuristic Algorithms
Second Edition (2010)
Xin-She Yang
©Luniver Press

Chapter 4

HOW TO DEAL WITH CONSTRAINTS

The optimization we have discussed so far is unconstrained, as we have not considered any constraints. A natural and important question is how to incorporate the constraints (both inequality and equality constraints). There are mainly three ways to deal with constraints: direct approach, Lagrange multipliers, and penalty method.

Direct approach intends to find the feasible regions enclosed by the constraints. This is often difficult, except for a few special cases. Numerically, we can generate a potential solution, and check if all the constraints are satisfied. If all the constraints are met, then it is a feasible solution, and the evaluation of the objective function can be carried out. If one or more constraints are not satisfied, this potential solution is discarded, and a new solution should be generated. We then proceed in a similar manner. As we can expect, this process is slow and inefficient. A better approach is to incorporate the constraints so as to formulate the problem as an unconstrained one. The method of Lagrange multiplier has rigorous mathematical basis, while the penalty method is simple to implement in practice.

4.1 METHOD OF LAGRANGE MULTIPLIERS

The method of Lagrange multipliers converts a constrained problem to an unconstrained one. For example, if we want to minimize a function

$$\underset{\boldsymbol{x}\in\Re^n}{\text{minimize}} f(\boldsymbol{x}), \qquad \boldsymbol{x} = (x_1, ..., x_n)^T \in \Re^n, \tag{4.1}$$

subject to multiple nonlinear equality constraints

$$g_j(\boldsymbol{x}) = 0, \qquad j = 1, 2, ..., M. \tag{4.2}$$

We can use M Lagrange multipliers $\lambda_j (j = 1, ..., M)$ to reformulate the above problem as the minimization of the following function

$$L(\boldsymbol{x}, \lambda_j) = f(\boldsymbol{x}) + \sum_{j=1}^{M} \lambda_j g_j(\boldsymbol{x}). \tag{4.3}$$

The optimality requires that the following stationary conditions hold

$$\frac{\partial L}{\partial x_i} = \frac{\partial f}{\partial x_i} + \sum_{j=1}^{M} \lambda_j \frac{\partial g_j}{\partial x_i}, \quad (i = 1, ..., n), \tag{4.4}$$

and

$$\frac{\partial L}{\partial \lambda_j} = g_j = 0, \quad (j = 1, ..., M). \tag{4.5}$$

These $M + n$ equations will determine the n components of $\boldsymbol{x}$ and M Lagrange multipliers. As $\frac{\partial L}{\partial g_j} = \lambda_j$, we can consider λ_j as the rate of the change of the quantity $L(\boldsymbol{x}, \lambda_j)$ as a functional of g_j.

Now let us look at a simple example

$$\underset{u,v}{\text{maximize}} \ f = u^{2/3} v^{1/3},$$

subject to

$$3u + v = 9.$$

First, we write it as an unconstrained problem using a Lagrange multiplier λ, and we have

$$L = u^{2/3} v^{1/3} + \lambda(3u + v - 9).$$

The conditions to reach optimality are

$$\frac{\partial L}{\partial u} = \frac{2}{3} u^{-1/3} v^{1/3} + 3\lambda = 0, \qquad \frac{\partial L}{\partial v} = \frac{1}{3} u^{2/3} v^{-2/3} + \lambda = 0,$$

and

$$\frac{\partial L}{\partial \lambda} = 3u + v - 9 = 0.$$

The first two conditions give $2v = 3u$, whose combination with the third condition leads to

$$u = 2, \qquad v = 3.$$

Thus, the maximum of f_* is $\sqrt[3]{12}$.

Here we only discussed the equality constraints. For inequality constraints, things become more complicated. We need the so-called Karush-Kuhn-Tucker conditions.

Let us consider the following, generic, nonlinear optimization problem

$$\underset{\boldsymbol{x} \in \Re^n}{\text{minimize}} f(\boldsymbol{x}),$$

$$\text{subject to } \phi_i(\boldsymbol{x}) = 0, \ (i = 1, ..., M),$$

$$\psi_j(\boldsymbol{x}) \le 0, \ \ (j = 1, ..., N). \tag{4.6}$$

If all the functions are continuously differentiable, at a local minimum $\boldsymbol{x}_*$, there exist constants $\lambda_1, ..., \lambda_M$ and $\mu_0, \mu_1, ..., \mu_N$ such that the following KKT optimality conditions hold

$$\mu_0 \nabla f(\boldsymbol{x}_*) + \sum_{i=1}^{M} \lambda_i \nabla \phi_i(\boldsymbol{x}_*) + \sum_{j=1}^{N} \mu_j \nabla \psi_j(\boldsymbol{x}_*) = 0, \tag{4.7}$$

and

$$\psi_j(\boldsymbol{x}_*) \le 0, \qquad \mu_j \psi_j(\boldsymbol{x}_*) = 0, \quad (j = 1, 2, ..., N), \tag{4.8}$$

where

$$\mu_j \ge 0, \quad (j = 0, 1, ..., N). \tag{4.9}$$

The last non-negativity conditions hold for all μ_j, though there is no constraint on the sign of λ_i.

The constants satisfy the following condition

$$\sum_{j=0}^{N} \mu_j + \sum_{i=1}^{M} |\lambda_i| \ge 0. \tag{4.10}$$

This is essentially a generalized method of Lagrange multipliers. However, there is a possibility of degeneracy when $\mu_0 = 0$ under certain conditions. There are two possibilities: 1) there exist vectors $\boldsymbol{\lambda}^* = (\lambda_1^*, ..., \lambda_M^*)^T$ and $\boldsymbol{\mu}^* = (\mu_1^*, .., \mu_N^*)^T$ such that above equations hold, or 2) all the vectors $\nabla\phi_1(\boldsymbol{x}_*), \nabla\phi_2(\boldsymbol{x}_*), ..., \nabla\psi_1(\boldsymbol{x}_*), ..., \nabla\psi_N(\boldsymbol{x}_*)$ are linearly independent, and in this case, the stationary conditions $\frac{\partial L}{\partial x_i}$ do not necessarily hold. As the second case is a special case, we will not discuss this further.

The condition $\mu_j \psi_j(\boldsymbol{x}_*) = 0$ in (4.8) is often called the complementarity condition or complementary slackness condition. It either means $\mu_j = 0$ or $\psi_j(\boldsymbol{x}_*) = 0$. The later case $\psi_j(\boldsymbol{x}_*) = 0$ for any particular j means the inequality becomes tight, and thus becoming an equality. For the former case $\mu_j = 0$, the inequality for a particular j holds and is not tight; however, $\mu_j = 0$ means that this corresponding inequality can be ignored. Therefore, those inequalities that are not tight are ignored, while inequalities which are tight become equalities; consequently, the constrained problem with equality and inequality constraints now essentially becomes a modified constrained problem with selected equality constraints. This is the beauty of the KKT conditions. The main issue remains to identify which inequality becomes tight, and this depends on the individual optimization problem.

The KKT conditions form the basis for mathematical analysis of nonlinear optimization problems, but the numerical implementation of these conditions is not easy, and often inefficient. From the numerical point of view, the penalty method is more straightforward to implement.

4.2 PENALTY METHOD

For a nonlinear optimization problem with equality and inequality constraints, a common method of incorporating constraints is the penalty method. For the optimization problem

$$\begin{aligned} &\underset{\boldsymbol{x}\in\Re^n}{\text{minimize}} f(\boldsymbol{x}), \quad \boldsymbol{x}=(x_1,...,x_n)^T\in\Re^n,\\ &\text{subject to } \phi_i(\boldsymbol{x})=0, \ (i=1,...,M),\\ &\qquad \psi_j(\boldsymbol{x})\le 0, \ \ (j=1,...,N), \end{aligned} \tag{4.11}$$

the idea is to define a penalty function so that the constrained problem is transformed into an unconstrained problem. Now we define

$$\Pi(\boldsymbol{x},\mu_i,\nu_j)=f(\boldsymbol{x})+\sum_{i=1}^{M}\mu_i\phi_i^2(\boldsymbol{x})+\sum_{j=1}^{N}\nu_j\psi_j^2(\boldsymbol{x}), \tag{4.12}$$

where $\mu_i \gg 1$ and $\nu_j \ge 0$ which should be large enough, depending on the solution quality needed.

As we can see, when an equality constraint it met, its effect or contribution to Π is zero. However, when it is violated, it is penalized heavily as it increases Π significantly. Similarly, it is true when inequality constraints become tight or exact. For the ease of numerical implementation, we should use index functions H to rewrite above penalty function as

$$\Pi=f(\boldsymbol{x})+\sum_{i=1}^{M}\mu_i H_i[\phi_i(\boldsymbol{x})]\phi_i^2(\boldsymbol{x})+\sum_{j=1}^{N}\nu_j H_j[\psi_j(\boldsymbol{x})]\psi_j^2(\boldsymbol{x}), \tag{4.13}$$

Here $H_i[\phi_i(\boldsymbol{x})]$ and $H_j[\psi_j(\boldsymbol{x})]$ are index functions.

More specifically, $H_i[\phi_i(\boldsymbol{x})]=1$ if $\phi_i(\boldsymbol{x})\neq 0$, and $H_i=0$ if $\phi_i(\boldsymbol{x})=0$. Similarly, $H_j[\psi_j(\boldsymbol{x})]=0$ if $\psi_j(\boldsymbol{x})\le 0$ is true, while $H_j=1$ if $\psi_j(\boldsymbol{x})>0$. In principle, the numerical accuracy depends on the values of μ_i and ν_j which should be reasonably large. But how large is large enough? As most computers have a machine precision of $\epsilon=2^{-52}\approx 2.2\times 10^{-16}$, μ_i and ν_j should be close to the order of 10^{15}. Obviously, it could cause numerical problems if they are too large.

In addition, for simplicity of implementation, we can use $\mu=\mu_i$ for all i and $\nu=\nu_j$ for all j. That is, we can use a simplified

$$\Pi(\boldsymbol{x},\mu,\nu)=f(\boldsymbol{x})+\mu\sum_{i=1}^{M}H_i[\phi_i(\boldsymbol{x})]\phi_i^2(\boldsymbol{x})+\nu\sum_{j=1}^{N}H_j[\psi_j(\boldsymbol{x})]\psi_j^2(\boldsymbol{x}).$$

In general, for most applications, μ and ν can be taken as 10^{10} to 10^{15}. We will use these values in our implementation.

Sometimes, it might be easier to change an equality constraint to two inequality constraints, so that we only have to deal with inequalities in the implementation. This is because $g(\boldsymbol{x}) = 0$ is always equivalent to $g(\boldsymbol{x}) \le 0$ and $g(\boldsymbol{x}) \ge 0$ (or $-g(\boldsymbol{x}) \le 0$).

4.3 STEP SIZE IN RANDOM WALKS

As random walks are widely used for randomization and local search, a proper step size is very important. In the generic equation

$$\boldsymbol{x}^{t+1} = \boldsymbol{x}^t + s\,\boldsymbol{\epsilon}_t, \tag{4.14}$$

$\boldsymbol{\epsilon}_t$ is drawn from a standard normal distribution with zero mean and unity standard deviation. Here the step size s determines how far a random walker (e.g., an agent or particle in metaheuristics) can go for a fixed number of iterations.

If s is too large, then the new solution $\boldsymbol{x}^{t+1}$ generated will be too far away from the old solution (or more often the current best). Then, such a move is unlikely to be accepted. If s is too small, the change is too small to be significant, and consequently such search is not efficient. So a proper step size is important to maintain the search as efficient as possible.

From the theory of simple isotropic random walks, we know that the average distance r traveled in the d-dimension space is

$$r^2 = 2dDt, \tag{4.15}$$

where $D = s^2/2\tau$ is the effective diffusion coefficient. Here s is the step size or distance traveled at each jump, and τ is the time taken for each jump. The above equation implies that

$$s^2 = \frac{\tau\, r^2}{t\, d}. \tag{4.16}$$

For a typical length scale L of a dimension of interest, the local search is typically limited in a region of $L/10$. That is, $r = L/10$. As the iterations are discrete, we can take $\tau = 1$. Typically in metaheuristics, we can expect that the number of generations is usually $t = 100$ to 1000, which means that

$$s \approx \frac{r}{\sqrt{td}} = \frac{L/10}{\sqrt{t\, d}}. \tag{4.17}$$

For $d = 1$ and $t = 100$, we have $s = 0.01L$, while $s = 0.001L$ for $d = 10$ and $t = 1000$. As step sizes could differ from variable to variable, a step size ratio s/L is more generic. Therefore, we can use $s/L = 0.001$ to 0.01 for most problems. We will use this step size factor in our implementation, to be discussed later in the last section of this chapter.

In order to demonstrate the way we incorporate the constraints and the way to do the random walk, it is easy to illustrate using a real-world design example in engineering applications. Now let us look at the well-known welded beam design.

4.4 WELDED BEAM DESIGN

The welded beam design problem is a standard test problem for constrained design optimization, which was described in detail in the literature (Ragsdell and Phillips 1976, Cagnina et al 2008). The problem has four design variables: the width w and length L of the welded area, the depth d and thickness h of the beam. The objective is to minimize the overall fabrication cost, under the appropriate constraints of shear stress τ, bending stress σ, buckling load P and end deflection δ.

The problem can be written as

$$\text{minimize} \quad f(\boldsymbol{x}) = 1.10471 w^2 L + 0.04811 dh(14.0 + L), \tag{4.18}$$

subject to

$$\begin{aligned} g_1(\boldsymbol{x}) &= \tau(\boldsymbol{x}) - 13,600 \le 0 \\ g_2(\boldsymbol{x}) &= \sigma(\boldsymbol{x}) - 30,000 \le 0 \\ g_3(\boldsymbol{x}) &= w - h \le 0 \\ g_4(\boldsymbol{x}) &= 0.10471 w^2 + 0.04811 hd(14 + L) - 5.0 \le 0 \\ g_5(\boldsymbol{x}) &= 0.125 - w \le 0 \\ g_6(\boldsymbol{x}) &= \delta(\boldsymbol{x}) - 0.25 \le 0 \\ g_7(\boldsymbol{x}) &= 6000 - P(\boldsymbol{x}) \le 0, \end{aligned} \tag{4.19}$$

where

$$\sigma(\boldsymbol{x}) = \frac{504,000}{hd^2}, \quad \delta = \frac{65,856}{30,000 hd^3}, \quad Q = 6000(14 + \frac{L}{2}),$$

$$D = \frac{1}{2}\sqrt{L^2 + (w+d)^2}, \quad J = \sqrt{2}\, wL[\frac{L^2}{6} + \frac{(w+d)^2}{2}], \quad \beta = \frac{QD}{J},$$

$$\alpha = \frac{6000}{\sqrt{2} wL}, \quad \tau(\boldsymbol{x}) = \sqrt{\alpha^2 + \frac{\alpha\beta L}{D} + \beta^2},$$

$$P = 0.61423 \times 10^6 \, \frac{dh^3}{6}(1 - \frac{d\sqrt{30/48}}{28}). \tag{4.20}$$

The simple limits or bounds are $0.1 \le L, d \le 10$ and $0.1 \le w, h \le 2.0$.

If we use the simulated annealing algorithm to solve this problem (see next section), we can get the optimal solution which is about the same solution obtained by Cagnina et al (2008)

$$f_* = 1.724852 \text{ at } \quad (0.205730, 3.470489, 9.036624, 0.205729). \tag{4.21}$$

It is worth pointing out that you have to run the programs a few times using values such as $\alpha = 0.95$ (default) and $\alpha = 0.99$ to see how the results vary. In addition, as SA is a stochastic optimization algorithm, we cannot expect the results are the same. In fact, they will be slightly different, every time we run the program. Therefore, we should understand and interpret the results using statistical measures such as mean and standard deviation.

4.5 SA IMPLEMENTATION

We just formulated the welded beam design problem using different notations from some literature. Here we try to illustrate a point.

As the input to a function is a vector (either column vector or less often row vector), we have to write

$$\boldsymbol{x} = \begin{pmatrix} w & L & d & h \end{pmatrix} = [x(1) \;\; x(2) \;\; x(3) \;\; x(4)]. \tag{4.22}$$

With this vector, the objective becomes

$$\text{minimize } f(\boldsymbol{x}) = 1.10471 * x(1)^2 * x(2) + 0.04811 * x(3) * x(4)(14.0 + x(2)),$$

which can easily be converted to a formula in Matlab. Similarly, the third inequality constraint can be rewritten as

$$g_3 = g(3) = x(1) - x(4) \le 0. \tag{4.23}$$

Other constraints can be rewritten in a similar way.

Using the pseudo code for simulated annealing and combining with the penalty method, we can solve the above welded beam design problem using simulated annealing in Matlab as follows:

```
% Simulated Annealing for constrained optimization
% by Xin-She Yang @ Cambridge University @2008
% Usage: sa_mincon(0.99) or sa_mincon;

function [bestsol,fval,N]=sa_mincon(alpha)
% Default cooling factor
if nargin<1,
    alpha=0.95;
end

% Display usage
disp('sa_mincon or [Best,fmin,N]=sa_mincon(0.9)');

% Welded beam design optimization
Lb=[0.1 0.1  0.1 0.1];
```

```
Ub=[2.0 10.0 10.0 2.0];
u0=(Lb+Ub)/2;

if length(Lb) ~=length(Ub),
    disp('Simple bounds/limits are improper!');
    return
end

%% Start of the main program -------------------------
d=length(Lb);       % Dimension of the problem

% Initializing parameters and settings
T_init = 1.0;       % Initial temperature
T_min =  1e-10;     % Finial stopping temperature
F_min = -1e+100;    % Min value of the function
max_rej=500;        % Maximum number of rejections
max_run=150;        % Maximum number of runs
max_accept = 50;    % Maximum number of accept
initial_search=500; % Initial search period
k = 1;              % Boltzmann constant
Enorm=1e-5;         % Energy norm (eg, Enorm=1e-8)

% Initializing the counters i,j etc
i= 0; j = 0; accept = 0; totaleval = 0;
% Initializing various values
T = T_init;
E_init = Fun(u0);
E_old = E_init; E_new=E_old;
best=u0;  % initially guessed values
% Starting the simulated annealing
while ((T > T_min) & (j <= max_rej) & E_new>F_min)
    i = i+1;
    % Check if max numbers of run/accept are met
    if (i >= max_run) | (accept >= max_accept)
        % reset the counters
        i = 1;  accept = 1;
      % Cooling according to a cooling schedule
        T = cooling(alpha,T);
    end

    % Function evaluations at new locations
    if totaleval<initial_search,
        init_flag=1;
        ns=newsolution(u0,Lb,Ub,init_flag);
```

```
    else
        init_flag=0;
        ns=newsolution(best,Lb,Ub,init_flag);
    end
      totaleval=totaleval+1;
      E_new = Fun(ns);
    % Decide to accept the new solution
    DeltaE=E_new-E_old;
    % Accept if improved
    if (DeltaE <0)
        best = ns; E_old = E_new;
        accept=accept+1;   j = 0;
    end
    % Accept with a probability if not improved
    if (DeltaE>=0 & exp(-DeltaE/(k*T))>rand );
        best = ns; E_old = E_new;
        accept=accept+1;
    else
        j=j+1;
    end
    % Update the estimated optimal solution
    f_opt=E_old;
end

bestsol=best;
fval=f_opt;
N=totaleval;

%% New solutions
function s=newsolution(u0,Lb,Ub,init_flag)
  % Either search around
if length(Lb)>0 & init_flag==1,
  s=Lb+(Ub-Lb).*rand(size(u0));
else
  % Or local search by random walk
  stepsize=0.01;
  s=u0+stepsize*(Ub-Lb).*randn(size(u0));
end

s=bounds(s,Lb,Ub);

%% Cooling
function T=cooling(alpha,T)
T=alpha*T;
```

```
function ns=bounds(ns,Lb,Ub)
if length(Lb)>0,
% Apply the lower bound
  ns_tmp=ns;
  I=ns_tmp<Lb;
  ns_tmp(I)=Lb(I);
  % Apply the upper bounds
  J=ns_tmp>Ub;
  ns_tmp(J)=Ub(J);
% Update this new move
  ns=ns_tmp;
else
  ns=ns;
end

% d-dimensional objective function
function z=Fun(u)

% Objective
z=fobj(u);

% Apply nonlinear constraints by penalty method
% Z=f+sum_k=1^N lam_k g_k^2 *H(g_k)
z=z+getnonlinear(u);

function Z=getnonlinear(u)
Z=0;
% Penalty constant
lam=10^15; lameq=10^15;
[g,geq]=constraints(u);

% Inequality constraints
for k=1:length(g),
    Z=Z+ lam*g(k)^2*getH(g(k));
end

% Equality constraints (when geq=[], length->0)
for k=1:length(geq),
   Z=Z+lameq*geq(k)^2*geteqH(geq(k));
end

% Test if inequalities hold
function H=getH(g)
```

```
if g<=0,
    H=0;
else
    H=1;
end

% Test if equalities hold
function H=geteqH(g)
if g==0,
    H=0;
else
    H=1;
end

% Objective functions
function z=fobj(u)
% Welded beam design optimization
z=1.10471*u(1)^2*u(2)+0.04811*u(3)*u(4)*(14.0+u(2));

% All constraints
function [g,geq]=constraints(x)
% Inequality constraints
Q=6000*(14+x(2)/2);
D=sqrt(x(2)^2/4+(x(1)+x(3))^2/4);
J=2*(x(1)*x(2)*sqrt(2)*(x(2)^2/12+(x(1)+x(3))^2/4));
alpha=6000/(sqrt(2)*x(1)*x(2));
beta=Q*D/J;
tau=sqrt(alpha^2+2*alpha*beta*x(2)/(2*D)+beta^2);
sigma=504000/(x(4)*x(3)^2);
delta=65856000/(30*10^6*x(4)*x(3)^3);
tmpf=4.013*(30*10^6)/196;
P=tmpf*sqrt(x(3)^2*x(4)^6/36)*(1-x(3)*sqrt(30/48)/28);

g(1)=tau-13600;
g(2)=sigma-30000;
g(3)=x(1)-x(4);
g(4)=0.10471*x(1)^2+0.04811*x(3)*x(4)*(14+x(2))-5.0;
g(5)=0.125-x(1);
g(6)=delta-0.25;
g(7)=6000-P;

% Equality constraints
geq=[];
%% End of the program --------------------------------
```

How to Get the Files

To get the files of all the Matlab programs provided in this book, readers can send an email (with the subject 'Nature-Inspired Algorithms: Files') to `Metaheuristic.Algorithms@gmail.com` – A zip file will be provided (via email) by the author.

REFERENCES

1. Cagnina L. C., Esquivel S. C., and Coello C. A., Solving engineering optimization problems with the simple constrained particle swarm optimizer, *Informatica*, **32**, 319-326 (2008)
2. Cerny V., A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm, *Journal of Optimization Theory and Applications*, **45**, 41-51 (1985).
3. Deb K., *Optimisation for Engineering Design*: Algorithms and Examples, Prentice-Hall, New Delhi, (1995).
4. Gill P. E., Murray W., and Wright M. H., *Practical optimization*, Academic Press Inc, (1981).
5. Hamacher K., Wenzel W., The scaling behaviour of stochastic minimization algorithms in a perfect funnel landscape, *Phys. Rev. E.*, **59**, 938-941(1999).
6. Kirkpatrick S., Gelatt C. D., and Vecchi M. P., Optimization by simulated annealing, *Science*, **220**, No. 4598, 671-680 (1983).
7. Metropolis N., Rosenbluth A. W., Rosenbluth M. N., Teller A. H., and Teller E., Equations of state calculations by fast computing machines, *Journal of Chemical Physics*, **21**, 1087-1092 (1953).
8. Ragsdell K. and Phillips D., Optimal design of a class of welded structures using geometric programming, *J. Eng. Ind.*, **98**, 1021-1025 (1976).
9. Wenzel W. and Hamacher K., A stochastic tunneling approach for global optimization, *Phys. Rev. Lett.*, **82**, 3003-3007 (1999).
10. Yang X. S., Biology-derived algorithms in engineering optimization (Chapter 32), in *Handbook of Bioinspired Algorithms*, edited by Olariu S. and Zomaya A., Chapman & Hall / CRC, (2005).
11. E. G. Talbi, *Metaheuristics: From Design to Implementation*, Wiley, (2009).

Chapter 5

GENETIC ALGORITHMS

Genetic algorithms are probably the most popular evolutionary algorithms in terms of the diversity of their applications. A vast majority of well-known optimization problems have been tried by genetic algorithms. In addition, genetic algorithms are population-based, and many modern evolutionary algorithms are directly based on genetic algorithms, or have some strong similarities.

5.1 INTRODUCTION

The genetic algorithm (GA), developed by John Holland and his collaborators in the 1960s and 1970s, is a model or abstraction of biological evolution based on Charles Darwin's theory of natural selection. Holland was the first to use the crossover and recombination, mutation, and selection in the study of adaptive and artificial systems. These genetic operators form the essential part of the genetic algorithm as a problem-solving strategy. Since then, many variants of genetic algorithms have been developed and applied to a wide range of optimization problems, from graph colouring to pattern recognition, from discrete systems (such as the travelling salesman problem) to continuous systems (e.g., the efficient design of airfoil in aerospace engineering), and from financial market to multiobjective engineering optimization.

There are many advantages of genetic algorithms over traditional optimization algorithms, and two most noticeable advantages are: the ability of dealing with complex problems and parallelism. Genetic algorithms can deal with various types of optimization whether the objective (fitness) function is stationary or non-stationary (change with time), linear or nonlinear, continuous or discontinuous, or with random noise. As multiple offsprings in a population act like independent agents, the population (or any subgroup) can explore the search space in many directions simultaneously. This feature makes it ideal to parallelize the algorithms for implementation. Different parameters and even different groups of encoded strings can be manipulated at the same time.

Chapter 10

FIREFLY ALGORITHM

10.1 BEHAVIOUR OF FIREFLIES

The flashing light of fireflies is an amazing sight in the summer sky in the tropical and temperate regions. There are about two thousand firefly species, and most fireflies produce short and rhythmic flashes. The pattern of flashes is often unique for a particular species. The flashing light is produced by a process of bioluminescence, and the true functions of such signaling systems are still being debated. However, two fundamental functions of such flashes are to attract mating partners (communication), and to attract potential prey. In addition, flashing may also serve as a protective warning mechanism to remind potential predators of the bitter taste of fireflies.

The rhythmic flash, the rate of flashing and the amount of time form part of the signal system that brings both sexes together. Females respond to a male's unique pattern of flashing in the same species, while in some species such as *Photuris*, female fireflies can eavesdrop on the bioluminescent courtship signals and even mimic the mating flashing pattern of other species so as to lure and eat the male fireflies who may mistake the flashes as a potential suitable mate. Some tropic fireflies can even synchronize their flashes, thus forming emerging biological self-organized behavior.

We know that the light intensity at a particular distance r from the light source obeys the inverse square law. That is to say, the light intensity I decreases as the distance r increases in terms of $I \propto 1/r^2$. Furthermore, the air absorbs light which becomes weaker and weaker as the distance increases. These two combined factors make most fireflies visual to a limit distance, usually several hundred meters at night, which is good enough for fireflies to communicate.

The flashing light can be formulated in such a way that it is associated with the objective function to be optimized, which makes it possible to formulate new optimization algorithms. In the rest of this chapter, we will first outline the basic formulation of the Firefly Algorithm (FA) and then discuss the implementation in detail.

Firefly Algorithm

Objective function $f(\boldsymbol{x})$, $\boldsymbol{x} = (x_1, ..., x_d)^T$
Generate initial population of fireflies $\boldsymbol{x}_i$ $(i = 1, 2, ..., n)$
Light intensity I_i *at* $\boldsymbol{x}_i$ *is determined by* $f(\boldsymbol{x}_i)$
Define light absorption coefficient γ
while *(t <MaxGeneration)*
for $i = 1 : n$ *all n fireflies*
  for $j = 1 : n$ *all n fireflies (inner loop)*
  if *($I_i < I_j$), Move firefly i towards j;* **end if**
    Vary attractiveness with distance r via $\exp[-\gamma r]$
    Evaluate new solutions and update light intensity
  end for j
end for i
Rank the fireflies and find the current global best $\boldsymbol{g}_*$
end while
Postprocess results and visualization

Figure 10.1: Pseudo code of the firefly algorithm (FA).

10.2 FIREFLY ALGORITHM

Now we can idealize some of the flashing characteristics of fireflies so as to develop firefly-inspired algorithms. For simplicity in describing our new Firefly Algorithm (FA) which was developed by Xin-She Yang at Cambridge University in 2007, we now use the following three idealized rules:

- All fireflies are unisex so that one firefly will be attracted to other fireflies regardless of their sex;
- Attractiveness is proportional to the their brightness, thus for any two flashing fireflies, the less brighter one will move towards the brighter one. The attractiveness is proportional to the brightness and they both decrease as their distance increases. If there is no brighter one than a particular firefly, it will move randomly;
- The brightness of a firefly is affected or determined by the landscape of the objective function.

For a maximization problem, the brightness can simply be proportional to the value of the objective function. Other forms of brightness can be defined in a similar way to the fitness function in genetic algorithms. Based on these three rules, the basic steps of the firefly algorithm (FA) can be summarized as the pseudo code shown in Figure 11.1.

10.3 LIGHT INTENSITY AND ATTRACTIVENESS

In the firefly algorithm, there are two important issues: the variation of light intensity and formulation of the attractiveness. For simplicity, we can always assume that the attractiveness of a firefly is determined by its brightness which in turn is associated with the encoded objective function.

In the simplest case for maximum optimization problems, the brightness I of a firefly at a particular location $\boldsymbol{x}$ can be chosen as $I(\boldsymbol{x}) \propto f(\boldsymbol{x})$. However, the attractiveness β is relative, it should be seen in the eyes of the beholder or judged by the other fireflies. Thus, it will vary with the distance r_{ij} between firefly i and firefly j. In addition, light intensity decreases with the distance from its source, and light is also absorbed in the media, so we should allow the attractiveness to vary with the degree of absorption.

In the simplest form, the light intensity $I(r)$ varies according to the inverse square law

$$I(r) = \frac{I_s}{r^2}, \tag{10.1}$$

where I_s is the intensity at the source. For a given medium with a fixed light absorption coefficient γ, the light intensity I varies with the distance r. That is

$$I = I_0 e^{-\gamma r}, \tag{10.2}$$

where I_0 is the original light intensity. In order to avoid the singularity at $r = 0$ in the expression I_s/r^2, the combined effect of both the inverse square law and absorption can be approximated as the following Gaussian form

$$I(r) = I_0 e^{-\gamma r^2}. \tag{10.3}$$

As a firefly's attractiveness is proportional to the light intensity seen by adjacent fireflies, we can now define the attractiveness β of a firefly by

$$\beta = \beta_0 e^{-\gamma r^2}, \tag{10.4}$$

where β_0 is the attractiveness at $r = 0$. As it is often faster to calculate $1/(1 + r^2)$ than an exponential function, the above function, if necessary, can conveniently be approximated as

$$\beta = \frac{\beta_0}{1 + \gamma r^2}. \tag{10.5}$$

Both (10.4) and (10.5) define a characteristic distance $\Gamma = 1/\sqrt{\gamma}$ over which the attractiveness changes significantly from β_0 to $\beta_0 e^{-1}$ for equation (10.4) or $\beta_0/2$ for equation (10.5).

In the actual implementation, the attractiveness function $\beta(r)$ can be any monotonically decreasing functions such as the following generalized form

$$\beta(r) = \beta_0 e^{-\gamma r^m}, \qquad (m \geq 1). \tag{10.6}$$

For a fixed γ, the characteristic length becomes

$$\Gamma = \gamma^{-1/m} \to 1, \qquad m \to \infty. \tag{10.7}$$

Conversely, for a given length scale Γ in an optimization problem, the parameter γ can be used as a typical initial value. That is

$$\gamma = \frac{1}{\Gamma^m}. \tag{10.8}$$

The distance between any two fireflies i and j at $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, respectively, is the Cartesian distance

$$r_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\| = \sqrt{\sum_{k=1}^{d}(x_{i,k} - x_{j,k})^2}, \tag{10.9}$$

where $x_{i,k}$ is the kth component of the spatial coordinate $\boldsymbol{x}_i$ of ith firefly. In 2-D case, we have

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \tag{10.10}$$

The movement of a firefly i is attracted to another more attractive (brighter) firefly j is determined by

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \beta_0 e^{-\gamma r_{ij}^2}(\boldsymbol{x}_j - \boldsymbol{x}_i) + \alpha\, \boldsymbol{\epsilon}_i, \tag{10.11}$$

where the second term is due to the attraction. The third term is randomization with α being the randomization parameter, and $\boldsymbol{\epsilon}_i$ is a vector of random numbers drawn from a Gaussian distribution or uniform distribution. For example, the simplest form is $\boldsymbol{\epsilon}_i$ can be replaced by $\texttt{rand} - 1/2$ where `rand` is a random number generator uniformly distributed in $[0, 1]$. For most our implementation, we can take $\beta_0 = 1$ and $\alpha \in [0, 1]$.

It is worth pointing out that (10.11) is a random walk biased towards the brighter fireflies. If $\beta_0 = 0$, it becomes a simple random walk. Furthermore, the randomization term can easily be extended to other distributions such as Lévy flights.

The parameter γ now characterizes the variation of the attractiveness, and its value is crucially important in determining the speed of the convergence and how the FA algorithm behaves. In theory, $\gamma \in [0, \infty)$, but in practice, $\gamma = O(1)$ is determined by the characteristic length Γ of the system to be optimized. Thus, for most applications, it typically varies from 0.1 to 10.

10.4 SCALINGS AND ASYMPTOTICS

It is worth pointing out that the distance r defined above is *not* limited to the Euclidean distance. We can define other distance r in the n-dimensional

hyperspace, depending on the type of problem of our interest. For example, for job scheduling problems, r can be defined as the time lag or time interval. For complicated networks such as the Internet and social networks, the distance r can be defined as the combination of the degree of local clustering and the average proximity of vertices. In fact, any measure that can effectively characterize the quantities of interest in the optimization problem can be used as the 'distance' r.

The typical scale Γ should be associated with the scale concerned in our optimization problem. If Γ is the typical scale for a given optimization problem, for a very large number of fireflies $n \gg k$ where k is the number of local optima, then the initial locations of these n fireflies should distribute relatively uniformly over the entire search space. As the iterations proceed, the fireflies would converge into all the local optima (including the global ones). By comparing the best solutions among all these optima, the global optima can easily be achieved. Our recent research suggests that it is possible to prove that the firefly algorithm will approach global optima when $n \to \infty$ and $t \gg 1$. In reality, it converges very quickly and this will be demonstrated later in this chapter.

There are two important limiting or asymptotic cases when $\gamma \to 0$ and $\gamma \to \infty$. For $\gamma \to 0$, the attractiveness is constant $\beta = \beta_0$ and $\Gamma \to \infty$, this is equivalent to saying that the light intensity does not decrease in an idealized sky. Thus, a flashing firefly can be seen anywhere in the domain. Thus, a single (usually global) optima can easily be reached. If we remove the inner loop for j in Figure 11.1 and replace $\boldsymbol{x}_j$ by the current global best $\boldsymbol{g}_*$, then the Firefly Algorithm becomes the special case of accelerated particle swarm optimization (PSO) discussed earlier. Subsequently, the efficiency of this special case is the same as that of PSO.

On the other hand, the limiting case $\gamma \to \infty$ leads to $\Gamma \to 0$ and $\beta(r) \to \delta(r)$ which is the Dirac delta function, which means that the attractiveness is almost zero in the sight of other fireflies. This is equivalent to the case where the fireflies roam in a very thick foggy region randomly. No other fireflies can be seen, and each firefly roams in a completely random way. Therefore, this corresponds to the completely random search method.

As the firefly algorithm is usually in the case between these two extremes, it is possible to adjust the parameter γ and α so that it can outperform both the random search and PSO. In fact, FA can find the global optima as well as the local optima simultaneously and effectively. This advantage will be demonstrated in detail later in the implementation.

A further advantage of FA is that different fireflies will work almost independently, it is thus particular suitable for parallel implementation. It is even better than genetic algorithms and PSO because fireflies aggregate more closely around each optimum. It can be expected that the interactions between different subregions are minimal in parallel implementation.

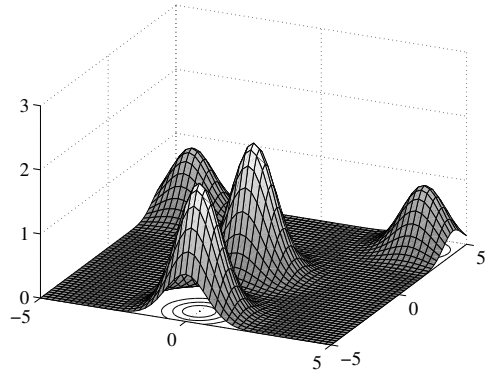

Figure 10.2: Landscape of a function with two equal global maxima.

10.5 IMPLEMENTATION

In order to demonstrate how the firefly algorithm works, we have implemented it in Matlab/Octave to be given later.

In order to show that both the global optima and local optima can be found simultaneously, we now use the following four-peak function

$$f(x,y) = e^{-(x-4)^2-(y-4)^2} + e^{-(x+4)^2-(y-4)^2} + 2[e^{-x^2-y^2} + e^{-x^2-(y+4)^2}],$$

where $(x,y) \in [-5,5] \times [-5,5]$. This function has four peaks. Two local peaks with $f = 1$ at $(-4,4)$ and $(4,4)$, and two global peaks with $f_{\max} = 2$ at $(0,0)$ and $(0,-4)$, as shown in Figure 10.2. We can see that all these four optima can be found using 25 fireflies in about 20 generations (see Fig. 10.3). So the total number of function evaluations is about 500. This is much more efficient than most of existing metaheuristic algorithms.

```
% Firefly Algorithm by X S Yang (Cambridge University)
% Usage: ffa_demo([number_of_fireflies,MaxGeneration])
%  eg:   ffa_demo([12,50]);
function [best]=firefly_simple(instr)
% n=number of fireflies
% MaxGeneration=number of pseudo time steps
if nargin<1,   instr=[12 50];     end
n=instr(1);  MaxGeneration=instr(2);
rand('state',0);  % Reset the random generator
% ------ Four peak functions ---------------------
str1='exp(-(x-4)^2-(y-4)^2)+exp(-(x+4)^2-(y-4)^2)';
str2='+2*exp(-x^2-(y+4)^2)+2*exp(-x^2-y^2)';
funstr=strcat(str1,str2);
% Converting to an inline function
f=vectorize(inline(funstr));
```

```
% range=[xmin xmax ymin ymax];
range=[-5 5 -5 5];

% ------------------------------------------------
alpha=0.2;      % Randomness 0--1 (highly random)
gamma=1.0;      % Absorption coefficient
% ------------------------------------------------
% Grid values are used for display only
Ngrid=100;
dx=(range(2)-range(1))/Ngrid;
dy=(range(4)-range(3))/Ngrid;
[x,y]=meshgrid(range(1):dx:range(2),...
               range(3):dy:range(4));
z=f(x,y);
% Display the shape of the objective function
figure(1);    surfc(x,y,z);

% ------------------------------------------------
% generating the initial locations of n fireflies
[xn,yn,Lightn]=init_ffa(n,range);
% Display the paths of fireflies in a figure with
% contours of the function to be optimized
 figure(2);
% Iterations or pseudo time marching
for i=1:MaxGeneration,     %%%%% start iterations
% Show the contours of the function
 contour(x,y,z,15); hold on;
% Evaluate new solutions
zn=f(xn,yn);

% Ranking the fireflies by their light intensity
[Lightn,Index]=sort(zn);
xn=xn(Index); yn=yn(Index);
xo=xn;   yo=yn;    Lighto=Lightn;
% Trace the paths of all roaming  fireflies
plot(xn,yn,'.','markersize',10,'markerfacecolor','g');
% Move all fireflies to the better locations
[xn,yn]=ffa_move(xn,yn,Lightn,xo,yo,...
        Lighto,alpha,gamma,range);
drawnow;
% Use "hold on" to show the paths of fireflies
    hold off;
end   %%%%% end of iterations
best(:,1)=xo'; best(:,2)=yo'; best(:,3)=Lighto';

% ----- All subfunctions are listed here ---------
% The initial locations of n fireflies
function [xn,yn,Lightn]=init_ffa(n,range)
```

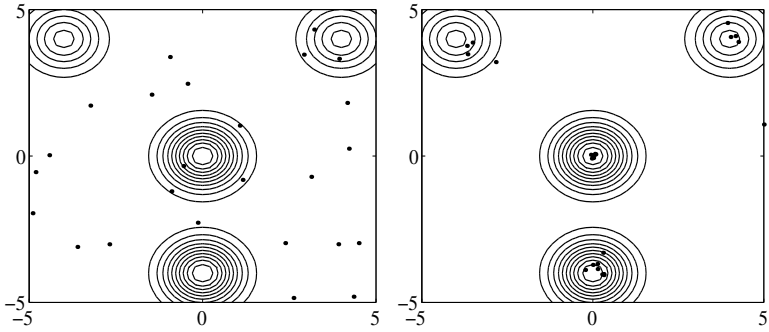

Figure 10.3: The initial locations of 25 fireflies (left) and their final locations after 20 iterations (right).

```
xrange=range(2)-range(1);
yrange=range(4)-range(3);
xn=rand(1,n)*xrange+range(1);
yn=rand(1,n)*yrange+range(3);
Lightn=zeros(size(yn));

% Move all fireflies toward brighter ones
function [xn,yn]=ffa_move(xn,yn,Lightn,xo,yo,...
    Lighto,alpha,gamma,range)
ni=size(yn,2); nj=size(yo,2);
for i=1:ni,
% The attractiveness parameter beta=exp(-gamma*r)
    for j=1:nj,
r=sqrt((xn(i)-xo(j))^2+(yn(i)-yo(j))^2);
if Lightn(i)<Lighto(j), % Brighter and more attractive
beta0=1;     beta=beta0*exp(-gamma*r.^2);
xn(i)=xn(i).*(1-beta)+xo(j).*beta+alpha.*(rand-0.5);
yn(i)=yn(i).*(1-beta)+yo(j).*beta+alpha.*(rand-0.5);
end
    end % end for j
end % end for i
[xn,yn]=findrange(xn,yn,range);

% Make sure the fireflies are within the range
function [xn,yn]=findrange(xn,yn,range)
for i=1:length(yn),
    if xn(i)<=range(1), xn(i)=range(1); end
    if xn(i)>=range(2), xn(i)=range(2); end
    if yn(i)<=range(3), yn(i)=range(3); end
    if yn(i)>=range(4), yn(i)=range(4); end
end
```

In the implementation, the values of the parameters are $\alpha = 0.2$, $\gamma = 1$ and $\beta_0 = 1$. Obviously, these parameters can be adjusted to suit for solving various problems with different scales.

10.6 FA VARIANTS

The basic firefly algorithm is very efficient, but we can see that the solutions are still changing as the optima are approaching. It is possible to improve the solution quality by reducing the randomness.

A further improvement on the convergence of the algorithm is to vary the randomization parameter α so that it decreases gradually as the optima are approaching. For example, we can use

$$\alpha = \alpha_\infty + (\alpha_0 - \alpha_\infty)e^{-t}, \tag{10.12}$$

where $t \in [0, t_{\max}]$ is the pseudo time for simulations and $t_{\max}$ is the maximum number of generations. α_0 is the initial randomization parameter while α_∞ is the final value. We can also use a similar function to the geometrical annealing schedule. That is

$$\alpha = \alpha_0 \theta^t, \tag{10.13}$$

where $\theta \in (0, 1]$ is the randomness reduction constant.

In addition, in the current version of the FA algorithm, we do not explicitly use the current global best $\boldsymbol{g}_*$, even though we only use it to decode the final best solutions. Our recent studies show that the efficiency may significantly improve if we add an extra term $\lambda \boldsymbol{\epsilon}_i (x_i - \boldsymbol{g}_*)$ to the updating formula (10.11). Here λ is a parameter similar to α and β, and $\boldsymbol{\epsilon}_i$ is a vector of random numbers. These could form important topics for further research.

10.7 SPRING DESIGN

The design of a tension and compression spring is a well-known benchmark optimization problem. The main aim is to minimize the weight subject to constraints on deflection, stress, surge frequency and geometry. It involves three design variables: the wire diameter x_1, coil diameter x_2 and number/length of the coil x_3. This problem can be summarized as

$$\text{minimize } f(\boldsymbol{x}) = x_1^2 x_2 (2 + x_3), \tag{10.14}$$

subject to the following constraints

$$g_1(\boldsymbol{x}) = 1 - \frac{x_2^3 x_3}{71785 x_1^4} \le 0,$$

$$g_2(\boldsymbol{x}) = \frac{4x_2^2 - x_1 x_2}{12566(x_1^3 x_2 - x_1^4)} + \frac{1}{5108 x_1^2} - 1 \le 0,$$

$$g_3(\boldsymbol{x}) = 1 - \frac{140.45 x_1}{x_2^2 x_3} \le 0,$$

$$g_4(\boldsymbol{x}) = \frac{x_1 + x_2}{1.5} - 1 \le 0. \tag{10.15}$$

The simple bounds on the design variables are

$$0.05 \le x_1 \le 2.0, \qquad 0.25 \le x_2 \le 1.3, \qquad 2.0 \le x_3 \le 15.0. \tag{10.16}$$

The best solution found in the literature (e.g., Cagnina et al. 2008) is

$$\boldsymbol{x}_* = (0.051690, 0.356750, 11.287126), \tag{10.17}$$

with the objective

$$f(\boldsymbol{x}_*) = 0.012665. \tag{10.18}$$

We now provide the Matlab implementation of our firefly algorithm together with the penalty method for incorporating constraints. You may need a newer version of Matlab to deal with function handles. If you run the program a few times, you can get the above optimal solutions. It is even possible to produce better results if you experiment the program for a while.

```
% ------------------------------------------------------%
% Firefly Algorithm for constrained optimization       %
% by Xin-She Yang (Cambridge University) Copyright @2009 %
% ------------------------------------------------------%
function fa_mincon_demo

% parameters [n N_iteration alpha betamin gamma]
para=[40 150 0.5 0.2 1];

% This demo uses the Firefly Algorithm to solve the
% [Spring Design Problem as described by Cagnina et al.,
% Informatica, vol. 32, 319-326 (2008). ]

% Simple bounds/limits
disp('Solve the simple spring design problem ...');
Lb=[0.05 0.25 2.0];
Ub=[2.0 1.3 15.0];

% Initial random guess
u0=(Lb+Ub)/2;

[u,fval,NumEval]=ffa_mincon(@cost,@constraint,u0,Lb,Ub,para);
```

```
% Display results
bestsolution=u
bestojb=fval
total_number_of_function_evaluations=NumEval

%%% Put your own cost/objective function here --------%%%
%% Cost or Objective function
 function z=cost(x)
z=(2+x(3))*x(1)^2*x(2);

% Constrained optimization using penalty methods
% by changing f to F=f+ \sum lam_j*g^2_j*H_j(g_j)
% where H(g)=0 if g<=0 (true), =1 if g is false

%%% Put your own constraints here ---------------------%%%
function [g,geq]=constraint(x)
% All nonlinear inequality constraints should be here
% If no inequality constraint at all, simple use g=[];
g(1)=1-x(2)^3*x(3)/(7178*x(1)^4);
tmpf=(4*x(2)^2-x(1)*x(2))/(12566*(x(2)*x(1)^3-x(1)^4));
g(2)=tmpf+1/(5108*x(1)^2)-1;
g(3)=1-140.45*x(1)/(x(2)^2*x(3));
g(4)=x(1)+x(2)-1.5;

% all nonlinear equality constraints should be here
% If no equality constraint at all, put geq=[] as follows
geq=[];

%%% End of the part to be modified -------------------%%%

%%% ------------------------------------------------%%%
%%% Do not modify the following codes unless you want %%%
%%% to improve its performance etc                  %%%
% -----------------------------------------------------
% ===Start of the Firefly Algorithm Implementation ======
% Inputs: fhandle => @cost (your own cost function,
%                   can be an external file  )
%     nonhandle => @constraint, all nonlinear constraints
%                   can be an external file or a function
%         Lb = lower bounds/limits
%         Ub = upper bounds/limits
%   para == optional (to control the Firefly algorithm)
% Outputs: nbest   = the best solution found so far
%          fbest   = the best objective value
%       NumEval = number of evaluations: n*MaxGeneration
% Optional:
% The alpha can be reduced (as to reduce the randomness)
% -----------------------------------------------------
```

```
% Start FA
function [nbest,fbest,NumEval]...
           =ffa_mincon(fhandle,nonhandle,u0, Lb, Ub, para)
% Check input parameters (otherwise set as default values)
if nargin<6, para=[20 50 0.25 0.20 1]; end
if nargin<5, Ub=[]; end
if nargin<4, Lb=[]; end
if nargin<3,
disp('Usuage: FA_mincon(@cost, @constraint,u0,Lb,Ub,para)');
end

% n=number of fireflies
% MaxGeneration=number of pseudo time steps
% ------------------------------------------------
% alpha=0.25;      % Randomness 0--1 (highly random)
% betamn=0.20;     % minimum value of beta
% gamma=1;         % Absorption coefficient
% ------------------------------------------------
n=para(1);  MaxGeneration=para(2);
alpha=para(3); betamin=para(4); gamma=para(5);

% Total number of function evaluations
NumEval=n*MaxGeneration;

% Check if the upper bound & lower bound are the same size
if length(Lb) ~=length(Ub),
    disp('Simple bounds/limits are improper!');
    return
end

% Calcualte dimension
d=length(u0);

% Initial values of an array
zn=ones(n,1)*10^100;
% ------------------------------------------------
% generating the initial locations of n fireflies
[ns,Lightn]=init_ffa(n,d,Lb,Ub,u0);

% Iterations or pseudo time marching
for k=1:MaxGeneration,     %%%%% start iterations

% This line of reducing alpha is optional
alpha=alpha_new(alpha,MaxGeneration);

% Evaluate new solutions (for all n fireflies)
for i=1:n,
```

```
   zn(i)=Fun(fhandle,nonhandle,ns(i,:));
   Lightn(i)=zn(i);
end

% Ranking fireflies by their light intensity/objectives
[Lightn,Index]=sort(zn);
ns_tmp=ns;
for i=1:n,
 ns(i,:)=ns_tmp(Index(i),:);
end

%% Find the current best
nso=ns; Lighto=Lightn;
nbest=ns(1,:); Lightbest=Lightn(1);

% For output only
fbest=Lightbest;

% Move all fireflies to the better locations
[ns]=ffa_move(n,d,ns,Lightn,nso,Lighto,nbest,...
      Lightbest,alpha,betamin,gamma,Lb,Ub);

end   %%%%% end of iterations

% ----------------------------------------------------------
% ----- All the subfunctions are listed here ------------
% The initial locations of n fireflies
function [ns,Lightn]=init_ffa(n,d,Lb,Ub,u0)
  % if there are bounds/limits,
if length(Lb)>0,
   for i=1:n,
   ns(i,:)=Lb+(Ub-Lb).*rand(1,d);
   end
else
   % generate solutions around the random guess
   for i=1:n,
   ns(i,:)=u0+randn(1,d);
   end
end

% initial value before function evaluations
Lightn=ones(n,1)*10^100;

% Move all fireflies toward brighter ones
function [ns]=ffa_move(n,d,ns,Lightn,nso,Lighto,...
                nbest,Lightbest,alpha,betamin,gamma,Lb,Ub)
% Scaling of the system
scale=abs(Ub-Lb);
```

```
% Updating fireflies
for i=1:n,
% The attractiveness parameter beta=exp(-gamma*r)
   for j=1:n,
      r=sqrt(sum((ns(i,:)-ns(j,:)).^2));
      % Update moves
if Lightn(i)>Lighto(j), % Brighter and more attractive
   beta0=1; beta=(beta0-betamin)*exp(-gamma*r.^2)+betamin;
   tmf=alpha.*(rand(1,d)-0.5).*scale;
   ns(i,:)=ns(i,:).*(1-beta)+nso(j,:).*beta+tmpf;
      end
   end % end for j

end % end for i

% Check if the updated solutions/locations are within limits
[ns]=findlimits(n,ns,Lb,Ub);

% This function is optional, as it is not in the original FA
% The idea to reduce randomness is to increase the convergence,
% however, if you reduce randomness too quickly, then premature
% convergence can occur. So use with care.
function alpha=alpha_new(alpha,NGen)
% alpha_n=alpha_0(1-delta)^NGen=0.005
% alpha_0=0.9
delta=1-(0.005/0.9)^(1/NGen);
alpha=(1-delta)*alpha;

% Make sure the fireflies are within the bounds/limits
function [ns]=findlimits(n,ns,Lb,Ub)
for i=1:n,
     % Apply the lower bound
  ns_tmp=ns(i,:);
  I=ns_tmp<Lb;
  ns_tmp(I)=Lb(I);

  % Apply the upper bounds
  J=ns_tmp>Ub;
  ns_tmp(J)=Ub(J);
  % Update this new move
  ns(i,:)=ns_tmp;
end

%---------------------------------------
% d-dimensional objective function
function z=Fun(fhandle,nonhandle,u)
% Objective
```

```
z=fhandle(u);

% Apply nonlinear constraints by the penalty method
% Z=f+sum_k=1^N lam_k g_k^2 *H(g_k) where lam_k >> 1
z=z+getnonlinear(nonhandle,u);

function Z=getnonlinear(nonhandle,u)
Z=0;
% Penalty constant >> 1
lam=10^15; lameq=10^15;
% Get nonlinear constraints
[g,geq]=nonhandle(u);

% Apply inequality constraints as a penalty function
for k=1:length(g),
    Z=Z+ lam*g(k)^2*getH(g(k));
end
% Apply equality constraints (when geq=[], length->0)
for k=1:length(geq),
   Z=Z+lameq*geq(k)^2*geteqH(geq(k));
end

% Test if inequalities hold
% H(g) which is something like an index function
function H=getH(g)
if g<=0,
    H=0;
else
    H=1;
end

% Test if equalities hold
function H=geteqH(g)
if g==0,
    H=0;
else
    H=1;
end
%% ==== End of Firefly Algorithm implementation ======
```

REFERENCES

1. J. Arora, *Introduction to Optimum Design*, McGraw-Hill, (1989).
2. L. C. Cagnina, S. C. Esquivel, C. A. Coello, Solving engineering optimization problems with the simple constrained particle swarm optimizer, *Informatica*, **32**, 319-326 (2008).

3. S. Lukasik and S. Zak, Firefly algorithm for continuous constrained optimization tasks, ICCCI 2009, Lecture Notes in Artificial Intelligence (Eds. N. T. Ngugen et al.), **5796**, 97-106 (2009).

4. S. M. Lewis and C. K. Cratsley, Flash signal evolution, mate choice, and predation in fireflies, *Annual Review of Entomology*, **53**, 293-321 (2008).

5. C. O'Toole, *Firefly Encyclopedia of Insects and Spiders*, Firefly Books Ltd, 2002.

6. A. M. Reynolds and C. J. Rhodes, The Lévy flight paradigm: random search patterns and mechanisms, *Ecology*, **90**, 877-87 (2009).

7. E. G. Talbi, *Metaheuristics: From Design to Implementation*, Wiley, (2009).

8. X. S. Yang, *Nature-Inspired Metaheuristic Algorithms*, Luniver Press, (2008).

9. X. S. Yang, Firefly algorithms for multimodal optimization, in: *Stochastic Algorithms: Foundations and Applications*, SAGA 2009, Lecture Notes in Computer Science, **5792**, 169-178 (2009).

10. X. S. Yang, Firefly algorithm, Lévy flights and global optimization, in: *Research and Development in Intelligent Systems XXVI*, (Eds M. Bramer et al.), Springer, London, pp. 209-218 (2010).

Chapter 12

CUCKOO SEARCH

Cuckoo search (CS) is one of the latest nature-inspired metaheuristic algorithms, developed in 2009 by Xin-She Yang of Cambridge University and Suash Deb of C. V. Raman College of Engineering. CS is based on the brood parasitism of some cuckoo species. In addition, this algorithm is enhanced by the so-called Lévy flights, rather than by simple isotropic random walks. Recent studies show that CS is potentially far more efficient than PSO and genetic algorithms.

12.1 CUCKOO BREEDING BEHAVIOUR

Cuckoo are fascinating birds, not only because of the beautiful sounds they can make, but also because of their aggressive reproduction strategy. Some species such as the *ani* and *Guira* cuckoos lay their eggs in communal nests, though they may remove others' eggs to increase the hatching probability of their own eggs. Quite a number of species engage the obligate brood parasitism by laying their eggs in the nests of other host birds (often other species).

There are three basic types of brood parasitism: intraspecific brood parasitism, cooperative breeding, and nest takeover. Some host birds can engage direct conflict with the intruding cuckoos. If a host bird discovers the eggs are not their owns, they will either get rid of these alien eggs or simply abandon its nest and build a new nest elsewhere. Some cuckoo species such as the New World brood-parasitic *Tapera* have evolved in such a way that female parasitic cuckoos are often very specialized in the mimicry in colour and pattern of the eggs of a few chosen host species. This reduces the probability of their eggs being abandoned and thus increases their reproductivity.

In addition, the timing of egg-laying of some species is also amazing. Parasitic cuckoos often choose a nest where the host bird just laid its own eggs. In general, the cuckoo eggs hatch slightly earlier than their host eggs. Once the first cuckoo chick is hatched, the first instinct action it will take is to evict the host eggs by blindly propelling the eggs out of the nest, which increases the cuckoo chick's share of food provided by its host bird. Studies also show that a cuckoo chick can also mimic the call of host chicks to gain access to more feeding opportunity.

12.2 LÉVY FLIGHTS

On the other hand, various studies have shown that flight behaviour of many animals and insects has demonstrated the typical characteristics of Lévy flights. A recent study by Reynolds and Frye shows that fruit flies or *Drosophila melanogaster*, explore their landscape using a series of straight flight paths punctuated by a sudden 90^o turn, leading to a Lévy-flight-style intermittent scale free search pattern. Studies on human behaviour such as the Ju/'hoansi hunter-gatherer foraging patterns also show the typical feature of Lévy flights. Even light can be related to Lévy flights. Subsequently, such behaviour has been applied to optimization and optimal search, and preliminary results show its promising capability.

12.3 CUCKOO SEARCH

For simplicity in describing our new Cuckoo Search, we now use the following three idealized rules:

- Each cuckoo lays one egg at a time, and dumps its egg in a randomly chosen nest;
- The best nests with high-quality eggs will be carried over to the next generations;
- The number of available host nests is fixed, and the egg laid by a cuckoo is discovered by the host bird with a probability $p_a \in [0, 1]$. In this case, the host bird can either get rid of the egg, or simply abandon the nest and build a completely new nest.

As a further approximation, this last assumption can be approximated by a fraction p_a of the n host nests are replaced by new nests (with new random solutions).

For a maximization problem, the quality or fitness of a solution can simply be proportional to the value of the objective function. Other forms of fitness can be defined in a similar way to the fitness function in genetic algorithms.

For the implementation point of view, we can use the following simple representations that each egg in a nest represents a solution, and each cuckoo can lay only one egg (thus representing one solution), the aim is to use the new and potentially better solutions (cuckoos) to replace a not-so-good solution in the nests. Obviously, this algorithm can be extended to the more complicated case where each nest has multiple eggs representing a set of solutions. For this present work, we will use the simplest approach where each nest has only a single egg. In this case, there is no distinction between egg, nest or cuckoo, as each nest corresponds to one egg which also represents one cuckoo.

Based on these three rules, the basic steps of the Cuckoo Search (CS) can be summarized as the pseudo code shown in Fig. 12.1.

When generating new solutions $\boldsymbol{x}^{(t+1)}$ for, say, a cuckoo i, a Lévy flight is performed

$$\boldsymbol{x}_i^{(t+1)} = \boldsymbol{x}_i^{(t)} + \alpha \oplus \text{Lévy}(\lambda), \tag{12.1}$$

where $\alpha > 0$ is the step size which should be related to the scales of the problem of interests. In most cases, we can use $\alpha = O(L/10)$ where L is the characteristic

Cuckoo Search via Lévy Flights

Objective function $f(\boldsymbol{x})$, $\boldsymbol{x} = (x_1, ..., x_d)^T$
Generate initial population of n *host nests* $\boldsymbol{x}_i$
while *(*$t <$*MaxGeneration) or (stop criterion)*
  Get a cuckoo randomly/generate a solution by Lévy flights
    and then evaluate its quality/fitness F_i
  Choose a nest among n *(say,* j*) randomly*
  if $(F_i > F_j)$,
    Replace j *by the new solution*
  end
  A fraction (p_a) *of worse nests are abandoned*
    and new ones/solutions are built/generated
  Keep best solutions (or nests with quality solutions)
  Rank the solutions and find the current best
end while
Postprocess results and visualization

Figure 12.1: Pseudo code of the Cuckoo Search (CS).

scale of the problem of interest. The above equation is essentially the stochastic equation for a random walk. In general, a random walk is a Markov chain whose next status/location only depends on the current location (the first term in the above equation) and the transition probability (the second term). The product $\oplus$ means entrywise multiplications. This entrywise product is similar to those used in PSO, but here the random walk via Lévy flight is more efficient in exploring the search space, as its step length is much longer in the long run.

The Lévy flight essentially provides a random walk whose random step length is drawn from a Lévy distribution

$$\text{Lévy} \sim u = t^{-\lambda}, \qquad (1 < \lambda \leq 3), \tag{12.2}$$

which has an infinite variance with an infinite mean. Here the steps essentially form a random walk process with a power-law step-length distribution with a heavy tail. Some of the new solutions should be generated by Lévy walk around the best solution obtained so far, this will speed up the local search. However, a substantial fraction of the new solutions should be generated by far field randomization and whose locations should be far enough from the current best solution, this will make sure that the system will not be trapped in a local optimum.

From a quick look, it seems that there is some similarity between CS and hill-climbing in combination with some large scale randomization. But there are some significant differences. Firstly, CS is a population-based algorithm, in a way similar to GA and PSO, but it uses some sort of elitism and/or selection similar to that used in harmony search. Secondly, the randomization in CS is more efficient as the step length is heavy-tailed, and any large step is possible. Thirdly, the number of parameters in CS to be tuned is fewer than GA and PSO,

and thus it is potentially more generic to adapt to a wider class of optimization problems. In addition, each nest can represent a set of solutions, CS can thus be extended to the type of meta-population algorithms.

12.4 CHOICE OF PARAMETERS

After implementation, we have to validate the algorithm using test functions with analytical or known solutions. For example, one of the many test functions we have used is the bivariate Michalewicz function

$$f(x,y) = -\sin(x)\sin^{2m}(\frac{x^2}{\pi}) - \sin(y)\sin^{2m}(\frac{2y^2}{\pi}), \tag{12.3}$$

where $m = 10$ and $(x, y) \in [0, 5] \times [0, 5]$. This function has a global minimum $f_* \approx -1.8013$ at $(2.20319, 1.57049)$. This global optimum can easily be found using Cuckoo Search, and the results are shown in Fig. 12.2 where the final locations of the nests are also marked with $\diamond$ in the figure. Here we have used $n = 15$ nests, $\alpha = 1$ and $p_a = 0.25$. In most of our simulations, we have used $n = 15$ to 50.

From the figure, we can see that, as the optimum is approaching, most nests aggregate towards the global optimum. We also notice that the nests are also distributed at different (local) optima in the case of multimodal functions. This means that CS can find all the optima simultaneously if the number of nests are much higher than the number of local optima. This advantage may become more significant when dealing with multimodal and multiobjective optimization problems.

We have also tried to vary the number of host nests (or the population size n) and the probability p_a. We have used $n = 5, 10, 15$, 20, 30, $40, 50$, 100, 150, $250, 500$ and $p_a = 0$, 0.01, $0.05, 0.1$, $0.15, 0.2$, 0.25, $0.3, 0.4, 0.5$. From our simulations, we found that $n = 15$ to 40 and $p_a = 0.25$ are sufficient for most optimization problems. Results and analysis also imply that the convergence rate, to some extent, is not sensitive to the parameters used. This means that the fine adjustment is not needed for any given problems.

12.5 IMPLEMENTATION

```
% ---------------------------------------------------------
% Cuckoo algorithm by Xin-She Yang and Suasg Deb          %
% Programmed by Xin-She Yang at Cambridge University      %
% ---------------------------------------------------------
function [bestsol,fval]=cuckoo_search(Ngen)
% Here Ngen is the max number of function evaluations
if nargin<1, Ngen=1500; end

% d-dimensions (any dimension)
d=2;
% Number of nests (or different solutions)
```

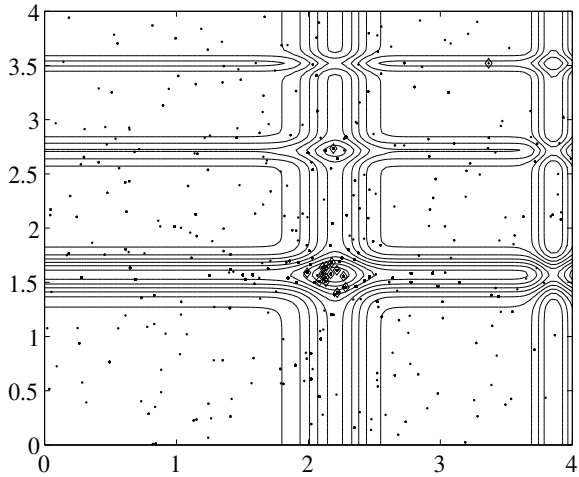

Figure 12.2: Search paths of nests using Cuckoo Search. The final locations of the nests are marked with ◇ in the figure.

```
n=25;

% Discovery rate of alien eggs/solutions
pa=0.25;

% Random initial solutions
nest=randn(n,d);
fbest=ones(n,1)*10^(100);   % minimization problems
Kbest=1;

for j=1:Ngen,
    % Find the current best
    Kbest=get_best_nest(fbest);
    % Choose a random nest (avoid the current best)
    k=choose_a_nest(n,Kbest);
    bestnest=nest(Kbest,:)
    % Generate a new solution (but keep the current best)
    s=get_a_cuckoo(nest(k,:),bestnest);

    % Evaluate this solution
    fnew=fobj(s);
    if fnew<=fbest(k),
        fbest(k)=fnew;
        nest(k,:)=s;
    end
    % discovery and randomization
    if rand<pa,
```

```
        k=get_max_nest(fbest);
        s=emptyit(nest(k,:));
        nest(k,:)=s;
        fbest(k)=fobj(s);
    end
end

%% Post-optimization processing
%% Find the best and display
[fval,I]=min(fbest)
bestsol=nest(I,:)

%% Display all the nests
nest

%% --------- All subfunctions are listed below -----------
%% Choose a nest randomly
function k=choose_a_nest(n,Kbest)
k=floor(rand*n)+1;
% Avoid the best
if k==Kbest,
 k=mod(k+1,n)+1;
end

%% Get a cuckoo and generate new solutions by ramdom walk
function s=get_a_cuckoo(s,star)
% This is a random walk, which is less efficient
% than Levy flights. In addition, the step size
% should be a vector for problems with different scales.
% Here is the simplified implementation for demo only!
stepsize=0.05;
s=star+stepsize*randn(size(s));

%% Find the worse nest
function k=get_max_nest(fbest)
[fmax,k]=max(fbest);

%% Find the current best nest
function k=get_best_nest(fbest)
[fmin,k]=min(fbest);

%% Replace some (of the worst nests)
%% by constructing new solutions/nests
function s=emptyit(s)
% Again the step size should be varied
% Here is a simplified approach
s=s+0.05*randn(size(s));
```

```
% d-dimensional objective function
function z=fobj(u)
% Rosenbrock's function (in 2D)
% It has an optimal solution at (1.000,1.000)
z=(1-u(1))^2+100*(u(2)-u(1)^2)^2;
```

If we run this program using some standard test functions, we can observe that CS outperforms many existing algorithms such as GA and PSO. The primary reasons are: 1) a fine balance of randomization and intensification, and 2) fewer number of control parameters. As for any metaheuristic algorithms, a good balance of intensive local search and an efficient exploration of the whole search space will usually lead to a more efficient algorithm. On the other hand, there are only two parameters in this algorithm, the population size n, and p_a. Once n is fixed, p_a essentially controls the elitism and the balance of randomization and local search. Few parameters make an algorithm less complex and thus potentially more generic. Such observations deserve more systematic research and further elaboration in the future work.

It is worth pointing out that there are three ways to carry out randomization: uniform randomization, random walks and heavy-tailed walks. The simplest way is to use a uniform distribution so that new solutions are limited between upper and lower bounds. Random walks can be used for global randomization or local randomization, depending on the step size used in the implementation. Lévy flights are heavy-tailed, which is most suitable for the randomization on the global scale.

As an example for solving constrained optimization, we now solved the spring design problem discussed in the chapter on firefly algorithm. The Matlab code is given below

```
% Cuckoo Search for nonlinear constrained optimization
% Programmed by Xin-She Yang @ Cambridge University 2009
function [bestsol,fval]=cuckoo_spring(N_iter)
format long;
% number of iterations
if nargin<1, N_iter=15000; end
% Number of nests
n=25;
disp('Searching ... may take a minute or so ...');
% d variables and simple bounds
% Lower and upper bounds
Lb=[0.05 0.25 2.0];
Ub=[2.0  1.3  15.0];
% Number of variables
d=length(Lb);

% Discovery rate
pa=0.25;
% Random initial solutions
nest=init_cuckoo(n,d,Lb,Ub);
fbest=ones(n,1)*10^(10);   % minimization problems
```

```
Kbest=1;

% Start of the cuckoo search
for j=1:N_iter,
    % Find the best nest
    [fmin,Kbest]=get_best_nest(fbest);
    % Choose a nest randomly
    k=choose_a_nest(n,Kbest);
    bestnest=nest(Kbest,:) ;
    % Get a cuckoo with a new solution
    s=get_a_cuckoo(nest(k,:),bestnest,Lb,Ub);

    % Update if the solution improves
    fnew=fobj(s);
    if fnew<=fbest(k),
        fbest(k)=fnew;
        nest(k,:)=s;
    end

    % Discovery and randomization
    if rand<pa,
    k=get_max_nest(fbest);
    s=emptyit(nest(k,:),Lb,Ub);
    nest(k,:)=s;
    fbest(k)=fobj(s);
    end
end

%% Find the best
[fmin,I]=min(fbest)
bestsol=nest(I,:);

% Show all the nests
nest
% Display the best solution
bestsol, fmin

% Initial locations of all n cuckoos
function [guess]=init_cuckoo(n,d,Lb,Ub)
for i=1:n,
    guess(i,1:d)=Lb+rand(1,d).*(Ub-Lb);
end

%% Choose a nest randomly
function k=choose_a_nest(n,Kbest)
k=floor(rand*n)+1;
% Avoid the best
if k==Kbest,
```

```
 k=mod(k+1,n)+1;
end

%% Get a cuckoo with a new solution via a random walk
%% Note: Levy flights were not implemented in this demo
function s=get_a_cuckoo(s,star,Lb,Ub)
s=star+0.01*(Ub-Lb).*randn(size(s));
s=bounds(s,Lb,Ub);

%% Find the worse nest
function k=get_max_nest(fbest)
[fmax,k]=max(fbest);

%% Find the best nest
function [fmin,k]=get_best_nest(fbest)
[fmin,k]=min(fbest);

%% Replace an abandoned nest by constructing a new nest
function s=emptyit(s,Lb,Ub)
s=s+0.01*(Ub-Lb).*randn(size(s));
s=bounds(s,Lb,Ub);

% Check if bounds are met
function ns=bounds(ns,Lb,Ub)
% Apply the lower bound
  ns_tmp=ns;
  I=ns_tmp<Lb;
  ns_tmp(I)=Lb(I);
  % Apply the upper bounds
  J=ns_tmp>Ub;
  ns_tmp(J)=Ub(J);
% Update this new move
  ns=ns_tmp;

% d-dimensional objective function
function z=fobj(u)
% The well-known spring design problem
z=(2+u(3))*u(1)^2*u(2);
z=z+getnonlinear(u);

function Z=getnonlinear(u)
Z=0;
% Penalty constant
lam=10^15;

% Inequality constraints
g(1)=1-u(2)^3*u(3)/(71785*u(1)^4);
gtmp=(4*u(2)^2-u(1)*u(2))/(12566*(u(2)*u(1)^3-u(1)^4));
```

```
g(2)=gtmp+1/(5108*u(1)^2)-1;
g(3)=1-140.45*u(1)/(u(2)^2*u(3));
g(4)=(u(1)+u(2))/1.5-1;

% No equality constraint in this problem, so empty;
geq=[];

% Apply inequality constraints
for k=1:length(g),
    Z=Z+ lam*g(k)^2*getH(g(k));
end
% Apply equality constraints
for k=1:length(geq),
   Z=Z+lam*geq(k)^2*getHeq(geq(k));
end

% Test if inequalities hold
% Index function H(g) for inequalities
function H=getH(g)
if g<=0,
    H=0;
else
    H=1;
end
% Index function for equalities
function H=getHeq(geq)
if geq==0,
   H=0;
else
   H=1;
end
% ----------------- end ------------------------------
```

This potentially powerful optimization algorithm can easily be extended to study multiobjective optimization applications with various constraints, even to NP-hard problems. Further studies can focus on the sensitivity and parameter studies and their possible relationships with the convergence rate of the algorithm. Hybridization with other popular algorithms such as PSO and differential evolution will also be potentially fruitful.

REFERENCES

1. Barthelemy P., Bertolotti J., Wiersma D. S., A Lévy flight for light, *Nature*, **453**, 495-498 (2008).

2. Bradley D., Novel 'cuckoo search algorithm' beats particle swarm optimization in engineering design (news article), *Science Daily*, May 29, (2010). Also in *Scientific Computing* (magazine), 1 June 2010.
3. Brown C., Liebovitch L. S., Glendon R., Lévy flights in Dobe Ju/'hoansi foraging patterns, *Human Ecol.*, **35**, 129-138 (2007).
4. Chattopadhyay R., A study of test functions for optimization algorithms, *J. Opt. Theory Appl.*, **8**, 231-236 (1971).
5. Passino K. M., *Biomimicry of Bacterial Foraging for Distributed Optimization*, University Press, Princeton, New Jersey (2001).
6. Payne R. B., Sorenson M. D., and Klitz K., *The Cuckoos*, Oxford University Press, (2005).
7. Pavlyukevich I., Lévy flights, non-local search and simulated annealing, *J. Computational Physics*, **226**, 1830-1844 (2007).
8. Pavlyukevich I., Cooling down Lévy flights, *J. Phys. A:Math. Theor.*, **40**, 12299-12313 (2007).
9. Reynolds A. M. and Frye M. A., Free-flight odor tracking in Drosophila is consistent with an optimal intermittent scale-free search, *PLoS One*, **2**, e354 (2007).
10. A. M. Reynolds and C. J. Rhodes, The Lévy flight paradigm: random search patterns and mechanisms, *Ecology*, **90**, 877-87 (2009).
11. Schoen F., A wide class of test functions for global optimization, *J. Global Optimization*, **3**, 133-137, (1993).
12. Shlesinger M. F., Search research, *Nature*, **443**, 281-282 (2006).
13. Yang X. S. and Deb S., Cuckoo search via Lévy flights, in: *Proc. of World Congress on Nature & Biologically Inspired Computing* (NaBic 2009), IEEE Publications, USA, pp. 210-214 (2009).
14. Yang X. S. and Deb S,, Engineering optimization by cuckoo search, *Int. J. Math. Modelling & Numerical Optimisation*, **1**, 330-343 (2010).