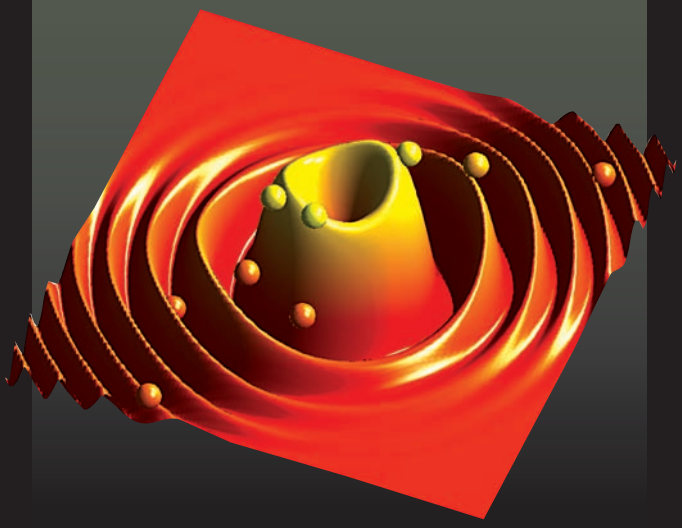
Introduction to Mathematical Optimization

From Linear Programming to Metaheuristics



Xin-She Yang

CISP
Cambridge International Science Publishing

Introduction to Mathematical Optimization

– From Linear Programming to Metaheuristics

Introduction to Mathematical Optimization

– From Linear Programming to Metaheuristics

Xin-She Yang

University of Cambridge, United Kingdom

Cambridge International Science Publishing

Published by
Cambridge International Science Publishing
7 Meadow Walk, Great Abington, Cambridge CB21 6AZ, UK
http://www.cisp-publishing.com

First Published 2008

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

ISBN 978-1-904602-82-8

Cover design by Terry Callanan

Printed and bound in the UK by Lightning Source UK Ltd

Contents

Preface

Optimization is everywhere from routine business transactions to important decisions of any sort, from engineering design to industrial manufacturing, and from choosing a career path to planning our holidays. In all these activities, there are always some things (objectives) we are trying to optimize and these objectives could be cost, profit, performance, quality, enjoyment, customer-rating and others. The formal approach to these optimization problems forms the major part of the mathematical optimization or mathematical programming.

The topics of mathematical optimization are broad and the related literature is vast. It is often a daunting task for beginners to find a right book and to learn the right (and useful) algorithms widely used in mathematical programming. Even for lecturers and educators, it is not trivial to decide what algorithms to teach and to provide a balanced coverage of a wide range of topics because there are so many algorithms to choose from. From my own learning experience, I understand that some algorithms took substantial amount of time and effort in programming and it was frustrating to realise that it did not work well for optimization problems at hand in the end. After some frustrations, then I realized other algorithms worked much efficiently for a given problem. The initial cause was often that the advantages and disadvantages were not explicitly explained in the literature or I was too eager to do some optimization simulations without realizing certain pitfalls of the related algorithms. Such learning experience is valuable to me in writing this book so that we can endeavour to provide a

balanced view of various algorithms and to provide a right coverage of useful and yet efficient algorithms selected from a wide range of optimization techniques.

Therefore, this book strives to provide a balanced coverage of efficient algorithms commonly used in solving mathematical optimization problems. It covers both the convectional algorithms and modern heuristic and metaheuristic methods. Topics include gradient-based algorithms (such as the Newton-Raphson method and steepest descent method), Hooke-Jeeves pattern search, Lagrange multipliers, linear programming, particle swarm optimization (PSO), simulated annealing (SA), and Tabu search. We also briefly introduce the multiobjective optimization including important concepts such as Pareto optimality and utility method, and provide three Matlab and Octave programs so as to demonstrate how PSO and SA work. In addition, we will use an example to demonstrate how to modify these programs to solve multiobjective optimization problems using the recursive method.

I would like to thank many of my mentors, friends and colleagues: Drs A. C. Fowler and S. Tsou at Oxford University, Drs J. M. Lees, T. Love, C. Morley, and G. T. Parks at Cambridge University. Special thanks to Dr G. T. Parks who introduced me to the wonderful technique of Tabu Search.

I also thank my publisher, Dr Victor Riecansky, at Cambridge International Science Publishing, for his help and professionalism. Last but not least, I thank my wife, Helen, and son, Young, for their help.

Xin-She Yang

Cambridge, 2008

Part I

Fundamentals

Chapter 1

Mathematical Optimization

Optimization is everywhere, from business to engineering design, from planning your holiday to your daily routine. Business organizations have to maximize their profit and minimize the cost. Engineering design has to maximize the performance of the designed product while of course minimizing the cost at the same time. Even when we plan holidays we want to maximize the enjoyment and minimize the cost. Therefore, the studies of optimization are of both scientific interest and practical implications and subsequently the methodology will have many applications.

1.1 Optimization

Whatever the real world problem is, it is usually possible to formulate the optimization problem in a generic form. All optimization problems with explicit objectives can in general be expressed as nonlinearly constrained optimization problems in the following generic form

$$\underset{\boldsymbol{x}\in\Re^n}{\text{maximize/minimize}} f(\boldsymbol{x}), \quad \boldsymbol{x} = (x_1, x_2, ..., x_n)^T \in \Re^n,$$

$$\text{subject to } \phi_j(\boldsymbol{x}) = 0, \quad (j = 1, 2, ..., M),$$

$$\psi_k(\boldsymbol{x}) \geq 0, \quad (k = 1, ..., N), \tag{1.1}$$

where $f(\boldsymbol{x})$, $\phi_i(\boldsymbol{x})$ and $\psi_j(\boldsymbol{x})$ are scalar functions of the real column vector $\boldsymbol{x}$. Here the components x_i of $\boldsymbol{x} = (x_1, ..., x_n)^T$ are called design variables or more often decision variables, and they can be either continuous, or discrete or mixed of these two. The vector $\boldsymbol{x}$ is often called a decision vector which varies in a n-dimensional space $\Re^n$. The function $f(\boldsymbol{x})$ is called the objective function or cost function. In addition, $\phi_i(\boldsymbol{x})$ are constraints in terms of M equalities, and $\psi_j(\boldsymbol{x})$ are constraints written as N inequalities. So there are $M + N$ constraints in total. The optimization problem formulated here is a nonlinear constrained problem.

The space spanned by the decision variables is called the search space $\Re^n$, while the space formed by the objective function values is called the solution space. The optimization problem essentially maps the $\Re^n$ domain or space of decision variables into a solution space $\Re$ (or the real axis in general).

The objective function $f(\boldsymbol{x})$ can be either linear or nonlinear. If the constraints ϕ_i and ψ_j are all linear, it becomes a linearly constrained problem. Furthermore, ϕ_i, ψ_j and the objective function $f(\boldsymbol{x})$ are all linear, then it becomes a linear programming problem. If the objective is at most quadratic with linear constraints, then it is called quadratic programming. If all the values of the decision variables can be integers, then this type of linear programming is called integer programming or integer linear programming.

Linear programming is very important in applications and has been well-studied, while there is still no generic method for solving nonlinear programming in general, though some important progress has been made in the last few decades. It is worth pointing out that the term *programming* here means *planning*, it has nothing to do with computer programming and the wording coincidence is purely incidental.

On the other hand, if no constraints are specified so that x_i can take any values in the real axis (or any integers), the optimization problem is referred to as the unconstrained optimization problem.

The simplest optimization without any constraints is probably the search of the maxima or minima of a function. For example, finding the maximum of an univariate function $f(x)$

$$f(x) = xe^{-x^2}, \qquad -\infty < x < \infty, \tag{1.2}$$

is a simple unconstrained problem. While the following problem is a simple constrained minimization problem

$$f(x_1, x_2) = x_1^2 + x_1 x_2 + x_2^2, \qquad (x_1, x_2) \in \Re^2, \tag{1.3}$$

subject to

$$x_1 \geq 1, \qquad x_2 - 2 = 0. \tag{1.4}$$

Example 1.1: To find the minimum of $f(x) = x^2 e^{-x^2}$, we have the stationary condition $f'(x) = 0$ or

$$f'(x) = 2x \times e^{-x^2} + x^2 \times (-2x)e^{-x^2} = 2(x - x^3)e^{-x^2} = 0.$$

As $e^{-x^2} > 0$, we have

$$x(1 - x^2) = 0,$$

or

$$x = 0, \qquad x = \pm 1.$$

The second derivative

$$f''(x) = 2e^{-x^2}(1 - 5x^2 + 2x^4),$$

which is an even function with respect to x. So at $x = \pm 1$, $f''(\pm 1) = 2[1 - 5(\pm 1)^2 + 2(\pm 1)^4]e^{-(\pm 1)^2} = -4e^{-1} < 0$. Thus, the maximum of $f_{\max} = e^{-1}$ occur at $x_* = \pm 1$. At $x = 0$, we have $f''(0) = 2 > 0$, thus the minimum of $f(x)$ occur at $x_* = 0$ with $f_{\min}(0) = 0$.

It is worth pointing out that the objectives are explicitly known in all the optimization problems to be discussed in this book. However, in reality, it is often difficult to quantify what we want to achieve, but we still try to optimize certain things

such as the degree of enjoyment or the quality of service on holiday. In other cases, it might be impossible to write the objective function in an explicit mathematical form.

Whatever the objectives, we have to evaluate the objectives many times. In most cases, the evaluations of the objective functions consume a lot of computational power (which costs money) and design time. Any efficient algorithm that can reduce the number of objective evaluations will save both time and money. The algorithms presented in this book will still be applicable to the cases where the objectives are not known explicitly, though certain modifications are required to suit a particular application. The basic principle of these search algorithms remain the same.

1.2 Optimality Criteria

In mathematical programming, there are many important concepts that will be introduced in this book. Now we first introduce three related concepts: feasible solutions, the strong local maximum and the weak local maximum.

A point $\boldsymbol{x}$ which satisfies all the constraints is called a feasible point and thus is a feasible solution to the problem. The set of all feasible points is called the feasible region. A point $\boldsymbol{x}_*$ is called a strong local maximum of the nonlinearly constrained optimization problem if $f(\boldsymbol{x})$ is defined in a δ-neigbourhood $N(\boldsymbol{x}_*, \delta)$ and satisfies $f(\boldsymbol{x}_*) > f(\boldsymbol{u})$ for $\forall \boldsymbol{u} \in N(\boldsymbol{x}_*, \delta)$ where $\delta > 0$ and $\boldsymbol{u} \neq \boldsymbol{x}_*$. If $\boldsymbol{x}_*$ is not a strong local maximum, the inclusion of equality in the condition $f(\boldsymbol{x}_*) \geq f(\boldsymbol{u})$ for $\forall \boldsymbol{u} \in N(\boldsymbol{x}_*, \delta)$ defines the point $\boldsymbol{x}_*$ as a weak local maximum (see Fig. 1.1). The local minima can be defined in the similar manner when $>$ and $\geq$ are replaced by $<$ and $\leq$, respectively.

Figure 1.1 shows various local maxima and minima. Point A is a strong local maximum, while point B is a weak local maximum because there are many (well, infinite) different values of x which will lead to the same value of $f(x_*)$. Point D is a global maximum. However, point C is a strong local minimum, but it has a discontinuity in $f'(x_*)$. So the stationary condition

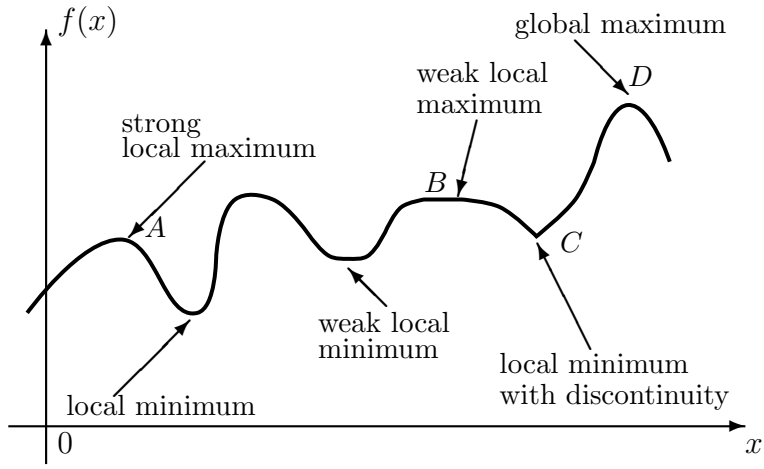


Figure 1.1: Strong and weak maxima and minima.

for this point $f'(x_*) = 0$ is not valid. We will not deal with this type of minima or maxima in detail. In our present discussion, we will assume that both $f(\boldsymbol{x})$ and $f'(\boldsymbol{x})$ are always continuous or $f(\boldsymbol{x})$ is everywhere twice-continuously differentiable.

Example 1.2: The minimum of $f(x) = x^2$ at $x = 0$ is a strong local minimum. The minimum of $g(x, y) = (x - y)^2 + (x - y)^2$ at $x = y = 0$ is a weak local minimum because $g(x, y) = 0$ along the line $x = y$ so that $g(x, y = x) = 0 = g(0, 0)$.

1.3 Computational Complexity

The efficiency of an algorithm is often measured by the algorithmic complexity or computational complexity. In literature, this complexity is also called Kolmogorov complexity. For a given problem size n, the complexity is denoted using Big-O notations such as $O(n^2)$ or $O(n \log n)$.

Loosely speaking, for two functions $f(x)$ and $g(x)$, if

$$\lim_{x \to x_0} \frac{f(x)}{g(x)} \to K, \tag{1.5}$$

where K is a finite, non-zero limit, we write

$$f = O(g). \tag{1.6}$$

The big O notation means that f is asymptotically equivalent to the order of $g(x)$. If the limit is unity or $K = 1$, we say $f(x)$ is order of $g(x)$. In this special case, we write

$$f \sim g, \tag{1.7}$$

which is equivalent to $f/g \to 1$ and $g/f \to 1$ as $x \to x_0$. Obviously, x_0 can be any value, including 0 and ∞. The notation $\sim$ does not necessarily mean $\approx$ in general, though they might give the same results, especially in the case when $x \to 0$ [for example, $\sin x \sim x$ and $\sin x \approx x$ if $x \to 0$].

When we say f is order of 100 (or $f \sim 100$), this does not mean $f \approx 100$, but it can mean that f is between about 50 to 150. The small o notation is often used if the limit tends to 0. That is

$$\lim_{x \to x_0} \frac{f}{g} \to 0, \tag{1.8}$$

or

$$f = o(g). \tag{1.9}$$

If $g > 0$, $f = o(g)$ is equivalent to $f \ll g$. For example, for $\forall x \in \mathcal{R}$, we have $e^x \approx 1 + x + O(x^2) \approx 1 + x + \frac{x^2}{2} + o(x)$.

Example 1.3: A classical example is Stirling's asymptotic series for factorials

$$n! \sim \sqrt{2\pi n}\,(\frac{n}{e})^n(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51480n^3} - ...). \tag{1.10}$$

This is a good example of asymptotic series. For standard power expansions, the error $R_k(h^k) \to 0$, but for an asymptotic series, the error of the truncated series R_k decreases and gets smaller compared with the leading term [here $\sqrt{2\pi n}(n/e)^n$]. However, R_n does not necessarily tend to zero. In fact,

$$R_2 = \frac{1}{12n} \cdot \sqrt{2\pi n}(n/e)^n,$$

is still very large as $R_2 \to \infty$ if $n \gg 1$. For example, for $n = 100$, we have $n! = 9.3326 \times 10^{157}$, while the leading approximation is $\sqrt{2\pi n}(n/e)^n = 9.3248 \times 10^{157}$. The difference between these two values is 7.7740×10^{154}, which is still very large, though three orders smaller than the leading approximation.

Let us come back to the computational complexity of an algorithm. For the sorting algorithm for a given number of n data entries, sorting these numbers into either ascending or descending order will take the computational time as a function of the problem size n. $O(n)$ means a linear complexity, while $O(n^2)$ has a quadratic complexity. That is, if n is doubled, then the time will double for linear complexity, but it will quadruple for quadratic complexity.

For example, the bubble sorting algorithm starts at the beginning of the data set by comparing the first two elements. If the first is smaller than the second, then swap them. This comparison and swap process continues for each possible pair of adjacent elements. There are $n \times n$ pairs as we need two loops over the whole data set, then the algorithm complexity is $O(n^2)$. On the other hand, the quicksort algorithm uses a divide-and-conquer approach via partition. By first choosing a pivot element, we put all the elements into two sublists with all the smaller elements before the pivot and all the greater elements after it. Then, the sublists are recursively sorted in a similar manner. This algorithm will result in a complexity of $O(n \log n)$. The quicksort is much more efficient than the bubble algorithm. For $n = 1000$, the bubble algorithm will need about $O(n^2) \approx O(10^6)$ calculations, while the quicksort only requires $O(n \log n) \approx O(3 \times 10^3)$ calculations (at least two orders less).

1.4 NP-Complete Problems

In mathematical programming, an easy or tractable problem is a problem whose solution can be obtained by computer algorithms with a solution time (or number of steps) as a polyno-

mial function of problem size n. Algorithms with polynomial-time are considered efficient. A problem is called the P-problem or polynomial-time problem if the number of steps needed to find the solution is bounded by a polynomial in n and it has at least one algorithm to solve it.

On the other hand, a hard or intractable problem requires solution time that is an exponential function of n, and thus exponential-time algorithms are considered inefficient. A problem is called nondeterministic polynomial (NP) if its solution can only be guessed and evaluated in polynomial time, and there is no known rule to make such guess (hence, nondeterministic). Consequently, guessed solutions cannot guarantee to be optimal or even near optimal. In fact, no known algorithms exist to solve NP-hard problems, and only approximate solutions or heuristic solutions are possible. Thus, heuristic and metaheuristic methods are very promising in obtaining approximate solutions or nearly optimal/suboptimal solutions.

A problem is called NP-complete if it is an NP-hard problem and all other problems in NP are reducible to it via certain reduction algorithms. The reduction algorithm has a polynomial time. An example of NP-hard problem is the Travelling Salesman Problem, and its objective is to find the shortest route or minimum travelling cost to visit all given n cities exactly once and then return to the starting city.

The solvability of NP-complete problems (whether by polynomial time or not) is still an unsolved problem which the Clay Mathematical Institute is offering a million dollars reward for a formal proof. Most real-world problems are NP-hard, and thus any advance in dealing with NP problems will have potential impact on many applications.

Chapter 2

Norms and Hessian Matrices

Before we proceed to study various optimization methods, let us first review some of the fundamental concepts such as norms and Hessian matrices that will be used frequently through this book.

2.1 Vector and Matrix Norms

For a vector $\boldsymbol{v}$, its p-norm is denoted by $\|\boldsymbol{v}\|_p$ and defined as

$$\|\boldsymbol{v}\|_p = (\sum_{i=1}^{n} |v_i|^p)^{1/p}, \tag{2.1}$$

where p is a positive integer. From this definition, it is straightforward to show that the p-norm satisfies the following conditions: $\|\boldsymbol{v}\| \geq 0$ for all $\boldsymbol{v}$, and $\|\boldsymbol{v}\| = 0$ if and only if $\boldsymbol{v} = \boldsymbol{0}$. This is the non-negativeness condition. In addition, for any real number α, we have the scaling condition: $\|\alpha\boldsymbol{v}\| = \alpha\|\boldsymbol{v}\|$.

Three most common norms are one-, two- and infinity-norms when $p = 1, 2$, and ∞, respectively. For $p = 1$, the one-norm is just the simple sum of each component $|v_i|$, while the two-norm $\|\boldsymbol{v}\|_2$ for $p = 2$ is the standard Euclidean norm

because $\|\boldsymbol{v}\|_2$ is the length of the vector $\boldsymbol{v}$

$$\|\boldsymbol{v}\|_2 = \sqrt{\boldsymbol{v}\cdot\boldsymbol{v}} = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}, \tag{2.2}$$

where $\boldsymbol{u}\cdot\boldsymbol{v}$ is the inner product of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$.

For the special case $p = \infty$, we denote $v_{\max}$ the maximum absolute value of all the components v_i, or $v_{\max} \equiv \max|v_i| = \max(|v_1|, |v_2|, ..., |v_n|)$.

$$\|\boldsymbol{v}\|_\infty = \lim_{p\to\infty}(\sum_{i=1}^{n}|v_i|^p)^{1/p} = \lim_{p\to\infty}\Big(v_{\max}^p\sum_{i=1}^{n}|\frac{v_i}{v_{\max}}|^p\Big)^{1/p}$$

$$= \lim_{p\to\infty}(v_{\max}^p)^{\frac{1}{p}}(\sum|\frac{v_i}{v_{\max}}|^p)^{\frac{1}{p}} = v_{\max}\lim_{p\to\infty}(\sum_{i=1}^{n}|\frac{v_i}{v_{\max}}|^p)^{\frac{1}{p}}. \tag{2.3}$$

Since $|v_i/v_{\max}| \le 1$ and for all terms $|v_i/v_{\max}| < 1$, we have $|v_i/v_{\max}|^p \to 0$ when $p \to \infty$. Thus, the only non-zero term in the sum is one when $|v_i/v_{\max}| = 1$, which means that

$$\lim_{p\to\infty}\sum_{i=1}^{n}|v_i/v_{\max}|^p = 1. \tag{2.4}$$

Therefore, we finally have

$$\|\boldsymbol{v}\|_\infty = v_{\max} = \max|v_i|. \tag{2.5}$$

For the uniqueness of norms, it is necessary for the norms to satisfy the triangle inequality

$$\|\boldsymbol{u}+\boldsymbol{v}\| \le \|\boldsymbol{u}\| + \|\boldsymbol{v}\|. \tag{2.6}$$

It is straightforward to check that for $p = 1, 2$, and ∞ from their definitions, they indeed satisfy the triangle inequality. The equality occurs when $\boldsymbol{u} = \boldsymbol{v}$. It leaves as an exercise to check this inequality is true for any $p > 0$.

Example 2.1: For the vector $\boldsymbol{u} = \begin{pmatrix}5 & 2 & 3 & -2\end{pmatrix}^T$ and $\boldsymbol{v} = \begin{pmatrix}-2 & 0 & 1 & 2\end{pmatrix}^T$, then the p-norms of $\boldsymbol{u}$ are

$$\|\boldsymbol{u}\|_1 = |5| + |2| + |3| + |-2| = 12,$$

$$\|\boldsymbol{u}\|_2 = \sqrt{5^2 + 2^2 + 3^2 + (-2)^2} = \sqrt{42},$$

and

$$\|\boldsymbol{u}\|_\infty = \max(5, 2, 3, -2) = 5.$$

Similarly, $\|\boldsymbol{v}\|_1 = 5$, $\|\boldsymbol{v}\|_2 = 3$ and $\|\boldsymbol{v}\|_\infty = 2$. We know that

$$\boldsymbol{u} + \boldsymbol{v} = \begin{pmatrix} 5 + -2 \\ 2 + 0 \\ 3 + 1 \\ -2 + 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 4 \\ 0 \end{pmatrix},$$

and its corresponding norms are $\|\boldsymbol{u} + \boldsymbol{v}\|_1 = 9$, $\|\boldsymbol{u} + \boldsymbol{v}\|_2 = \sqrt{29}$ and $\|\boldsymbol{u} + \boldsymbol{v}\|_\infty = 4$. It is straightforward to check that

$$\|\boldsymbol{u} + \boldsymbol{v}\|_1 = 9 < 12 + 5 = \|\boldsymbol{u}\|_1 + \|\boldsymbol{v}\|_1,$$

$$\|\boldsymbol{u} + \boldsymbol{v}\|_2 = \sqrt{29} < \sqrt{42} + 3 = \|\boldsymbol{u}\|_2 + \|\boldsymbol{v}\|_2,$$

and

$$\|\boldsymbol{u} + \boldsymbol{v}\|_\infty = 4 < 5 + 4 = \|\boldsymbol{u}\|_\infty + \|\boldsymbol{v}\|_\infty.$$

Matrices are the extension of vectors, so we can define the corresponding norms. For an $m \times n$ matrix $\boldsymbol{A} = [a_{ij}]$, a simple way to extend the norms to use the fact that $\boldsymbol{Au}$ is a vector for any vector $\|\boldsymbol{u}\| = 1$. So the p-norm is defined as

$$\|\boldsymbol{A}\|_p = (\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^p)^{1/p}. \tag{2.7}$$

Alternatively, we can consider that all the elements or entries a_{ij} form a vector. A popular norm, called Frobenius form (also called the Hilbert-Schmidt norm), is defined as

$$\|\boldsymbol{A}\|_F = \Big(\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2\Big)^{1/2}. \tag{2.8}$$

In fact, Frobenius norm is 2-norm.

Other popular norms are based on the absolute column sum or row sum. For example,

$$\|\boldsymbol{A}\|_1 = \max_{1\le j\le n}(\sum_{i=1}^{m}|a_{ij}|), \tag{2.9}$$

which is the maximum of the absolute column sum, while

$$\|\boldsymbol{A}\|_\infty = \max_{1\le i\le m}(\sum_{j=1}^{n}|a_{ij}|), \tag{2.10}$$

is the maximum of the absolute row sum. The max norm is defined as

$$\|\boldsymbol{A}\|_{\max} = \max\{|a_{ij}|\}. \tag{2.11}$$

From the definitions of these norms, we know that they satisfy the non-negativeness condition $\|\boldsymbol{A}\| \ge 0$, the scaling condition $\|\alpha\boldsymbol{A}\| = |\alpha|\|\boldsymbol{A}\|$, and the triangle inequality $\|\boldsymbol{A}+\boldsymbol{B}\| \le \|\boldsymbol{A}\| + \|\boldsymbol{B}\|$.

Example 2.2: For the matrix $\mathbf{A} = \begin{pmatrix} 2 & 3 \\ 4 & -5 \end{pmatrix}$, we know that

$$\|\mathbf{A}\|_F = \|\mathbf{A}\|_2 = \sqrt{2^2+3^2+4^2+(-5)^2} = \sqrt{54},$$

$$\|\mathbf{A}\|_\infty = \max\begin{bmatrix} |2|+|3| \\ \\ |4|+|-5| \end{bmatrix} = 9,$$

and

$$\|\mathbf{A}\|_{\max} = 5.$$

2.2 Eigenvalues and Eigenvectors

The eigenvalues λ of a $n \times n$ square matrix $\boldsymbol{A}$ are determined by

$$\boldsymbol{A}\boldsymbol{u} = \lambda\boldsymbol{u}, \tag{2.12}$$

or

$$(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{u} = 0. \tag{2.13}$$

where $\boldsymbol{I}$ is a unitary matrix with the same size as $\boldsymbol{A}$. Any non-trivial solution requires that

$$\det |\boldsymbol{A} - \lambda \boldsymbol{I}| = 0, \tag{2.14}$$

or

$$\begin{vmatrix} a_{11} - \lambda & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} - \lambda & ... & a_{2n} \\ & \vdots & \ddots & \\ a_{n1} & a_{n2} & ... & a_{nn} - \lambda \end{vmatrix} = 0, \tag{2.15}$$

which again can be written as a polynomial

$$\lambda^n + \alpha_{n-1}\lambda^{n-1} + ... + \alpha_0 = (\lambda - \lambda_1)...(\lambda - \lambda_n) = 0, \tag{2.16}$$

where λ_i are the eigenvalues which could be complex numbers. For each eigenvalue λ, there is a corresponding eigenvector $\boldsymbol{u}$ whose direction can be uniquely determined. However, the length of the eigenvector is not unique because any non-zero multiple of $\boldsymbol{u}$ will also satisfy equation (2.12), and thus can be considered as an eigenvector. For this reason, it is usually necessary to apply an additional condition by setting the length as unity, and subsequently the eigenvector becomes a unit eigenvector.

In general, a real $n \times n$ matrix $\boldsymbol{A}$ has n eigenvalues $\lambda_i (i = 1, 2, ..., n)$, however, these eigenvalues are not necessarily distinct. If the real matrix is symmetric, that is to say $\boldsymbol{A}^T = \boldsymbol{A}$, then the matrix has n distinct eigenvectors, and all the eigenvalues are real numbers. Furthermore, the inverse of a positive definite matrix is also positive definite. For a linear system $\boldsymbol{Au} = \boldsymbol{f}$ where $\boldsymbol{f}$ is a known column vector, if $\boldsymbol{A}$ is positive definite, then the system can be solved more efficiently by the matrix decomposition method.

Example 2.3: The eigenvalues of the square matrix

$$\boldsymbol{A} = \begin{pmatrix} 4 & 9 \\ 2 & -3 \end{pmatrix},$$

can be obtained by solving

$$\begin{vmatrix} 4-\lambda & 9 \\ 2 & -3-\lambda \end{vmatrix} = 0.$$

We have

$$(4-\lambda)(-3-\lambda) - 18 = (\lambda - 6)(\lambda + 5) = 0.$$

Thus, the eigenvalues are $\lambda = 6$ and $\lambda = -5$. Let $\boldsymbol{v} = (v_1\ v_2)^T$ be the eigenvector, we have for $\lambda = 6$

$$|\boldsymbol{A} - \lambda \boldsymbol{I}| = \begin{pmatrix} -2 & 9 \\ 2 & -9 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = 0,$$

which means that

$$-2v_1 + 9v_2 = 0, \qquad 2v_1 - 9v_2 = 0.$$

These two equations are virtually the same (not linearly independent), so the solution is

$$v_1 = \frac{9}{2} v_2.$$

Any vector parallel to $\boldsymbol{v}$ is also an eigenvector. In order to get a unique eigenvector, we have to impose an extra requirement, that is the length of the vector is unity. We now have

$$v_1^2 + v_2^2 = 1,$$

or

$$(\frac{9v_2}{2})^2 + v_2^2 = 1,$$

which gives $v_2 = \pm 2/\sqrt{85}$, and $v_1 = \pm 9/\sqrt{85}$. As these two vectors are in opposite direction, we can choose any of the two directions. So the eigenvector for the eigenvalue $\lambda = 6$ is

$$\boldsymbol{v} = \begin{pmatrix} 9/\sqrt{85} \\ \\ 2/\sqrt{85} \end{pmatrix}.$$

Similarly, the corresponding eigenvector for the eigenvalue $\lambda = -5$ is $\boldsymbol{v} = (-\sqrt{2}/2 \;\; \sqrt{2}/2)^T$.

A square symmetric matrix $\boldsymbol{A}$ is said to be positive definite if all its eigenvalues are strictly positive ($\lambda_i > 0$ where $i = 1, 2, ..., n$). By multiplying (2.12) by $\boldsymbol{u}^T$, we have

$$\boldsymbol{u}^T \boldsymbol{A}\boldsymbol{u} = \boldsymbol{u}^T \lambda \boldsymbol{u} = \lambda \boldsymbol{u}^T \boldsymbol{u}, \tag{2.17}$$

which leads to

$$\lambda = \frac{\boldsymbol{u}^T \boldsymbol{A}\boldsymbol{u}}{\boldsymbol{u}^T \boldsymbol{u}}. \tag{2.18}$$

This means that

$$\boldsymbol{u}^T \boldsymbol{A}\boldsymbol{u} > 0, \qquad \text{if } \lambda > 0. \tag{2.19}$$

In fact, for any vector $\boldsymbol{v}$, the following relationship holds

$$\boldsymbol{v}^T \boldsymbol{A}\boldsymbol{v} > 0. \tag{2.20}$$

For $\boldsymbol{v}$ can be a unit vector, thus all the diagonal elements of $\boldsymbol{A}$ should be strictly positive as well. If all the eigenvalues are non-negative or $\lambda_i \geq 0$, then the matrix is called positive semi-definite. In general, an indefinite matrix can have both positive and negative eigenvalues.

Example 2.4: In general, a 2×2 symmetric matrix $\boldsymbol{A}$

$$\boldsymbol{A} = \begin{pmatrix} \alpha & \beta \\ \beta & \gamma \end{pmatrix},$$

is positive definite if

$$\alpha u_1^2 + 2\beta u_1 u_2 + \gamma u_2^2 > 0,$$

for all $\boldsymbol{u} = (u_1, u_2)^T \neq 0$. The inverse of $\boldsymbol{A}$ is

$$\boldsymbol{A}^{-1} = \frac{1}{\alpha\gamma - \beta^2} \begin{pmatrix} \gamma & -\beta \\ -\beta & \alpha \end{pmatrix},$$

which is also positive definite.

From the previous example, we know that the eigenvalues of

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix},$$

are $\lambda = 3, -1$. So the matrix is indefinite. For another matrix

$$\boldsymbol{B} = \begin{pmatrix} 4 & 6 \\ 6 & 20 \end{pmatrix},$$

we can find its eigenvalues using the similar method as discussed earlier, and the eigenvalues are $\lambda = 2, 22$. So matrix $\boldsymbol{B}$ is positive definite. The inverse of $\boldsymbol{B}$

$$\boldsymbol{B}^{-1} = \frac{1}{44}\begin{pmatrix} 20 & -6 \\ -6 & 4 \end{pmatrix},$$

is also positive definite because $\boldsymbol{B}^{-1}$ has two eigenvalues: $\lambda = 1/2, 1/22$.

2.3 Spectral Radius of Matrices

Another important concept related to the eigenvalues of matrix is the spectral radius of a square matrix. If $\lambda_i (i = 1, 2, ..., n)$ are the eigenvalues (either real or complex) of a matrix $\boldsymbol{A}$, then the spectral radius $\rho(\boldsymbol{A})$ is defined as

$$\rho(\boldsymbol{A}) \equiv \max_{1 \le i \le n} \{|\lambda_i|\}, \tag{2.21}$$

which is the maximum absolute value of all the eigenvalues. Geometrically speaking, if we plot all the eigenvalues of the matrix $\boldsymbol{A}$ on the complex plane, and draw a circle on a complex plane so that it encloses all the eigenvalues inside, then the minimum radius of such a circle is the spectral radius.

For any $0 < p \in \Re$, the eigenvectors have non-zero norms $\|\boldsymbol{u}\| \neq 0$ and $\|\boldsymbol{u}^p\| \neq 0$. Using $\boldsymbol{A}\boldsymbol{u} = \lambda \boldsymbol{u}$ and taking the norms, we have

$$|\lambda|^p \|\boldsymbol{u}^p\| = \|(\lambda \boldsymbol{u})^p\| = \|(\boldsymbol{A}\boldsymbol{u})^p\| \le \|\boldsymbol{A}^p\| \|\boldsymbol{u}^p\|. \tag{2.22}$$

By dividing both sides of the above equation by $\|\boldsymbol{u}^p\| \neq 0$, we reach the following inequality

$$|\lambda|^p \leq \|\boldsymbol{A}^p\|^{1/p}, \tag{2.23}$$

which is valid for any eigenvalue. Therefore, it should also be valid for the maximum absolute value or $\rho(\boldsymbol{A})$. We finally have

$$\rho(\boldsymbol{A}) \leq \|\boldsymbol{A}^p\|^{1/p}, \tag{2.24}$$

which becomes an equality when $p \to \infty$.

The spectral radius is very useful in determining whether an iteration algorithm is stable or not. Most iteration schemes can be written as

$$\boldsymbol{u}^{(n+1)} = \boldsymbol{A}\boldsymbol{u}^{(n)} + \boldsymbol{b}, \tag{2.25}$$

where $\boldsymbol{b}$ is a known column vector and $\boldsymbol{A}$ is a square matrix with known coefficients. The iterations start from an initial guess $\boldsymbol{u}^{(0)}$ (often, set $\boldsymbol{u}^{(0)} = \boldsymbol{0}$), and proceed to the approximate solution $\boldsymbol{u}^{(n+1)}$. For the iteration procedure to be stable, it requires that $\rho(\boldsymbol{A}) \leq 1$. If $\rho(\boldsymbol{A}) > 1$, then the algorithm will not be stable and any initial errors will be amplified in each iteration.

In the case of $\mathbf{A}$ is a lower (or upper) matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & 0 & ... & 0 \\ a_{21} & a_{22} & ... & 0 \\ \vdots & & \ddots & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}, \tag{2.26}$$

then its eigenvalues are the diagonal entries: a_{11}, a_{22}, ..., a_{nn}. In addition, the determinant of the triangular matrix $\mathbf{A}$ is simply the product of its diagonal entries. That is

$$\det(\mathbf{A}) = |\mathbf{A}| = \prod_{i=1}^{n} a_{ii} = a_{11}a_{22}...a_{nn}. \tag{2.27}$$

Obviously, a diagonal matrix

$$\mathbf{D} = \text{diag}(d_1, d_2, ..., d_n) = \begin{pmatrix} d_1 & 0 & ... & 0 \\ 0 & d_2 & ... & 0 \\ & & \ddots & \\ 0 & 0 & ... & d_n \end{pmatrix}, \tag{2.28}$$

is just a special case of a triangular matrix. Thus, the properties for its inverse, eigenvalues and determinant are the same as the above.

These properties are convenient in determining the stability of an iteration scheme such as the Jacobi-type and Gauss-Seidel iteration methods where $\mathbf{A}$ may contain triangular matrices.

Example 2.5: Determine if the following iteration is stable or not

$$\begin{pmatrix} u_1 \\ u_2 \\ u_2 \end{pmatrix}_{n+1} = \begin{pmatrix} 5 & 2 & 0 \\ 1 & -2 & 2 \\ 4 & 1/2 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} + \begin{pmatrix} 2 \\ -2 \\ 5 \end{pmatrix}.$$

We know the eigenvalues of

$$\mathbf{A} = \begin{pmatrix} 5 & 2 & 0 \\ 1 & -2 & 2 \\ 4 & 1/2 & -2 \end{pmatrix},$$

are

$$\lambda_1 = 5.5548, \lambda_2 = -2.277 + 1.0556i, \lambda_3 = -2.277 - 1.0556i.$$

Then the spectral radius is

$$\rho(\boldsymbol{A}) = \max_{i \in \{1,2,3\}} (|\lambda_i|) \approx 5.55 > 1,$$

therefore, the iteration process will not be convergent.

2.4 Hessian Matrix

The gradient vector of a multivariate function $f(\boldsymbol{x})$ is defined as

$$\boldsymbol{G}_1(\boldsymbol{x}) \equiv \nabla f(\boldsymbol{x}) \equiv \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)^T, \tag{2.29}$$

where $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T$ is a vector. As the gradient $\nabla f(\boldsymbol{x})$ of a linear function $f(\boldsymbol{x})$ is always a constant vector $\boldsymbol{k}$, then any linear function can be written as

$$f(\boldsymbol{x}) = \boldsymbol{k}^T\boldsymbol{x} + \boldsymbol{b}, \tag{2.30}$$

where $\boldsymbol{b}$ is a constant vector.

The second derivatives of a generic function $f(\boldsymbol{x})$ form an $n \times n$ matrix, called Hessian matrix, given by

$$\boldsymbol{G}_2(\boldsymbol{x}) \equiv \nabla^2 f(\boldsymbol{x}) \equiv \begin{pmatrix} \frac{\partial f}{\partial x_1^2} & \cdots & \frac{\partial f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial {x_n}^2} \end{pmatrix}, \tag{2.31}$$

which is symmetric due to the fact that

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}. \tag{2.32}$$

When the Hessian matrix $\boldsymbol{G}_2(\boldsymbol{x}) = \boldsymbol{A}$ is a constant matrix (the values of its entries are independent of $\boldsymbol{x}$), the function $f(\boldsymbol{x})$ is called a quadratic function, and can subsequently be written as

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} + \boldsymbol{k}^T\boldsymbol{x} + \boldsymbol{b}. \tag{2.33}$$

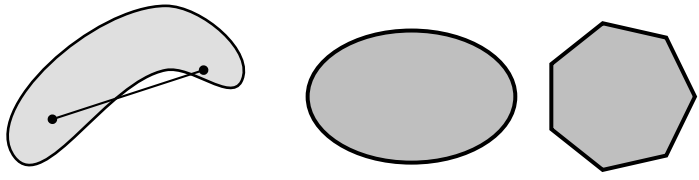
The factor $1/2$ in the expression is to avoid the appearance everywhere of a factor 2 in the derivatives, and this choice is purely out of convenience.

Example 2.6: The gradient of $f(x, y, z) = x^2 + y^2 + yz\sin(x)$ is

$$\boldsymbol{G}_1 = \begin{pmatrix} 2x + yz\cos(x) & 2y + z\sin(x) & y\sin(x) \end{pmatrix}^T.$$

The Hessian matrix is given by

$$\boldsymbol{G}_2 = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} \end{pmatrix} = \begin{pmatrix} 2 - yz\sin(x) & z\cos(x) & y\cos(x) \\ z\cos(x) & 2 & \sin(x) \\ y\cos(x) & \sin(x) & 0 \end{pmatrix}.$$



(a) (b)

Figure 2.1: Convexity: (a) non-convex, and (b) convex.

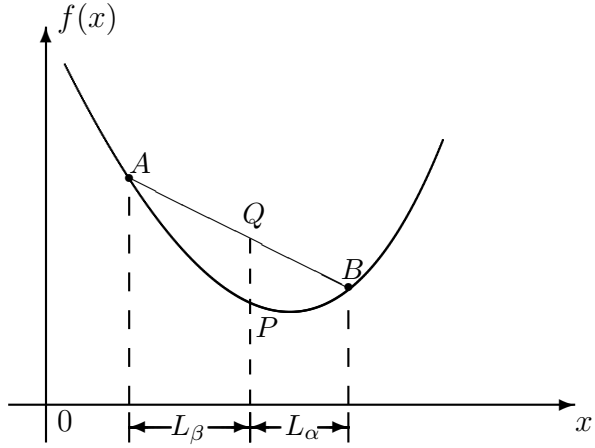


Figure 2.2: Convexity of a function $f(x)$. Chord AB lies above the curve segment joining A and B. For any point P, we have $L_\alpha = \alpha L$, $L_\beta = \beta L$ and $L = |x_B - x_A|$.

2.5 Convexity

Nonlinear programming problems are often classified according to the convexity of the defining functions. Geometrically speaking, an object is convex if for any two points within the object, every point on the straight line segment joining them is also within the object. Examples are a solid ball, a cube or a pyramid. Obviously, a hollow object is not convex. Three examples are given in Fig. 2.1.

Mathematically speaking, a set $S \in \Re^n$ in a real vector

space is called a convex set if

$$tx + (1-t)y \in S, \qquad \forall (x,y) \in S, \;\; t \in [0,1]. \tag{2.34}$$

A function $f(x)$ defined on a convex set Ω is called convex if and only if it satisfies

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y), \qquad \forall x, y \in \Omega, \tag{2.35}$$

and

$$\alpha \geq 0, \;\; \beta \geq 0, \;\; \alpha + \beta = 1. \tag{2.36}$$

Geometrically speaking, the chord AB lies above the curve segment APB joining A and B (see Fig. 2.2). For example, for any point P between A and B, we have $x_P = \alpha x_A + \beta x_B$ with

$$\alpha = \frac{L_\alpha}{L} = \frac{x_B - x_P}{x_B - x_A} \geq 0,$$

$$\beta = \frac{L_\beta}{L} = \frac{x_P - x_A}{x_B - x_A} \geq 0, \tag{2.37}$$

which indeed gives $\alpha + \beta = 1$. In addition, we know that

$$\alpha x_A + \beta x_B = \frac{x_A(x_B - x_P)}{x_B - x_A} + \frac{x_B(x_P - x_A)}{x_B - x_A} = x_P. \tag{2.38}$$

The value of the function $f(x_P)$ at P should be less than or equal to the weighted combination $\alpha f(x_A) + \beta f(x_B)$ (or the value at point Q). That is

$$f(x_P) = f(\alpha x_A + \beta x_B) \leq \alpha f(x_A) + \beta f(x_B). \tag{2.39}$$

Example 2.7: For example, the convexity of $f(x) = x^2 - 1$ requires

$$(\alpha x + \beta y)^2 - 1 \leq \alpha(x^2 - 1) + \beta(y^2 - 1), \;\; \forall x, y \in \Re, \tag{2.40}$$

where $\alpha, \beta \geq 0$ and $\alpha + \beta = 1$. This is equivalent to require

$$\alpha x^2 + \beta y^2 - (\alpha x + \beta y)^2 \geq 0, \tag{2.41}$$

where we have used $\alpha + \beta = 1$. We now have

$$\alpha x^2 + \beta y^2 - \alpha^2 x^2 - 2\alpha\beta xy - \beta^2 y^2$$

$$= \alpha(1-\alpha)(x-y)^2 = \alpha\beta(x-y)^2 \geq 0, \tag{2.42}$$

which is always true because $\alpha, \beta \geq 0$ and $(x-y)^2 \geq 0$. Therefore, $f(x) = x^2 - 1$ is convex for $\forall x \in \Re$.

A function $f(x)$ on Ω is concave if and only if $g(x) = -f(x)$ is convex. An interesting property of a convex function f is that the vanishing of the gradient $df/dx|_{x_*} = 0$ guarantees that the point x_* is a global minimum of f. If a function is not convex or concave, then it is much more difficult to find global minima or maxima.

Chapter 3

Root-Finding Algorithms

The essence of finding the solution of an optimization problem is equivalent to finding the critical points and extreme points. In order to find the critical points, we have to solve the stationary conditions when the first derivatives are zero, though it is a different matter for the extreme points at boundaries. Therefore, root-finding algorithms are important. Close-form solutions are rare, and in most cases, only approximate solutions are possible. In this chapter, we will introduce the fundamentals of the numerical techniques concerning root-finding algorithms.

3.1 Simple Iterations

The essence of root-finding algorithms is to use iteration procedure to obtain the approximate (well sometimes quite accurate) solutions, starting from some initial guess solution. For example, even ancient Babylonians knew how to find the square root of 2 using the iterative method. From the numerical technique we learnt at school, we know that we can numerically compute the square root of any real number k (so that $x = \sqrt{k}$) using

the equation

$$x = \frac{1}{2}(x + \frac{k}{x}), \tag{3.1}$$

starting from a random guess, say, $x = 1$. The reason is that the above equation can be rearranged to get $x = \sqrt{k}$. In order to carry out the iteration, we use the notation x_n for the value of x at n-th iteration. Thus, equation (3.1) provides a way of calculating the estimate of x at $n + 1$ (denoted as x_{n+1}). We have

$$x_{n+1} = \frac{1}{2}(x_n + \frac{k}{x_n}). \tag{3.2}$$

If we start from an initial value, say, $x_0 = 1$ at $n = 0$, we can do the iterations to meet the accuracy we want.

Example 3.1: To find $\sqrt{5}$, we have $k = 5$ with an initial guess $x_0 = 1$, and the first five iterations are as follows:

$$x_1 = \frac{1}{2}(x_0 + \frac{5}{x_0}) = 3, \tag{3.3}$$

$$x_2 = \frac{1}{2}(x_1 + \frac{5}{x_1}) \approx 2.333333333, \tag{3.4}$$

$$x_3 \approx 2.238095238, \quad x_4 \approx 2.236068895, \tag{3.5}$$

$$x_5 \approx 2.236067977. \tag{3.6}$$

We can see that x_5 after 5 iterations is very close to its true value $\sqrt{5} = 2.23606797749979...$, which shows that the iteration method is quite efficient.

The reason that this iterative process works is that the series $x_1, x_2, ..., x_n$ converges into the true value $\sqrt{k}$ due to the fact that $x_{n+1}/x_n = \frac{1}{2}(1 + k/x_n^2) \to 1$ as $x_n \to \sqrt{k}$. However, a good choice of the initial value x_0 will speed up the convergence. Wrong choice of x_0 could make the iteration fail, for example, we cannot use $x_0 = 0$ as the initial guess, and we cannot use $x_0 < 0$ either as $\sqrt{k} > 0$ (in this case, the iterations will approach another root $-\sqrt{k}$).

So a sensible choice should be an educated guess. At the initial step, if $x_0^2 < k$, x_0 is the lower bound and k/x_0 is upper

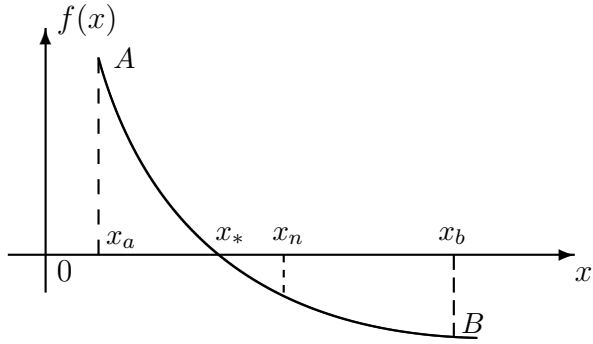


Figure 3.1: Bisection method for finding the root x_* of $f(x_*) = 0$ between two bounds x_a and x_b in the domain $x \in [a, b]$.

bound. If $x_0^2 > k$, then x_0 is the upper bound and k/x_0 is the lower bound. For other iterations, the new bounds will be x_n and k/x_n. In fact, the value x_{n+1} is always between these two bounds x_n and k/x_n, and the new estimate x_{n+1} is thus the mean or average of the two bounds. This guarantees that the series converges into the true value of $\sqrt{k}$. This method is similar to the bisection method below.

3.2 Bisection Method

The above-mentioned iteration method to find $x = \sqrt{k}$ is in fact equivalent to find the solution or the root of the function $f(x) = x^2 - k = 0$. For any function $f(x)$ in the interval $[a, b]$, the root-finding bisection method works in the following way as shown in Fig. 3.1.

The iteration procedure starts with two initial guessed bounds x_a (lower bound), and x_b (upper bound) so that the true root $x = x_*$ lies between these two bounds. This requires that $f(x_a)$ and $f(x_b)$ have different signs. In our case shown in Fig. 3.1, $f(x_a) > 0$ and $f(x_b) < 0$, but $f(x_a)f(x_b) < 0$. The obvious choice is $x_a = a$ and $x_b = b$. The next estimate is just the

midpoint of A and B, and we have

$$x_n = \frac{1}{2}(x_a + x_b). \tag{3.7}$$

We then have to test the sign of $f(x_n)$. If $f(x_n) < 0$ (having the same sign as $f(x_b)$), we then update the new upper bound as $x_b = x_n$. If $f(x_n) > 0$ (having the same sign as $f(x_a)$), we update the new lower bound as $x_a = x_n$. In a special case when $f(x_n) = 0$, we have found the true root. The iterations continue in the same manner until a given accuracy is achieved or the prescribed number of iterations is reached.

Example 3.2: If we want to find $\sqrt{\pi}$, we have

$$f(x) = x^2 - \pi = 0.$$

We can use $x_a = 1$ and $x_b = 2$ since $\pi < 4$ (thus $\sqrt{\pi} < 2$). The first bisection point is

$$x_1 = \frac{1}{2}(x_a + x_b) = \frac{1}{2}(1 + 2) = 1.5.$$

Since $f(x_a) < 0$, $f(x_b) > 0$ and $f(x_1) = -0.8916 < 0$, we update the new lower bound $x_a = x_1 = 1.5$. The second bisection point is

$$x_2 = \frac{1}{2}(1.5 + 2) = 1.75,$$

and $f(x_2) = -0.0791 < 0$, so we update lower bound again $x_a = 1.75$. The third bisection point is

$$x_3 = \frac{1}{2}(1.75 + 2) = 1.875.$$

Since $f(x_3) = 0.374 > 0$, we now update the new upper bound $x_b = 1.875$. The fourth bisection point is

$$x_4 = \frac{1}{2}(1.75 + 1.875) = 1.8125.$$

It is within 2.5% from the true value of $\sqrt{\pi} \approx 1.7724538509$.

In general, the convergence of the bisection method is very slow, and Newton's method is a much better choice in most cases.

3.3 Newton's Method

Newton's method is a widely-used classic method for finding the zeros of a nonlinear univariate function of $f(x)$ on the interval $[a, b]$. It is also referred to as the Newton-Raphson method. At any given point x_n shown in Fig. 3.2, we can approximate the function by a Taylor series for $\Delta x = x_{n+1} = x_n$ about x_n,

$$f(x_{n+1}) = f(x_n + \Delta x) \approx f(x_n) + f'(x_n)\Delta x, \tag{3.8}$$

which leads to

$$x_{n+1} - x_n = \Delta x \approx \frac{f(x_{n+1}) - f(x_n)}{f'(x_n)}, \tag{3.9}$$

or

$$x_{n+1} \approx x_n + \frac{f(x_{n+1}) - f(x_n)}{f'(x_n)}. \tag{3.10}$$

Since we try to find an approximation to $f(x) = 0$ with $f(x_{n+1})$, we can use the approximation $f(x_{n+1}) \approx 0$ in the above expression. Thus we have the standard Newton iterative formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{3.11}$$

The iteration procedure starts from an initial guess x_0 and continues until certain criterion is met. A good initial guess will use less number of steps, however, if there is no obvious initial good starting point, you can start at any point on the interval $[a, b]$. But if the initial value is too far from the true zero, the iteration process may fail. So it is a good idea to limit the number of iterations.

Example 3.3: To find the root of

$$f(x) = x - e^{-x} = 0,$$

we use the Newton's method starting from $x_0 = 1$. We know that

$$f'(x) = 1 + e^{-x},$$

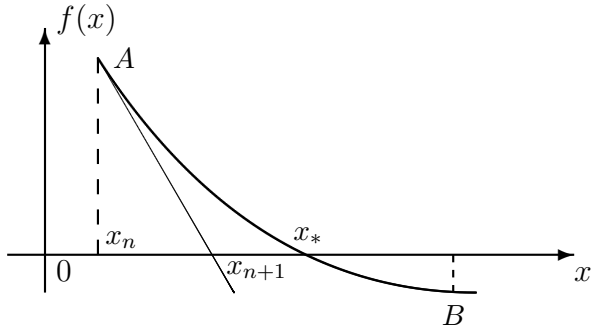


Figure 3.2: Newton's method to approximate the root x_* by x_{n+1} from the pervious value x_n.

and thus the iteration formula becomes

$$x_{n+1} = x_n - \frac{x_n - e^{-x_n}}{1 + e^{-x_n}}.$$

Using $x_0 = 1$, we have

$$x_1 = 1 - \frac{1 - e^{-1}}{1 + e^{-1}} \approx 0.5378828427,$$

and

$$x_2 \approx 0.5669869914, \qquad x_3 \approx 0.5671432859.$$

We can see that x_3 (only three iterations) is very close to the true root is $x_* \approx 0.5671432904$.

We have seen that Newton's method is very efficient and is thus so widely used. This method can be modified for solving unconstrained optimization problems because it is equivalent to find the root of the first derivative $f'(\boldsymbol{x}) = 0$ once the objective function $f(\boldsymbol{x})$ is given.

3.4 Iteration Methods

Sometimes we have to find roots of functions of multiple variables, and Newton's method can be extended to carry out such

task. For nonlinear multivariate functions

$$\boldsymbol{F}(\boldsymbol{x}) = [F_1(\boldsymbol{x}), F_2(\boldsymbol{x}), ..., F_N(\boldsymbol{x})]^T, \tag{3.12}$$

where $\boldsymbol{x} = (x, y, ..., z)^T = (x_1, x_2, ..., x_p)^T$, an iteration method is usually needed to find the roots

$$\boldsymbol{F}(\boldsymbol{x}) = 0. \tag{3.13}$$

The Newton-Raphson iteration procedure is widely used. We first approximate $\boldsymbol{F}(\boldsymbol{x})$ by a linear residual function $\mathbf{R}(\boldsymbol{x}; \boldsymbol{x}^n)$ in the neighbourhood of an existing approximation $\boldsymbol{x}^n$ to $\boldsymbol{x}$, and we have

$$\boldsymbol{R}(\boldsymbol{x}, \boldsymbol{x}^n) = \boldsymbol{F}(\boldsymbol{x}^n) + \mathbf{J}(\boldsymbol{x}^n)(\boldsymbol{x} - \boldsymbol{x}^n), \tag{3.14}$$

and

$$\mathbf{J}(\boldsymbol{x}) = \nabla \boldsymbol{F}, \tag{3.15}$$

where $\mathbf{J}$ is the Jacobian of $\boldsymbol{F}$. That is

$$\mathbf{J}_{ij} = \frac{\partial F_i}{\partial x_j}. \tag{3.16}$$

Here we have used the notation $\boldsymbol{x}^n$ for the vector $\boldsymbol{x}$ at the n-th iteration, which should not be confused with the power $\boldsymbol{u}^n$ of a vector $\boldsymbol{u}$. This might be confusing, but such notations are widely used in the literature of numerical analysis. An alternative (and better) notation is to denote $\boldsymbol{x}^n$ as $\boldsymbol{x}^{(n)}$, which shows the vector value at n-th iteration using a bracket. However, we will use both notations if no confusion could arise.

To find the next approximation $\boldsymbol{x}^{n+1}$ from the current estimate $\boldsymbol{x}^n$, we have to try to satisfy $\boldsymbol{R}(\boldsymbol{x}^{n+1}, \mathbf{u}^n) = 0$, which is equivalent to solve a linear system with $\mathbf{J}$ being the coefficient matrix

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n - \mathbf{J}^{-1}\mathbf{F}(\boldsymbol{x}^n), \tag{3.17}$$

under a given termination criterion

$$\|\boldsymbol{x}^{n+1} - \boldsymbol{x}^n\| \leq \epsilon.$$

Iterations require an initial starting vector $\boldsymbol{x}^0$, which is often set to $\boldsymbol{x}^0 = 0$.

Example 3.4: To find the roots of the system

$$x - e^{-y} = 0, \qquad x^2 - y = 0,$$

we first write it as

$$\boldsymbol{F}(\boldsymbol{x}) = \begin{pmatrix} x_1 - e^{-x_2} \\ x_1^2 - x_2 \end{pmatrix}, \qquad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}.$$

The Newton-Raphson iteration formula becomes

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n - \mathbf{J}^{-1}\boldsymbol{F}(\boldsymbol{x}^n),$$

where the Jacobian $\boldsymbol{J}$ is

$$\mathbf{J} = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 1 & e^{-x_2} \\ 2x_1 & -1 \end{pmatrix},$$

whose inverse is

$$\boldsymbol{A} = \mathbf{J}^{-1} = \frac{1}{-1 - 2x_1 e^{-x_2}} \begin{pmatrix} -1 & -e^{-x_2} \\ -2x_1 & 1 \end{pmatrix}$$

$$= \frac{1}{1 + 2x_1 e^{-x_2}} \begin{pmatrix} 1 & e^{-x_2} \\ 2x_1 & -1 \end{pmatrix}.$$

Therefore, the iteration equation becomes

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n - \boldsymbol{u}^n$$

where

$$\boldsymbol{u}^n = \boldsymbol{J}^{-1}\boldsymbol{F}(\boldsymbol{x}^n) = \frac{1}{1 + 2x_2 e^{-x_2}} \begin{pmatrix} 1 & e^{-x_2} \\ 2x_1 & -1 \end{pmatrix} \begin{pmatrix} x_1 - e^{-x_2} \\ x_1^2 - x_2 \end{pmatrix}$$

$$= \frac{1}{1 + 2x_1 e^{-x_2}} \begin{pmatrix} x_1 + (x_1^2 - 1 - x_2)e^{-x_2} \\ x_1^2 + x_2 - 2x_1 e^{-x_2} \end{pmatrix}.$$

If we start with the initial guess $\boldsymbol{x}^0 = (0,0)^T$, we have the first estimate $\boldsymbol{x}^1$ as

$$\boldsymbol{x}^1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

and the second iteration gives

$$\boldsymbol{x}^2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.33333 \\ -0.33333 \end{pmatrix} = \begin{pmatrix} 0.66667 \\ 0.3333 \end{pmatrix}.$$

If we continue this way, the third iteration gives

$$\boldsymbol{x}^3 = \boldsymbol{x}^2 - \begin{pmatrix} 0.01520796 \\ -0.09082847 \end{pmatrix} = \begin{pmatrix} 0.6514462 \\ 0.42415551 \end{pmatrix}.$$

Finally, the fourth iteration gives

$$\boldsymbol{x}^4 = \boldsymbol{x}^3 - \begin{pmatrix} -0.001472389 \\ -0.002145006 \end{pmatrix} = \begin{pmatrix} 0.65291859 \\ 0.42630051 \end{pmatrix}.$$

The true roots occur at $(0.6529186405, 0.4263027510)$, and we can see that even after only four iterations, the estimates are very close to the true values.

With these fundamentals of mathematics and numerical techniques, we are now ready to solve optimization problems. In Part II, we will introduce the conventional methods that are widely used in mathematical programming.

Chapter 4

System of Linear Equations

4.1 Linear systems

A linear system of m equations for n unknowns

$$\begin{aligned} a_{11}u_1 + a_{12}u_2 + ... + a_{1n}u_n &= b_1, \\ a_{21}u_1 + a_{22}u_2 + ... + a_{2n}u_n &= b_2, \\ \vdots \qquad\qquad & \quad \vdots \\ a_{m1}u_1 + a_{m2}u_2 + ... + a_{mn}u_n &= b_n, \end{aligned} \tag{4.1}$$

can be written in the compact form as

$$\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ \vdots & \vdots & & \\ a_{m1} & a_{m2} & ... & a_{mn} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \tag{4.2}$$

or simply

$$\mathbf{Au} = \mathbf{b}. \tag{4.3}$$

If $m < n$, the system is under-determined as the conditions are not adequate to guarantee a unique solution. On the other

hand, the system is over-determined if $m > n$ because there are too many conditions and a solution may not exist at all. The unique solution is only possible when $m = n$.

The solution of this matrix equation is important to many numerical problems, ranging from the solution of a large system of linear equations to linear mathematical programming, and from data interpolation to finding solutions to finite element problems.

The inverse of $\mathbf{A}$ is possible only if $m = n$. If the inverse $\mathbf{A}^{-1}$ does not exist, then the linear system is under-determined or there are no unique solutions (or even no solution at all). In order to find the solutions, we multiply both sides by $\mathbf{A}^{-1}$,

$$\mathbf{A^{-1}Au} = \mathbf{A^{-1}b}, \tag{4.4}$$

and we obtain the solution

$$\mathbf{u} = \mathbf{A}^{-1}\mathbf{b}. \tag{4.5}$$

A special case of the above equation is when $\mathbf{b} = \lambda\mathbf{u}$, and this becomes an eigenvalue problem. An eigenvalue λ and corresponding eigenvector $\mathbf{v}$ of a square matrix $\mathbf{A}$ satisfy

$$\mathbf{Av} = \lambda\mathbf{v}, \tag{4.6}$$

or

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}. \tag{4.7}$$

Eigenvalues have the interesting connections with the matrix,

$$\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii} = \lambda_1 + \lambda_2 + ... + \lambda_n, \tag{4.8}$$

and

$$\det(\mathbf{A}) = \prod_{i=1}^{n} \lambda_i. \tag{4.9}$$

For a symmetric square matrix, the two eigenvectors for two distinct eigenvalues λ_i and λ_j are orthogonal $\mathbf{v^T v} = 0$.

Mathematically speaking, a linear system can be solved in principle using the Cramer's rule,

$$u_i = \frac{\det \mathbf{A}_i}{\det \mathbf{A}}, \qquad i = 1, 2, ..., n, \tag{4.10}$$

where the matrix $\mathbf{A}_i$ is obtained by replacing the i-th column by the column vector $\mathbf{b}$. For three linear equations with three unknown u_1, u_2 and u_3,

$$\begin{aligned} a_{11}u_1 + a_{12}u_2 + a_{13}u_3 &= b_1, \\ a_{21}u_1 + a_{22}u_2 + a_{23}u_3 &= b_2, \\ a_{31}u_1 + a_{32}u_2 + a_{33}u_3 &= b_3, \end{aligned} \tag{4.11}$$

its solution vector is given by the following Cramer's rule

$$u_1 = \frac{1}{\Delta}\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}, \qquad u_2 = \frac{1}{\Delta}\begin{vmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{vmatrix},$$

$$u_3 = \frac{1}{\Delta}\begin{vmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{32} & b_3 \end{vmatrix},$$

where

$$\Delta = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}. \tag{4.12}$$

Though it is straightforward to extend to any dimensions in theory, in practice this is not an easy task because the calculation of the determinant of a large matrix is not easy. Though, the Cramer's rule is good for theorem-proof, but it is not good for numerical implementation. A far better method is to use the inverse matrix.

Finding the inverse $\mathbf{A}^{-1}$ of a square $n \times n$ matrix $\mathbf{A}$ is not an easy task either, especially when the size of the matrix is large and it usually requires the algorithm complexity of $O(n^3)$. In fact, many solution methods intend to avoid the calculations

of finding the inverse $\mathbf{A}^{-1}$ if possible. There are many ways of solving the linear equations, but they fall into two categories: direct algebraic methods and iteration methods. The former intends to find the solution by elimination, decomposition of the matrix, and substitutions, while the later involves certain iterations to find the approximate solutions. The choice of these methods depends on the characteristics of the matrix $\mathbf{A}$, the size of the problem, computational time, the type of problem, and the required solution quality.

4.2 Gauss Elimination

The basic idea of Gauss elimination is to transform the square matrix into a triangular matrix by elementary row operations, so that the simplified triangular system can be solved by direct back substitution. For the linear system

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1n} \\ a_{21} & a_{22} & a_{23} & ... & a_{2n} \\ & \vdots & & & \\ a_{n1} & a_{n2} & a_{n3} & ... & a_{nn} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \tag{4.13}$$

the aim in the first step is to try to make all the coefficients in the first column (a_{21}, ..., a_{n1}) become zero except the first element by elementary row operations. This is based on the principle that a linear system will remain the same if its rows are multiplied by some non-zero coefficients or any two rows are interchanged or any two (or more) rows are combined through addition and substraction.

To do this, we first divide the first equation by a_{11} (we can always assume $a_{11} \neq 0$, if not, we re-arrange the order of the equations to achieve this). We now have

$$\begin{pmatrix} 1 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} & ... & \frac{a_{1n}}{a_{11}} \\ a_{21} & a_{22} & a_{23} & ... & a_{2n} \\ & \vdots & & & \\ a_{n1} & a_{n2} & a_{n3} & ... & a_{nn} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} \frac{b_1}{a_{11}} \\ b_2 \\ \vdots \\ b_n \end{pmatrix}. \tag{4.14}$$

Then multiplying the first row by $-a_{21}$ and adding it to the second row, multiplying the first row by $-a_{i1}$ and adding it to the i-th row, we finally have

$$\begin{pmatrix} 1 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} & ... & \frac{a_{1n}}{a_{11}} \\ 0 & a_{22}-\frac{a_{21}a_{12}}{a_{11}} & & ... & a_{2n}-\frac{a_{21}a_{1n}}{a_{11}} \\ & \vdots & & & \\ 0 & a_{n2}-\frac{a_{n1}a_{12}}{a_{11}} & & ... & a_{nn}-\frac{a_{n1}a_{1n}}{a_{11}} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} \frac{b_1}{a_{11}} \\ b_2-\frac{a_{21}b_1}{a_{11}} \\ \vdots \\ b_n-\frac{a_{n1}b_n}{a_{11}} \end{pmatrix}.$$

We then repeat the same procedure for the third row to the n-th row, the final form of the linear system should be in the following generic form

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & ... & \alpha_{1n} \\ 0 & \alpha_{22} & \alpha_{23} & ... & \alpha_{2n} \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & ... & \alpha_{nn} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix}, \tag{4.15}$$

where $\alpha_{1j} = a_{1j}/a_{11}$, $\alpha_{2j} = a_{2j} - a_{1j}a_{21}/a_{11} (j = 1, 2, ..., n)$, ..., $\beta_1 = b_1/a_{11}$, $\beta_2 = b_2 - a_{21}b_1/a_{11}$ and others. From the above form, we see that $u_n = \beta_n/\alpha_{nn}$ because there is only one unknown u_n in the n-th row. We can then use the back substitution to obtain u_{n-1} and up to u_1. Therefore, we have

$$u_n = \frac{\beta_n}{\alpha_{nn}},$$

$$u_i = \frac{1}{\alpha_{ii}}(\beta_i - \sum_{j=i+1}^{n} \alpha_{ij}x_j), \tag{4.16}$$

where $i = n-1, n-2, ..., 1$. Obviously, in our present case, $\alpha_{11} = ... = \alpha_{nn} = 1$. Let us look at an example.

Example 4.1: For the linear system

$$\begin{pmatrix} 2 & -1 & 3 & 4 \\ 3 & 2 & -5 & 6 \\ -2 & 1 & 0 & 5 \\ 4 & -5 & -6 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 12 \\ -3 \end{pmatrix},$$

we first divide the first row by $a_{11} = 2$, we have

$$\begin{pmatrix} 1 & -1/2 & 3/2 & 2 \\ 3 & 2 & -5 & 6 \\ -2 & 1 & 0 & 5 \\ 4 & -5 & -6 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 21/2 \\ 9 \\ 12 \\ -3 \end{pmatrix}.$$

Multiplying the first row by 3 and subtracting it from the second row, and carrying out similar row manipulations for the other rows, we have

$$\begin{pmatrix} 1 & -1/2 & 3/2 & 2 \\ 0 & 7/2 & -19/2 & 0 \\ 0 & 0 & 3 & 9 \\ 0 & -3 & -12 & -8 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 21/2 \\ -45/7 \\ 33 \\ -45 \end{pmatrix}.$$

For the second row, we repeat this procedure again, we have

$$\begin{pmatrix} 1 & -1/2 & 3/2 & 2 \\ 0 & 1 & -19/7 & 0 \\ 0 & 0 & 3 & 9 \\ 0 & 0 & -141/7 & -8 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 21/2 \\ -45/7 \\ 33 \\ -450/7 \end{pmatrix}.$$

After the same procedure for the third row, we have

$$\begin{pmatrix} 1 & -1/2 & 3/2 & 2 \\ 0 & 1 & -19/7 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 367/7 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 21/2 \\ -45/7 \\ 11 \\ 1101/7 \end{pmatrix}.$$

The fourth row gives that $u_4 = 3$. Using the back substitution, we have $u_3 = 2$ from the third row. Similarly, we have $u_2 = -1$ and $u_1 = 1$. So the solution is

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 2 \\ 3 \end{pmatrix}.$$

We have seen from the example, there are many float-point calculations even for the simple system of three linear equations. In fact, the full Gauss elimination is computationally extensive with an algorithmic complexity of $O(2n^3/3)$.

4.3 Gauss-Jordan Elimination

Gauss-Jordan elimination is a variant of Gauss elimination which solves a linear system and, at the same time, can also compute the inverse of a square matrix. The first step is to formulate an augmented matrix from $\mathbf{A}$, $\mathbf{b}$ and the unit matrix $\mathbf{I}$ (with the same size of $\mathbf{A}$). That is

$$\mathbf{B} = [\mathbf{A}|\mathbf{b}|\mathbf{I}] = \begin{pmatrix} a_{11} & ... & a_{1n} & |b_1| & 1 & 0 & ... & 0 \\ a_{21} & ... & a_{2n} & |b_2| & 0 & 1 & ... & 0 \\ & \vdots & & \vdots & & & \vdots & \\ a_{n1} & ... & a_{nn} & |b_n| & 0 & 0 & ... & 1 \end{pmatrix}, \tag{4.17}$$

where the notation $\mathbf{A}|\mathbf{b}$ denotes the augmented form of two matrices $\mathbf{A}$ and $\mathbf{b}$. The aim is to reduce $\mathbf{B}$ to the following form by elementary row reductions in the similar way as those carried out in Gauss elimination.

$$\begin{pmatrix} 1 & 0 & ... & 0 & |u_1| & a'_{11} & ... & a'_{1n} \\ 0 & 1 & ... & 0 & |u_2| & a'_{21} & ... & a'_{2n} \\ & \vdots & & & \vdots & & \vdots & \\ 0 & 0 & ... & 1 & |u_n| & a'_{n1} & ... & a'_{nn} \end{pmatrix} = [\mathbf{I}|\mathbf{u}|\mathbf{A}^{-1}], \tag{4.18}$$

where $\mathbf{A}^{-1} = [a'_{ij}]$ is the inverse. This is better demonstrated by an example.

Example 4.2: In order to solve the following system

$$\mathbf{A}\mathbf{u} = \begin{pmatrix} 1 & 2 & 3 \\ -2 & 2 & 5 \\ 4 & 0 & -5 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -2 \\ 14 \end{pmatrix} = \mathbf{b},$$

we first write it in an augmented form

$$\mathbf{B} = \left(\begin{array}{ccc|c|ccc} 1 & 2 & 3 & 5 & 1 & 0 & 0 \\ -2 & 2 & 5 & -2 & 0 & 1 & 0 \\ 4 & 0 & -5 & 14 & 0 & 0 & 1 \end{array}\right).$$

By elementary row operations, this could be changed into

$$\mathbf{B}' = [\mathbf{I}|\mathbf{u}|\mathbf{A}^{-1}] = \begin{pmatrix} 1 & 0 & 0 & | & 1 & | & \frac{5}{7} & -\frac{5}{7} & -\frac{2}{7} \\ 0 & 1 & 0 & | & 5 & | & -\frac{5}{7} & \frac{17}{14} & \frac{11}{14} \\ 0 & 0 & 1 & | & -2 & | & \frac{4}{7} & -\frac{4}{7} & -\frac{3}{7} \end{pmatrix},$$

which gives

$$\mathbf{u} = \begin{pmatrix} 1 \\ 5 \\ -2 \end{pmatrix}, \qquad \mathbf{A}^{-1} = \frac{1}{14}\begin{pmatrix} 10 & -10 & -4 \\ -10 & 17 & 11 \\ 8 & -8 & -6 \end{pmatrix}.$$

We can see that both the solution $\mathbf{u}$ and the inverse $\mathbf{A}^{-1}$ are obtained in Gauss-Jordan elimination.

The Gauss-Jordan elimination is not quite stable numerically. In order to get better and stable schemes, common practice is to use pivoting. Basically, pivoting is a scaling procedure by dividing all the elements in a row by the element with the largest magnitude or norm. If necessary, rows can be exchanged so the the largest element is moved so that it becomes the leading coefficient, especially on the diagonal position. This makes all the scaled elements to be in the range $[-1, 1]$. Thus, exceptionally large numbers are removed, which makes the scheme more numerically stable.

An important issue in both Gauss elimination and Gauss-Jordan elimination is the non-zero requirement of leading coefficients such as $a_{11} \neq 0$. For a_{11}, it is possible to re-arrange the equations to achieve this requirement. However, there is no guarantee that other coefficients such as $a_{22} - a_{21}a_{12}/a_{11}$ should be nonzero. If it is zero, there is a potential difficulty due to the dividing by zero. In order to avoid this problem, we can use other methods such as the pivoting method and LU decomposition.

4.4 LU Factorization

Any square matrix $\mathbf{A}$ can be written as the product of two triangular matrices in the form

$$\mathbf{A} = \mathbf{LU}, \tag{4.19}$$

where $\mathbf{L}$ and $\mathbf{U}$ are the lower and upper triangular matrices, respectively. A lower (upper) triangular matrix has elements only on the diagonal and below (above). That is

$$\mathbf{L} = \begin{pmatrix} \beta_{11} & 0 & ... & 0 \\ \beta_{21} & \beta_{22} & ... & 0 \\ \vdots & & \ddots & \\ \beta_{n1} & \beta_{n2} & ... & \beta_{nn} \end{pmatrix}, \tag{4.20}$$

and

$$\mathbf{U} = \begin{pmatrix} \alpha_{11} & ... & \alpha_{1,n-1} & \alpha_{1,n} \\ & \ddots & & \vdots \\ 0 & ... & \alpha_{n-1,n-1} & \alpha_{n-1,n} \\ 0 & ... & 0 & \alpha_{nn} \end{pmatrix}. \tag{4.21}$$

The linear system $\mathbf{Au} = \mathbf{b}$ can be written as two step

$$\mathbf{Au} = (\mathbf{LU})\mathbf{u} = \mathbf{L}(\mathbf{Uu}) = \mathbf{b}, \tag{4.22}$$

or

$$\mathbf{Uu} = \mathbf{v}, \qquad \mathbf{Lv} = \mathbf{b}, \tag{4.23}$$

which are two linear systems with triangular matrices only, and these systems can be solved by forward and back substitutions. The solutions of v_i are given by

$$v_1 = \frac{b_1}{\beta_{11}}, \qquad v_i = \frac{1}{\beta_{ii}}(b_i - \sum_{j=1}^{i-1} \beta_{ij} v_j), \tag{4.24}$$

where $i = 2, 3, ..., n$. The final solutions u_i are then given by

$$u_n = \frac{v_n}{\alpha_{nn}}, \quad u_i = \frac{1}{\alpha_{ii}}(v_i - \sum_{j=i+1}^{n} \alpha_{ij} u_j), \tag{4.25}$$

where $i = n-1, ..., 1$.

For triangular matrices such as $\mathbf{L}$, there are some interesting properties. The inverse of a lower (upper) triangular matrix is also a lower (upper) triangular matrix. The determinant of the triangular matrix is simply the product of its diagonal entries. That is

$$\det(\mathbf{L}) = |\mathbf{L}| = \prod_{i=1}^{n} \beta_{ii} = \beta_{11}\beta_{22}...\beta_{nn}. \tag{4.26}$$

More interestingly, the eigenvalues of a triangular matrix are the diagonal entries: β_{11}, β_{22}, ..., β_{nn}. These properties are convenient in determining the stability of an iteration scheme.

But there is another issue here and that is how to decompose a square matrix $\mathbf{A} = [a_{ij}]$ into $\mathbf{L}$ and $\mathbf{U}$. As there are $n(n+1)/2$ coefficients α_{ij} and $n(n+1)/2$ coefficients β_{ij}, so there are $n^2 + n$ unknowns. For the equation (4.19), we know that it could provide only n^2 equations (as there are only n^2 coefficients a_{ij}). They are

$$\sum_{k=1}^{i} \beta_{ik}\alpha_{kj} = a_{ij}, \qquad (i < j), \tag{4.27}$$

$$\sum_{k=1}^{j=i} \beta_{ik}\alpha_{kj} = a_{ij}, \qquad (i = j), \tag{4.28}$$

and

$$\sum_{k=1}^{j} \beta_{ik}\alpha_{kj} = a_{ij}, \qquad (i > j), \tag{4.29}$$

which again form another system of n equations.

As $n^2 + n > n^2$, there are n free coefficients. Therefore, the factorization or decomposition is not uniquely determined. We have to impose some extra conditions. Fortunately, we can always set either $\alpha_{ii} = 1$ or $\beta_{ii} = 1$ where $i = 1, 2, ..., n$. If we set $\beta_{ii} = 1$, we can use the Crout's algorithm to determine α_{ij} and β_{ij}. We have the coefficients for the upper triangular matrix

$$\alpha_{ij} = a_{ij} - \sum_{k=1}^{i-1} \beta_{ik}\alpha_{kj}, \tag{4.30}$$

for $(i = 1, 2, ..., j)$ and $j = 1, 2, ..., n$. For the lower triangular matrix, we have

$$\beta_{ij} = \frac{1}{\alpha_{jj}}(a_{ij} - \sum_{k=1}^{j-1} \beta_{ik}\alpha_{kj}), \tag{4.31}$$

for $i = j + 1, j + 2, ..., n$.

The same issue appears again, that is, all the leading coefficients α_{ii} must be non-zero. For sparse matrices with many zero entries, this often causes some significant problems numerically. Better methods such as iteration methods should be used in this case.

4.5 Iteration Methods

For a linear system $\mathbf{Au} = \mathbf{b}$, the solution $\mathbf{u} = \mathbf{A^{-1}b}$ generally involves the inversion of a large matrix. The direct inversion becomes impractical if the matrix is very large (say, if $n > 1000000$). Many efficient algorithms have been developed for solving such systems. Jacobi and Gauss-Seidel iteration methods are just two examples.

4.5.1 Jacobi Iteration Method

The basic idea of the Jacobi-type iteration method is to decompose the $n \times n$ square matrix $\mathbf{A}$ into three simple matrices

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}, \tag{4.32}$$

where $\mathbf{D}$ is a diagonal matrix. $\mathbf{L}$ and $\mathbf{U}$ are the strictly lower and upper triangular matrices, respectively. Here the 'strict' means that the lower (or upper) triangular matrices do not include the diagonal elements. That is say, all the diagonal elements of the triangular matrices are zeros.

It is worth pointing out that here the triangular matrices $\mathbf{L}$ and $\mathbf{U}$ are different from those in the LU decomposition where it requires a matrix product. In comparison with the LU decomposition where $\mathbf{LU} = \mathbf{A}$, we have used the simple

additions here and this makes the decomposition an easier task. Using this decomposition, the linear system $\mathbf{Au} = \mathbf{b}$ becomes

$$\mathbf{Au} = (\mathbf{D} + \mathbf{L} + \mathbf{U})\mathbf{u} = \mathbf{b}, \tag{4.33}$$

which can be written as the iteration procedure

$$\mathbf{Du}^{(n+1)} = \mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{u}^{(n)}. \tag{4.34}$$

This can be used to calculate the next approximate solution $\mathbf{u}^{(n+1)}$ from current estimate $\mathbf{u}^{(n)}$. As the inverse of any diagonal matrix $\mathbf{D}$ =diag$[d_{ii}]$ is easy, we have

$$\mathbf{u}^{(n+1)} = \mathbf{D}^{-1}[\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{u}^{(n)}]. \tag{4.35}$$

Writing in terms of the elements, we have

$$u_i^{(n+1)} = \frac{1}{d_{ii}}[b_i - \sum_{j \neq i} a_{ij} u_j^{(n)}], \tag{4.36}$$

where $d_{ii} = a_{ii}$ are the diagonal elements only.

This iteration usually starts from an initial guess $\mathbf{u}^{(0)}$ (usually, $\mathbf{u}^{(0)} = \mathbf{0}$). However, this iteration scheme is only stable under the conditions that the square matrix is strictly diagonally dominant. That is to require that

$$|a_{ii}| > \sum_{j=1, j \neq i}^{n} |a_{ij}|, \tag{4.37}$$

for all $i = 1, 2, ..., n$.

In order to show how the iteration works, let us at look at an example by solving the following linear system

$$\begin{aligned} 5u_1 + u_2 - 2u_3 &= 5, \\ u_1 + 4u_2 &= -10, \\ 2u_1 + 2u_2 - 7u_3 &= -9. \end{aligned} \tag{4.38}$$

We know its exact solution is

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -3 \\ 1 \end{pmatrix}. \tag{4.39}$$

Now let us solve the system by the Jacobi-type iteration method.

Example 4.3: We first write the above simple system as the compact matrix form $\mathbf{Au} = \mathbf{b}$ or

$$\begin{pmatrix} 5 & 1 & -2 \\ 1 & 4 & 0 \\ 2 & 2 & -7 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} 5 \\ -10 \\ -9 \end{pmatrix}.$$

Then let us decompose the matrix $\mathbf{A}$ as

$$\mathbf{A} = \begin{pmatrix} 5 & 1 & -2 \\ 1 & 4 & 0 \\ 2 & 2 & -7 \end{pmatrix} = \mathbf{D} + (\mathbf{L} + \mathbf{U})$$

$$= \begin{pmatrix} 5 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & -7 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 2 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & -2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

The inverse of $\mathbf{D}$ is simply

$$\mathbf{D}^{-1} = \begin{pmatrix} 1/5 & 0 & 0 \\ 0 & 1/4 & 0 \\ 0 & 0 & -1/7 \end{pmatrix}.$$

The Jacobi-type iteration formula is

$$\mathbf{u}^{(n+1)} = \mathbf{D}^{-1}[\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{u}^{(n)}]$$

or

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}_{n+1} = \begin{pmatrix} 1/5 & 0 & 0 \\ 0 & 1/4 & 0 \\ 0 & 0 & -1/7 \end{pmatrix} \left[\begin{pmatrix} 5 \\ -10 \\ -9 \end{pmatrix} - \begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & 0 \\ 2 & 2 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}_n \right].$$

If we start from the initial guess $\mathbf{u}^{(0)} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$, we have

$$\mathbf{u}^{(1)} \approx \begin{pmatrix} 1 \\ -2.5 \\ 1.2857 \end{pmatrix}, \mathbf{u}^{(2)} \approx \begin{pmatrix} 2.0143 \\ -2.7500 \\ 0.8571 \end{pmatrix}, \mathbf{u}^{(3)} \approx \begin{pmatrix} 1.8929 \\ 3.0036 \\ 1.0755 \end{pmatrix},$$

$$\mathbf{u}^{(4)} \approx \begin{pmatrix} 2.0309 \\ -2.9732 \\ 0.9684 \end{pmatrix}, \quad \mathbf{u}^{(5)} \approx \begin{pmatrix} 1.9820 \\ -3.0077 \\ 1.0165 \end{pmatrix}.$$

We can see that after 5 iterations, the approximate solution is quite near the true solution $\mathbf{u} = \begin{pmatrix} 2 & -3 & 1 \end{pmatrix}^T$.

This is an issue here. If we interchange the second row (equation) and the third row (equation), then the new diagonal matrix is

$$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

which has no inverse as it is singular. This means the order of the equations is important to ensure that the matrix is diagonally dominant.

Furthermore, if we interchange the first equation (row) and second equation (row), we have an equivalent system

$$\begin{pmatrix} 1 & 4 & 0 \\ 5 & 1 & -2 \\ 2 & 2 & -7 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} -10 \\ 5 \\ -9 \end{pmatrix}.$$

Now the new decomposition becomes

$$\mathbf{A} = \begin{pmatrix} 1 & 4 & 0 \\ 5 & 1 & -2 \\ 2 & 2 & -7 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -7 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 5 & 0 & 0 \\ 2 & 2 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 4 & 0 \\ 0 & 0 & -2 \\ 0 & 0 & 0 \end{pmatrix},$$

which gives the following iteration formula

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}_{n+1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{7} \end{pmatrix} \left[\begin{pmatrix} -10 \\ 5 \\ -9 \end{pmatrix} - \begin{pmatrix} 0 & 4 & 0 \\ 5 & 0 & -2 \\ 2 & 2 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}_n \right].$$

Starting from $\mathbf{u}^{(0)} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ again, we have

$$\mathbf{u}^{(1)} = \begin{pmatrix} -10 \\ 5 \\ 1.2857 \end{pmatrix},\ \mathbf{u}^{(2)} = \begin{pmatrix} -30 \\ 57.5714 \\ -0.1429 \end{pmatrix},\ \mathbf{u}^{(3)} = \begin{pmatrix} -240.28 \\ 154.71 \\ 9.16 \end{pmatrix},\ ...$$

We can see that it diverges. So what is the problem? How can the order of the equation affect the results so significantly?

There are two important criteria for the iteration to converge correctly, and they are: the inverse of $\mathbf{D}^{-1}$ must exist and the spectral radius of the right matrix must be less than 1. The first condition is obvious, if $\mathbf{D}^{-1}$ does not exist (say, when any of the diagonal elements is zero), then we cannot carry out the iteration process at all. The second condition requires

$$\rho(\mathbf{D}^{-1}) \leq 1, \qquad \rho[\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})] \leq 1, \tag{4.40}$$

where $\rho(\mathbf{A})$ is the spectral radius of the matrix $\mathbf{A}$. From the diagonal matrix $\mathbf{D}$, its largest absolute eigenvalue is 1. So $\rho(\mathbf{D}^{-1}) = \max(|\lambda_i|) = 1$ seems to be no problem. How about the following matrix?

$$\mathbf{N} = \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U}) = \begin{pmatrix} 0 & 4 & 0 \\ 5 & 0 & -2 \\ -2/7 & -2/7 & 0 \end{pmatrix}. \tag{4.41}$$

The three eigenvalues of $\mathbf{N}$ are $\lambda_i = 4.590, -4.479, -0.111$. So its spectral radius is $\rho(\mathbf{N}) = \max(|\lambda_i|) = 4.59 > 1$. The iteration scheme will diverge.

If we revisit our earlier example, we have

$$\mathbf{D}^{-1} = \begin{pmatrix} 1/5 & 0 & 0 \\ 0 & 1/4 & 0 \\ 0 & 0 & -1/7 \end{pmatrix},\ \text{eig}(\mathbf{D}^{-1}) = \frac{1}{5}, \frac{1}{4}, \frac{-1}{7}, \tag{4.42}$$

and

$$\mathbf{N} = \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U}) = \begin{pmatrix} 0 & 1/5 & -2/5 \\ 1/4 & 0 & 0 \\ -2/7 & -2/7 & 0 \end{pmatrix}, \tag{4.43}$$

whose eigenvalues are

$$\lambda_1 = 0.4739, \qquad \lambda_{2,3} = -0.2369 \pm 0.0644i. \tag{4.44}$$

So we have

$$\rho(\mathbf{D}^{-1}) = 1/4 < 1, \qquad \rho(\mathbf{N}) = 0.4739 < 1. \tag{4.45}$$

That is why the earlier iteration procedure is convergent.

4.5.2 Gauss-Seidel Iteration

In the Jacobi-type iterations, we have to store both $\mathbf{u}^{(n+1)}$ and $\mathbf{u}^{(n)}$ as we have to use all the $\mathbf{u}_j^{(n)}$ values to compute the value at the next level $t = n + 1$, this means that we cannot use the running update when the new approximate has just been computed

$$\mathbf{u}_j^{(n+1)} \rightarrow \mathbf{u}_j^{(n)}, \qquad (j = 1, 2, ...).$$

If the vector size $\mathbf{u}$ is large (it usually is), then we can devise other iteration procedure to save memory using the running update. So only one vector storage is needed.

The Gauss-Seidel iteration procedure is such an iteration procedure to use the running update and also provides an efficient way of solving the linear matrix equation $\mathbf{Au} = \mathbf{b}$. It uses the same decomposition as the Jacobi-type iteration by splitting $\mathbf{A}$ into

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}, \tag{4.46}$$

but the difference from the Jacobi method is that we use $\mathbf{L} + \mathbf{D}$ instead of $\mathbf{D}$ for the inverse so that the running update is possible. The n-th step iteration is updated by

$$(\mathbf{L} + \mathbf{D})\mathbf{u}^{(n+1)} = \mathbf{b} - \mathbf{U}\mathbf{u}^{(n)}, \tag{4.47}$$

or

$$\mathbf{u}^{(n+1)} = (\mathbf{L} + \mathbf{D})^{-1}[\mathbf{b} - \mathbf{U}\mathbf{u}^{(n)}]. \tag{4.48}$$

This procedure, starting from an initial vector $\mathbf{u}^{(0)}$, stops if a prescribed criterion is reached.

It is worth pointing out that the Gauss-Seidel iteration requires the same criteria of convergence as the Jacobi-type iteration method. The inverse of the matrix must exist, and the largest spectral radius must be less than 1.

4.5.3 Relaxation Method

The above Gauss-Seidel iteration method is still slow, and the relaxation method provides a more efficient iteration procedure. A popular method is the successive over-relaxation method which consists of two steps

$$\mathbf{v}^{(n)} = (\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{u}^{(n)} - \mathbf{b}, \tag{4.49}$$

and

$$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} - \omega(\mathbf{L} + \mathbf{D})^{-1}\mathbf{v}^{(n)}, \tag{4.50}$$

where $0 < \omega < 2$ is the overrelaxation parameter. If we combine the above equations and re-arrange, we have

$$\mathbf{u}^{(n+1)} = (1 - \omega)\mathbf{u}^{(n)} + \omega\tilde{\mathbf{u}}^{(n)}, \tag{4.51}$$

where $\tilde{\mathbf{u}}^{(n)} = (\mathbf{L} + \mathbf{D})^{-1}(\mathbf{b} - \mathbf{U}\mathbf{u}^{(n)})$ is the standard Gauss-Seidel procedure. Therefore, this method is essentially the weighted average between the previous ieration and the successive Gauss-Seidel iteration. Clearly, if $\omega = 1$, then it reduces to the standard Gauss-Seidel iteration method.

Broadly speaking, a small value of $0 < \omega < 1$ corresponds to under-relaxation with slower convergence while $1 < \omega < 2$ leads to over-relaxation and faster convergence. It has been proved theoretically that the scheme will not converge if $\omega < 0$ or $\omega > 2$.

4.6 Nonlinear Equation

Sometimes, the algebraic equations we meet are nonlinear, and direct inversion is not the best technique. In this case, more elaborate techniques should be used.

4.6.1 Simple Iterations

The nonlinear algebraic equation

$$\mathbf{A}(\mathbf{u})\mathbf{u} = \mathbf{b}(\mathbf{u}), \qquad \text{or} \qquad \mathbf{F}(\mathbf{u}) = \mathbf{A}(\mathbf{u})\mathbf{u} - \mathbf{b}(\mathbf{u}) = \mathbf{0}, \quad (4.52)$$

can be solved using a simple iteration technique

$$\mathbf{A}(\mathbf{u}^{(n)})\mathbf{u}^{(n+1)} = \mathbf{b}(\mathbf{u}^{(n)}), \qquad n = 0, 1, 2, ... \quad (4.53)$$

until $\|\mathbf{u}^{(n+1)} - \mathbf{u}^{(n)}\|$ is sufficiently small. Iterations require a starting vector $\mathbf{u}^{(0)}$. This method is also referred to as the successive substitution.

If this simple method does not work, the relaxation method can be used. The relaxation technique first gives a tentative new approximation $\mathbf{u}^*$ from $\mathbf{A}(\mathbf{u}^{(n)})\mathbf{u}^* = \mathbf{b}(\mathbf{u}^{(n)})$, then we use

$$\mathbf{u}^{(n+1)} = \omega\mathbf{u}^* + (1 - \omega)\mathbf{u}^{(n)}, \qquad \omega \in (0, 1], \quad (4.54)$$

where ω is a prescribed relaxation parameter.

4.6.2 Newton-Raphson Method

The nonlinear equation (4.52) can also be solved using the Newton-Raphson procedure. We approximate $\mathbf{F}(\mathbf{u})$ by a linear function $\mathbf{R}(\mathbf{u}; \mathbf{u}^{(n)})$ in the vicinity of an existing approximation $\mathbf{u}^{(n)}$ to $\mathbf{u}$:

$$\mathbf{R}(\mathbf{u}; \mathbf{u}^{(n)}) = \mathbf{F}(\mathbf{u}^{(n)}) + \mathbf{J}(\mathbf{u}^{(n)})(\mathbf{u} - \mathbf{u}^{(n)}), \quad \mathbf{J}(\mathbf{u}) = \nabla\mathbf{F}, \quad (4.55)$$

where $\mathbf{J}$ is the Jacobian of $\mathbf{F}(\mathbf{u}) = (F_1, F_2, ..., F_M)^T$. For $\mathbf{u} = (u_1, u_2, ..., u_M)^T$, we have

$$\mathbf{J}_{ij} = \frac{\partial F_i}{\partial u_j}. \quad (4.56)$$

To find the next approximation $\mathbf{u}^{(n+1)}$ from $\mathbf{R}(\mathbf{u}^{(n+1)}; \mathbf{u}^{(n)}) = 0$, one has to solve a linear system with $\mathbf{J}$ as the coefficient matrix

$$\mathbf{u}^{(n+1)} = \mathbf{u}^{(n)} - \mathbf{J}^{-1}\mathbf{F}(\mathbf{u}^{(n)}), \quad (4.57)$$

under a given termination criterion $\|\mathbf{u}^{(n+1)} - \mathbf{u}^{(n)}\| \leq \epsilon$.

Part II

Mathematical Optimization

Chapter 5

Unconstrained Optimization

5.1 Univariate Functions

The simplest optimization problem without any constraints is probably the search of the maxima or minima of a univariate function $f(x)$. For unconstrained optimization problems, the optimality occurs at the critical points given by the stationary condition $f'(x) = 0$. However, this stationary condition is just a necessary condition, but it is not sufficient. If $f'(x_*) = 0$ and $f''(x_*) > 0$, it is a local minimum. Conversely, if $f'(x_*) = 0$ and $f''(x_*) < 0$, then it is a local maximum. However, if $f'(x_*) = 0$ but $f''(x)$ is indefinite (both positive and negative) when $x \to x_*$, then x_* corresponds to a saddle point. For example, $f(x) = x^3$ has a saddle point $x_* = 0$ because $f'(0) = 0$ but f'' changes sign from $f''(0+) > 0$ to $f''(0-) < 0$.

Example 5.1: For example, in order to find the maximum or minimum of an univariate function $f(x)$

$$f(x) = xe^{-x^2}, \qquad -\infty < x < \infty, \tag{5.1}$$

we have to find first the stationary point x_* when the first deriva-

tive $f'(x)$ is zero. That is

$$\frac{df(x_*)}{dx_*} = e^{-x_*^2} - 2x_*^2 e^{-x_*^2} = 0. \tag{5.2}$$

Since $\exp(-x_*^2) \neq 0$, we have

$$x_* = \pm\frac{\sqrt{2}}{2}. \tag{5.3}$$

From the basic calculus we know that the maximum requires $f''(x_*) \leq 0$ while minimum requires $f''(x_*) \geq 0$. At $x_* = \sqrt{2}/2$, we have

$$f''(x_*) = (4x_*^2 - 6)x_* e^{-x_*^2} = -2\sqrt{2}e^{-1/2} < 0, \tag{5.4}$$

so this point corresponds to a maximum $f(x_*) = \frac{1}{2}e^{-1/2}$. Similarly, $x_* = -\sqrt{2}/2$, $f''(x_*) = 2\sqrt{2}e^{-1/2} > 0$, we have a minimum $f(x_*) = -\frac{1}{2}e^{-1/2}$.

Since a maximum of a function $f(x)$ can be converted into a minimum of $A - f(x)$ where A is usually a large positive number (though $A = 0$ will do). For example, we know the maximum of $f(x) = e^{-x^2}, x \in (-\infty, \infty)$ is 1 at $x_* = 0$. This problem can be converted to a minimum problem $-f(x)$. For this reason, the optimization problems can be either expressed as minima or maxima depending the context and convenience of finding the solutions.

5.2 Multivariate Functions

For functions of multivariate $\boldsymbol{x} = (x_1, ..., x_n)^T$, the optimization can be expressed in the same way as the univariate optimization problems.

$$\underset{\boldsymbol{x}\in\Re^n}{\text{minimize/maximize}} f(\boldsymbol{x}). \tag{5.5}$$

For a quadratic function $f(\boldsymbol{x})$ it can be expanded using Taylor series about a point $\boldsymbol{x} = \boldsymbol{x}_*$ so that $\boldsymbol{x} = \boldsymbol{x}_* + \epsilon\boldsymbol{u}$

$$f(\boldsymbol{x}+\epsilon\boldsymbol{u}) = f(\boldsymbol{x}_*)+\epsilon\boldsymbol{u}\boldsymbol{G}_1(\boldsymbol{x}_*)+\frac{1}{2}\epsilon^2\boldsymbol{u}^T\boldsymbol{G}_2(\boldsymbol{x}_*+\epsilon\boldsymbol{u})\boldsymbol{u}+..., \tag{5.6}$$

where $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ are the gradient vector and the Hessian matrix, respectively. ϵ is a small parameter, and $\boldsymbol{u}$ is a vector. For example, $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} + \boldsymbol{k}^T\boldsymbol{x} + \boldsymbol{b}$, we have

$$f(\boldsymbol{x}_* + \epsilon\boldsymbol{u}) = f(\boldsymbol{x}_*) + \epsilon\boldsymbol{u}^T\boldsymbol{k} + \frac{1}{2}\epsilon^2\boldsymbol{u}^T\boldsymbol{A}\boldsymbol{u} + ..., \tag{5.7}$$

where

$$f(\boldsymbol{x}_*) = \frac{1}{2}\boldsymbol{x}_*^T\boldsymbol{A}\boldsymbol{x}_* + \boldsymbol{k}^T\boldsymbol{x}_* + \boldsymbol{b}. \tag{5.8}$$

Thus, in order to study the local behaviour of a quadratic function, we only need to study $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$. In addition, for simplicity, we can take $\boldsymbol{b} = \boldsymbol{0}$ as it is a constant vector anyway.

At a stationary point $\boldsymbol{x}_*$, the first derivatives are zero or $\boldsymbol{G}_1(\boldsymbol{x}_*) = 0$, therefore, equation (5.6 becomes

$$f(\boldsymbol{x}_* + \epsilon\boldsymbol{u}) \approx f(\boldsymbol{x}_*) + \frac{1}{2}\epsilon^2\boldsymbol{u}^T\boldsymbol{G}_2\boldsymbol{u}. \tag{5.9}$$

If $\boldsymbol{G}_2 = \boldsymbol{A}$, then

$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v} \tag{5.10}$$

forms an eigenvalue problem. For an $n \times n$ matrix $\boldsymbol{A}$, there will be n eigenvalues $\lambda_j (j = 1, ..., n)$ with n eigenvectors $\boldsymbol{v}$. As we have seen earlier that $\boldsymbol{A}$ is symmetric, these eigenvectors are orthonormal. That is,

$$\boldsymbol{v}_i^T\boldsymbol{v}_j = \delta_{ij}. \tag{5.11}$$

Near any stationary point $\boldsymbol{x}_*$, if we take $\boldsymbol{u}_j = \boldsymbol{v}_j$ as the local coordinate systems, we then have

$$f(\boldsymbol{x}_* + \epsilon\boldsymbol{v}_j) = f(\boldsymbol{x}_*) + \frac{1}{2}\epsilon^2\lambda_j, \tag{5.12}$$

which means that the variations of $f(\boldsymbol{x})$, when $\boldsymbol{x}$ moves away from the stationary point $\boldsymbol{x}_*$ along the direction $\boldsymbol{v}_j$, are characterised by the eigenvalues. If $\lambda_j > 0$, $|\epsilon| > 0$ will leads to $|\Delta f| = |f(\boldsymbol{x}) - f(\boldsymbol{x}_*)| > 0$. In other words, $f(\boldsymbol{x})$ will increase as $|\epsilon|$ increases. Conversely, if $\lambda_j < 0$, $f(\boldsymbol{x})$ will decrease as $|\epsilon| > 0$ increases. Obviously, in the special case $\lambda_j = 0$, the

function $f(\boldsymbol{x})$ will remain constant along the corresponding direction of $\boldsymbol{v}_j$.

Example 5.2: We know that the function $f(x, y) = xy$ has a saddle point at $(0, 0)$. It increases along the $x = y$ direction and decreases along $x = -y$ direction. From the above analysis, we know that $\boldsymbol{x}_* = (x_*, y_*)^T = (0, 0)^T$ and $f(x_*, y_*) = 0$. We now have

$$f(\boldsymbol{x}_* + \epsilon \boldsymbol{u}) \approx f(\boldsymbol{x}_*) + \frac{1}{2}\epsilon^2 \boldsymbol{u}^T \boldsymbol{A} \boldsymbol{u},$$

where

$$\boldsymbol{A} = \nabla^2 f(\boldsymbol{x}_*) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The eigenvalue problem is simply

$$\boldsymbol{A} v = \lambda_j \boldsymbol{v}_j, \quad (j = 1, 2),$$

or

$$\begin{vmatrix} -\lambda_j & 1 \\ 1 & -\lambda_j \end{vmatrix} = 0,$$

whose solutions are

$$\lambda_j = \pm 1.$$

For $\lambda_1 = 1$, the corresponding eigenvector is

$$\boldsymbol{v}_1 = \begin{pmatrix} \sqrt{2}/2 \\ \sqrt{2}/2 \end{pmatrix}.$$

Similarly, for $\lambda_2 = -1$, the eigenvector is

$$\boldsymbol{v}_2 = \begin{pmatrix} \sqrt{2}/2 \\ -\sqrt{2}/2 \end{pmatrix}.$$

Since A is symmetric, $\boldsymbol{v}_1$ and $\boldsymbol{v_2}$ are orthonormal. Indeed this is the case because $\|\boldsymbol{v}_1\| = \|\boldsymbol{v}_2\| = 1$ and

$$\boldsymbol{v}_1^T \boldsymbol{v}_2 = \frac{\sqrt{2}}{2} \times \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2} \times (-\frac{\sqrt{2}}{2}) = 0.$$

Thus, we have

$$f(\epsilon \boldsymbol{v}_j) = \frac{1}{2}\epsilon^2 \lambda_j, \qquad (j = 1, 2). \tag{5.13}$$

As $\lambda_1 = 1$ is positive, f increases along the direction $\boldsymbol{v}_1 = \frac{\sqrt{2}}{2}(1\ 1)^T$ which is indeed along the line $x = y$. Similarly, for $\lambda_2 = -1$, f will decrease along $\boldsymbol{v}_2 = \frac{\sqrt{2}}{2}(1\ -1)^T$ which is exactly along the line $x = -y$. As there is no zero eigenvalue, the function will not remain constant in the region around $(0, 0)$.

5.3 Gradient-Based Methods

The gradient-based methods are iterative methods that extensively use the information of the gradient of the function during iterations. For the minimization of a function $f(\boldsymbol{x})$, the essence of this class of method is

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \alpha g(\nabla f, \boldsymbol{x}^{(n)}), \tag{5.14}$$

where α is the step size which can vary during iterations. $g(\nabla f)$ is a function of the gradient ∇f. Different methods use different form of $g(\nabla f, \boldsymbol{x}^{(n)})$.

5.3.1 Newton's Method

We know that Newton's method is a popular iterative method for finding the zeros of a nonlinear univariate function of $f(x)$ on the interval $[a, b]$. It can be modified for solving optimization problems because it is equivalent to find the zero of the first derivative $f'(\boldsymbol{x})$ once the objective function $f(\boldsymbol{x})$ is given.

For a given function $f(\boldsymbol{x})$ which are continuously differentiable, we have the Taylor expansion about a known point $\boldsymbol{x} = \boldsymbol{x}_n$ (with $\Delta \boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_n$)

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_n) + (\nabla f(\boldsymbol{x}_n))^T \Delta \boldsymbol{x} + \frac{1}{2}\Delta \boldsymbol{x}^T \nabla^2 f(\boldsymbol{x}_n) \Delta \boldsymbol{x} + ..., \tag{5.15}$$

which is minimized near a critical point when $\Delta \boldsymbol{x}$ is the solution of the following linear equation

$$\nabla f(\boldsymbol{x}_n) + \nabla^2 f(\boldsymbol{x}_n)\Delta \boldsymbol{x} = 0. \tag{5.16}$$

This leads to

$$\boldsymbol{x} = \boldsymbol{x}_n - \boldsymbol{G}^{-1}\nabla f(\boldsymbol{x}_n), \tag{5.17}$$

where $\boldsymbol{G} = \nabla^2 f(\boldsymbol{x}_n)$ is the Hessian matrix. If the iteration procedure starts from the initial vector $\boldsymbol{x}^{(0)}$ (usually taken to be a guessed point in the domain), then Newton's iteration formula for the nth iteration is

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - \boldsymbol{G}^{-1}(\boldsymbol{x}^{(n)})f(\boldsymbol{x}^{(n)}). \tag{5.18}$$

It is worth pointing out that if $f(\boldsymbol{x})$ is quadratic, then the solution can exactly be found in a single step. However, this method is not efficient for non-quadratic functions.

In order to speed up the convergence, we can use a smaller step size $\alpha \in (0, 1]$ so that we have modified Newton's method

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - \alpha \boldsymbol{G}^{-1}(\boldsymbol{x}^{(n)})f(\boldsymbol{x}^{(n)}). \tag{5.19}$$

It sometimes might be time-consuming to calculate the Hessian matrix for second derivatives. A good alternative is to use an identity matrix $\boldsymbol{G}^{-1} = \boldsymbol{I}$, and we have the quasi-Newton method

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - \alpha \boldsymbol{I}\nabla f(\boldsymbol{x}^{(n)}), \tag{5.20}$$

which is essentially the steepest descent method.

5.3.2 Steepest Descent Method

The essence of this method is to find the lowest possible objective function $f(\boldsymbol{x})$ from the current point $\boldsymbol{x}^{(n)}$. From the Taylor expansion of $f(\boldsymbol{x})$ about $\boldsymbol{x}^{(n)}$, we have

$$f(\boldsymbol{x}^{(n+1)}) = f(\boldsymbol{x}^{(n)} + \Delta \boldsymbol{s}) \approx f(\boldsymbol{x}^{(n)} + (\nabla f(\boldsymbol{x}^{(n)}))^T \Delta \boldsymbol{s}, \tag{5.21}$$

where $\Delta \boldsymbol{s} = \boldsymbol{x}^{(n+1)} - \boldsymbol{x}^{(n)}$ is the increment vector. Since we try to find a lower (better) approximation to the objective function,

it requires that the second term on the right hand is negative. That is

$$f(\boldsymbol{x}^{(n)} + \Delta \boldsymbol{s}) - f(\boldsymbol{x}^{(n)}) = (\nabla f)^T \Delta \boldsymbol{s} < 0. \tag{5.22}$$

From vector analysis, we know the inner product $\boldsymbol{u}^T \boldsymbol{v}$ of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ is largest when they are parallel but in opposite directions. Therefore, $(\nabla f)^T \Delta \boldsymbol{s}$ becomes the smallest when

$$\Delta \boldsymbol{s} = -\alpha \nabla f(\boldsymbol{x}^{(n)}), \tag{5.23}$$

where $\alpha > 0$ is the step size. This the case when the direction $\Delta \boldsymbol{s}$ is along the steepest descent in the negative gradient direction. As we seen earlier, this method is a quasi-Newton method.

The choice of the step size α is very important. A very small step size means slow movement towards the local minimum, while a large step may overshoot and subsequently makes it move far away from the local minimum. Therefore, the step size $\alpha = \alpha^{(n)}$ should be different at each iteration step and should be chosen so that it minimizes the objective function $f(\boldsymbol{x}^{(n+1)}) = f(\boldsymbol{x}^{(n)}, \alpha^{(n)})$. Therefore, the steepest descent method can be written as

$$f(\boldsymbol{x}^{(n+1)}) = f(\boldsymbol{x}^{(n)}) - \alpha^{(n)} (\nabla f(\boldsymbol{x}^{(n)}))^T \nabla f(\boldsymbol{x}^{(n)}). \tag{5.24}$$

In each iteration, the gradient and the step size will be calculated. Again, a good initial guess of both the starting point and the step size is useful.

Example 5.3: Let us minimize the function

$$f(x_1, x_2) = 10x_1^2 + 5x_1 x_2 + 10(x_2 - 3)^2,$$

where

$$(x_1, x_2) = [-10, 10] \times [-15, 15],$$

using the steepest descent method starting with the initial $\boldsymbol{x}^{(0)} = (10, 15)^T$. We know that the gradient

$$\nabla f = (20x_1 + 5x_2, 5x_1 + 20x_2 - 60)^T,$$

therefore

$$\nabla f(\boldsymbol{x}^{(0)}) = (275, 290)^T.$$

In the first iteration, we have

$$\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(0)} - \alpha_0 \begin{pmatrix} 275 \\ 290 \end{pmatrix}.$$

The step size α_0 should be chosen such that $f(\boldsymbol{x}^{(1)})$ is at the minimum, which means that

$$f(\alpha_0) = 10(10 - 275\alpha_0)^2$$

$$+5(10 - 275\alpha_0)(15 - 290\alpha_0) + 10(12 - 290\alpha_0)^2,$$

should be minimized. This becomes an optimization problem for a single independent variable α_0. All the techniques for univariate optimization problems such as Newton's method can be used to find α_0. We can also obtain the solution by setting

$$\frac{df}{d\alpha_0} = -159725 + 3992000\alpha_0 = 0,$$

whose solution is $\alpha_0 \approx 0.04001$.

At the second step, we have

$$\nabla f(\boldsymbol{x}^{(1)}) = (-3.078, 2.919)^T, \quad \boldsymbol{x}^{(2)} = \boldsymbol{x}^{(1)} - \alpha_1 \begin{pmatrix} -3.078 \\ 2.919 \end{pmatrix}.$$

The minimization of $f(\alpha_1)$ gives $\alpha_1 \approx 0.066$, and the new location of the steepest descent is

$$\boldsymbol{x}^{(2)} \approx (-0.797, 3.202)^T.$$

At the third iteration, we have

$$\nabla f(\boldsymbol{x}^{(2)}) = (0.060, 0.064)^T, \quad \boldsymbol{x}^{(3)} = \boldsymbol{x}^{(2)} - \alpha_2 \begin{pmatrix} 0.060 \\ 0.064 \end{pmatrix}.$$

The minimization of $f(\alpha_2)$ leads to $\alpha_2 \approx 0.040$, and thus

$$\boldsymbol{x}^{(3)} \approx (-0.8000299, 3.20029)^T.$$

Then, the iterations continue until a prescribed tolerance is met.

From calculus, we know that we can set the first partial derivatives equal to zero

$$\frac{\partial f}{\partial x_1} = 20x_1 + 5x_2 = 0, \quad \frac{\partial f}{\partial x_2} = 5x_1 + 20x_2 - 60 = 0,$$

we know that the minimum occurs exactly at

$$\boldsymbol{x}_* = (-4/5, 16/5)^T = (-0.8, 3.2)^T.$$

The steepest descent method gives almost the exact solution after only 3 iterations.

In finding the step size α_n in the above steepest descent method, we have used the stationary condition $df(\alpha_n)/d\alpha_n = 0$. Well, you may say that if we use this stationary condition for $f(\alpha_0)$, why not use the same method to get the minimum point of $f(\boldsymbol{x})$ in the first place. There are two reasons here. The first reason is that this is a simple example for demonstrating how the steepest descent method works. The second reason is that even for complicated multiple variable $f(x_1, ..., x_p)$ (say $p = 500$), then $f(\alpha_n)$ at any step n is still an univariate function, and the optimization of such $f(\alpha_n)$ is much simpler compared with the original multivariate problem.

From our example, we know that the convergence from the second iteration to the third iteration is slow. In fact, the steepest descent is typically slow once the local minimization is near. This is because near the local minimization the gradient is nearly zero, and thus the rate of descent is also slow. If high accuracy is need near the local minimum, other local search methods should be used.

It is worth pointing out that there are many variations of the steepest descent methods. If the optimization is to find the maximum, then this method becomes the hill-climbing method because the aim is to climb up the hill to the highest peak.

5.4 Hooke-Jeeves Pattern Search

Many search algorithms such as the steepest descent method experience slow convergence near the local minimum. They are also memoryless because the pass information is not used to produce accelerated move. The only information they use is the current location $\boldsymbol{x}^{(n)}$, gradient and value of the objective itself at step n. If the past information such as the steps at $n-1$ and n is properly used to generate new move at step $n+1$, it may speed up the convergence. The Hooke-Jeeves pattern search method is one of such methods that incorporate the past history of iterations in producing a new search direction.

The Hooke-Jeeves pattern search method consists of two moves: exploratory move and pattern move. The exploratory moves explore the local behaviour and information of the objective function so as to identify any potential sloping valleys if they exist. For any given step size (each coordinate direction can have different increment) $\Delta_i (i = 1, 2, ..., p)$, exploration movement performs from an initial starting point along each coordinate direction by increasing or decreasing $\pm\Delta_i$, if the new value of the objective function does not increase (for minimization problem), that is $f(x_i^{(n)}) \leq f(x_i^{(n-1)})$, the exploratory move is considered as successful. If it is not successful, then try a step in the opposite direction, and the result is updated only if it is successful. When all the p coordinates have been explored, the resulting point forms a base point $\boldsymbol{x}^{(n)}$.

The pattern move intends to move the current base $\boldsymbol{x}^{(n)}$ along the base line $(\boldsymbol{x}^{(n)} - \boldsymbol{x}^{(n-1)})$ from the previous (historical) base point to the current base point. The move is carried out by the following formula

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + [\boldsymbol{x}^{(n)} - \boldsymbol{x}^{(n-1)}]. \tag{5.25}$$

Then $\boldsymbol{x}^{(n+1)}$ forms a new temporary base point for further new exploratory moves. If the pattern move produces improvement (lower value of $f(\boldsymbol{x})$), the new base point $\boldsymbol{x}^{(n+1)}$ is successfully updated. If the pattern move does not lead to improvement or lower value of the objective function, then the pattern move

Hooke-Jeeves Pattern Search Algorithm

begin
Objective function $f(\boldsymbol{x}), \boldsymbol{x} = (x_1, ..., x_p)^T$
Initialize starting point $\boldsymbol{x}^{(0)}$ *and increments* $\Delta_i (i = 1, ..., p)$
Initialize step reduction factor $\gamma > 1$ *and tolerance* $\epsilon > 0$
while ($||\boldsymbol{x}_{n+1} - \boldsymbol{x}_n|| \geq \epsilon$)
for *all* $i = 1$ *to* p,
Perform exploratory search by $x_i \pm \Delta_i$
Update until successful $f(\boldsymbol{x}^{(n+1)}) \leq f(\boldsymbol{x}^{(n)})$
If the move fails, try again using $\Delta_i = \Delta_i / \gamma$
end *(for)*
Perform pattern move:
$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + (\boldsymbol{x}^{(n)} - \boldsymbol{x}^{(n-1)})$
Update new base point $\boldsymbol{x}^{(n+1)}$
$n = n + 1$
end *(while)*
end

Figure 5.1: Hooke-Jeeves pattern search algorithm.

is discarded and new search starts from $\boldsymbol{x}^{(n)}$, and new search moves should use smaller step size by reducing increments D_i/γ when $\gamma > 1$ is the step reduction factor. Iterations continue until the prescribed tolerance ϵ is met. The algorithm is summarised in the pseudo code shown in Fig. 5.1.

Example 5.4: Let us minimize

$$f(\boldsymbol{x}) = x_1^2 + 2x_1x_2 + 3x_2^2,$$

using the Hooke-Jeeves pattern search. Suppose we start from $\boldsymbol{x}^{(0)} = (2, 2)^T$ with an initial step size $\Delta = 1$ for both coordinates x_1 and x_2. We know that $f(\boldsymbol{x}^{(0)}) = 24$.

First, the exploratory move in x_1 with $\boldsymbol{x}^{(1)} = (2 + 1, 2)^T$ produces $f(\boldsymbol{x}^{(1)}) = 33 > 24$. So this move is not successful, we discard it, and try the opposite direction $\boldsymbol{x}^{(1)} = (2 - 1, 2)^T$. It gives $f(\boldsymbol{x}^{(1)}) = 17 < 24$, and it is a good move. So we keep it and there is no need to reduce the step size $\Delta = 1$. Now we try

the other coordinate along x_2, we know that $\boldsymbol{x}^{(1)} = (1, 2-1)^T$ is a good move as it gives $f(\boldsymbol{x}^{(1)}) = 6 < 24$. Therefore, the base point is $\boldsymbol{x}^{(1)} = (1, 1)^T$.

We then perform the pattern move by using $\boldsymbol{x}^{(2)} = \boldsymbol{x}^{(1)} + (\boldsymbol{x}^{(1)} - \boldsymbol{x}^{(0)})$, we have

$$\boldsymbol{x}^{(2)} = 2\begin{pmatrix}1\\1\end{pmatrix} - \begin{pmatrix}2\\2\end{pmatrix} = \begin{pmatrix}0\\0\end{pmatrix}.$$

This pattern move produces $f(\boldsymbol{x}^{(2)}) = 0$ and it is a successful move. This is indeed the optimal solution as the minimum occurs exactly at $\boldsymbol{x}_* = (0, 0)^T$.

Chapter 6

Linear Mathematical Programming

6.1 Linear Programming

Linear programming is a powerful mathematical modelling technique which is widely used in business planning, engineering design, oil industry, and many other optimization applications. The basic idea in linear programming is to find the maximum or minimum under linear constraints.

For example, an Internet service provider (ISP) can provide two different services x_1 and x_2. The first service is, say, the fixed monthly rate with limited download limits and bandwidth, while the second service is the higher rate with no download limit. The profit of the first service is αx_1 while the second is βx_2, though the profit of the second product is higher $\beta > \alpha > 0$, so the total profit is

$$P(\boldsymbol{x}) = \alpha x_1 + \beta x_2, \qquad \beta/\alpha > 1, \tag{6.1}$$

which is the objective function because the aim of the ISP company is to increase the profit as much as possible. Suppose the provided service is limited by the total bandwidth of the ISP company, thus at most $n_1 = 16$ (in 1000 units) of the first and at most $n_2 = 10$ (in 1000 units) of the second can be

provided per unit of time, say, each day. Therefore, we have

$$x_1 \le n_1, \qquad x_2 \le n_2. \tag{6.2}$$

If the management of each of the two service packages take the same staff time, so that a maximum of $n = 20$ (in 1000 units) can be maintained, which means

$$x_1 + x_2 \le n. \tag{6.3}$$

The additional constraints are that both x_1 and x_2 must be non-negative since negative numbers are unrealistic. We now have the following constraints

$$0 \le x_1 \le n_1, \qquad 0 \le x_2 \le n_2. \tag{6.4}$$

The problem now is to find the best x_1 and x_2 so that the profit P is a maximum. Mathematically, we have

$$\begin{aligned} \underset{(x_1,x_2)\in\mathcal{N}^2}{\text{maximize}} \quad & P(x_1, x_2) = \alpha x_1 + \beta x_2, \\ \text{subject to} \quad & x_1 + x_2 \le n, \\ & 0 \le x_1 \le n_1,\ 0 \le x_2 \le n_2. \end{aligned} \tag{6.5}$$

Example 6.1: The feasible solutions to this problem can be graphically represented as the inside region of the polygon $OABCD$ as shown in Fig. 6.1.

As the aim is to maximize the profit P, thus the optimal solution is at the extreme point B with $(n - n_2, n_2)$ and $P = \alpha(n-n_2)+\beta n_2$. For example, if $\alpha = 2$, $\beta = 3$, $n_1 = 16$, $n_2 = 10$, and $n = 20$, then the optimal solution is $x_1 = n - n_2 = 10$ and $x_2 = n_2 = 10$ with the total profit $P = 2\times(20-10)+3\times 10 = 50$ thousand pounds.

Since the solution (x_1 and x_2) must be integers, an interesting thing is that the solution is independent of β/α if and only if $\beta/\alpha > 1$. However, the profit P does depend on α and β.

The number of feasible solutions are infinite if x_1 and x_2 are real numbers. Even for $x_1, x_2 \in \mathcal{N}$ are integers, the number of

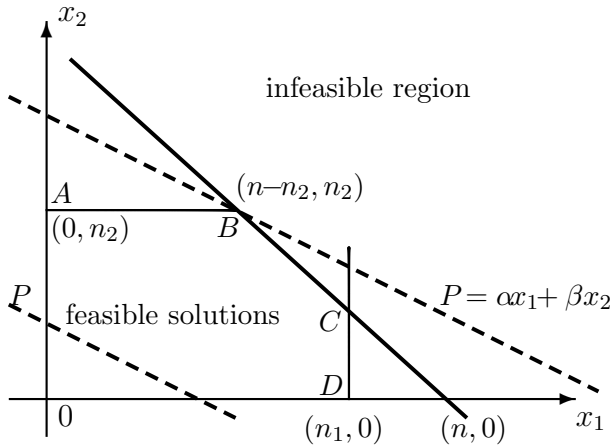


Figure 6.1: Schematic representation of linear programming. If $\alpha = 2$, $\beta = 3$, $n_1 = 16$, $n_2 = 10$ and $n = 20$, then the optimal solution is at $B(10, 10)$.

feasible solutions are quite large. Therefore, there is a need to use a systematic method to find the optimal solution.

In order to find the best solution, we first plot out all the constraints as straight lines, the feasible solution which satisfying all the constraints form the inside region of the polygon $OABCD$. The vertices of the polygon form the set of the extreme points. Then, we plot the objective function P as a family of parallel lines (shown as dashed lines) so as to find the maximum value of P. Obviously, the highest value of P corresponds to the case when the objective line goes through the extreme point B. Therefore, $x_1 = n - n_2$ and $x_2 = n_2$ at the point B is the best solution.

The current example is simple because it has only two decision variables and three constraints, which can be solved easily using a graphic approach. Many real-world problems involve hundreds of or thousands of decisional variables, and graphic approach is simply incapable of dealing with such complexity. Therefore, a formal approach is needed. Many powerful methods have been developed over the last several decades, and the most popular method is probably the simplex method.

6.2 Simplex Method

The simplex method was introduced by George Dantzig in 1947. The simplex method essentially works in the following way: For a given linear optimization problem such as the example of the ISP service we discussed earlier, it assumes that all the extreme points are known. If the extreme points are not known, the first step is to determine these extreme points or to check whether there any feasible solutions. With known extreme points, it is easy to test whether an extreme point is optimal or not using the algebraic relationship and the objective function. If the test for optimality is not passed, then move to an adjacent extreme point to do the same test. This process stops until an optimal extreme point is found or the unbounded case occurs.

6.2.1 Basic Procedure

Mathematically, the simplex method first transforms the constraint inequalities into equalities by using slack variables.

To convert an inequality such as

$$5x_1 + 6x_2 \leq 20, \tag{6.6}$$

we can use a new variable x_3 or $s_1 = 20 - 5x_1 - 6x_2$ so that the original inequality becomes an equality

$$5x_1 + 6x_2 + s_1 = 20, \tag{6.7}$$

with an auxiliary non-negativeness condition

$$s_1 \geq 0. \tag{6.8}$$

Such a variable is referred to as a slack variable.

Thus, the inequalities in our example

$$x_1 + x_2 \leq n, \qquad 0 \leq x_1 \leq n_1, \qquad 0 \leq x_2 \leq n_2, \tag{6.9}$$

can be written, using three slack variables s_1, s_2, s_3, as the following equalities

$$x_1 + x_2 + s_1 = n, \tag{6.10}$$

$$x_1 + s_2 = n_1, \qquad x_2 + s_3 = n_2, \tag{6.11}$$

and

$$x_i \geq 0 \ (i = 1, 2), \ \ s_j \geq 0 (j = 1, 2, 3). \tag{6.12}$$

The original problem (6.5) becomes

$$\underset{\boldsymbol{x} \in \Re^5}{\text{maximize}} P(\boldsymbol{x}) = \alpha x_1 + \beta x_2 + 0 s_1 + 0 s_2 + 0 s_3,$$

$$\text{subject to } \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} n \\ n_1 \\ n_2 \end{pmatrix},$$

$$x_i \geq 0, \qquad (i = 1, 2, ..., 5), \tag{6.13}$$

which has two control variables (x_1, x_2) and three slack variables $x_3 = s_1$, $x_4 = s_2$, $x_5 = s_3$.

In general, a linear programming problem can be written as the standard form

$$\underset{\boldsymbol{x} \in \Re^n}{\text{maximize}} f(\boldsymbol{x}) = Z = \sum_{i=1}^{p} \alpha_i x_i = \boldsymbol{\alpha}^T \boldsymbol{x},$$

$$\text{subject to } \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \ x_i \geq 0 \ (i = 1, ..., p), \tag{6.14}$$

where $\boldsymbol{A}$ is an $q \times p$ matrix, $\boldsymbol{b} = (b_1, ..., b_q)^T$, and

$$\boldsymbol{x} = [\boldsymbol{x}_p \ \ \boldsymbol{x}_{\boldsymbol{s}}]^T = (x_1, ..., x_m, s_1, ..., s_{p-m})^T. \tag{6.15}$$

This problem has p variables, and q equalities and all p variables are non-negative. In the standard form, all constraints are expressed as equalities and all variables including slack variables are non-negative. Suppose $q = 500$, even the simplest integer equalities $x_i + x_j = 1$ where $i, j = 1, 2, ..., 500$, would give a huge number of combinations 2^{500}. Thus the number of basic feasible solutions or extreme points will be the order of $2^{500} \approx 3 \times 10^{150}$, which is larger than the number of particles in the whole universe. This huge number of extreme points necessitates a systematic and efficient search method. The simplex

method is a powerful method to carry out such mathematical programming task.

A basic solution to the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ of q linear equations in p variables in the standard form is usually obtained by setting $p-q$ variables equal to zero, and subsequently solving the resulting $q \times q$ linear system to get a unique solution of the remaining q variables. The q variables (that are not bound to zero) are called the basic variables of the basic solution. The $p - q$ variables at zero are called non-basic variables. Any basic solution to this linear system is referred to as a basic feasible solution (BFS) if all its variables are non-negative. The important property of the basic feasible solutions is that there is a unique corner point (extreme point) for each basic feasible solution, and there is at least one basic feasible solution for each corner or extreme point. These corner or extreme points are points on the intersection of two adjacent boundary lines such as A and B in Fig. 6.1. Two basic feasible solutions are said to be adjacent if they have $q-1$ basic variables in common in the standard form.

6.2.2 Augmented Form

The linear optimization problem is usually converted into the following standard augmented form or the canonical form

$$\begin{pmatrix} 1 & -\boldsymbol{\alpha}^T \\ 0 & \boldsymbol{A} \end{pmatrix} \begin{pmatrix} Z \\ \boldsymbol{x} \end{pmatrix} = \begin{pmatrix} 0 \\ \boldsymbol{b} \end{pmatrix}, \tag{6.16}$$

with the objective to maximize Z. In this canonical form, all the constraints are expressed as equalities for all non-negative variables. All the right hand sides for all constraints are also non-negative, and each constraint equation has a single basic variable. The intention of writing in this canonical form is to identify basic feasible solutions, and move from one basic feasible solution to another via a so-called pivot operation. Geometrically speaking, this means to find all the corner or extreme points first, then evaluate the objective function by going

through the extreme points so as to determine if the current basic feasible solution can be improved or not.

In the framework of the canonical form, the basic steps of the simplex method are: 1) to find a basic feasible solution to start the algorithm. Sometimes, it might be difficult to start, this may either implies there is no feasible solution or it is necessary to reformulate the problem in a slightly different way by changing the canonical form so that a basic feasible solution can be found; 2) to see if the current basic feasible solution can be improved (even marginally) by increasing the non-basic variables from zero to non-negative values; 3) stop the process if the current feasible solution cannot be improved, which means that is optimal. If the current feasible solution is not optimal, then move to an adjacent basic feasible solution. This adjacent basic feasible solution can be obtained by changing the canonical form via elementary row operations.

The pivot manipulations are based on the fact that a linear system will remain an equivalent system by multiplying a non-zero constant on a row and adding it to the other row. This procedure continues by going to the second step and repeating the evaluation of the objective function. The optimality of the problem will be reached or stop the iteration if the solution becomes unbounded in the case that you can improve the objective indefinitely.

6.2.3 A Case Study

Now we come back to our example, if we use $\alpha = 2$, $\beta = 3$, $n_1 = 16$, $n_2 = 10$ and $n = 20$, we then have

$$\begin{pmatrix} 1 & -2 & -3 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Z \\ x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 20 \\ 16 \\ 10 \end{pmatrix}, \tag{6.17}$$

where $x_1, x_2, s_1, ..., s_3 \geq 0$. Now the first step is to identify a corner point or basic feasible solution by setting non-isolated

variables $x_1 = 0$ and $x_2 = 0$ (thus the basic variables are s_1, s_2, s_3). We now have

$$s_1 = 20,\ s_2 = 16,\ s_3 = 10. \tag{6.18}$$

The objective function $Z = 0$, which corresponds to the corner point O in Fig. 6.1. In the present canonical form, the corresponding column associated with each basic variables has only one non-zero entry (marked by a box) for each constraint equality, and all other entries in the same column are zero. The non-zero value usually converts into 1 if it is not unity. This is shown as follows:

$$\begin{array}{c} Z \quad x_1 \quad x_2 \quad s_1 \quad s_2 \quad s_3 \\ \begin{pmatrix} 1 & -2 & -3 & 0 & 0 & 0 \\ 0 & 1 & 1 & \boxed{1} & 0 & 0 \\ 0 & 1 & 0 & 0 & \boxed{1} & 0 \\ 0 & 0 & 1 & 0 & 0 & \boxed{1} \end{pmatrix} \end{array} \tag{6.19}$$

When we change the set or the bases of basic variables from one set to another, we will aim to convert to a similar form using pivot row operations. There are two ways of numbering this matrix. One way is to call the first row [1 –2 –3 0 0 0] as the zeroth row, so that all other rows correspond to their corresponding constraint equation. The other way is simply to use its order in the matrix, so [1 –2 –3 0 0 0] is simply the first row. We will use these standard notations.

Now the question is if we can improve the objective by increasing one of the non-basic variables x_1 and x_2? Obviously, if we increase x_1 by a unit, then Z will also increase by 2 units. However, if we increase x_2 by a unit, then Z will increase by 3 units. Since our objective is to increase Z as much as possible, so we choose to increase x_2. As the requirement of the non-negativeness of all variables, we cannot increase x_2 without limit. So we increase x_2 while holding $x_1 = 0$, we have

$$s_1 = 20 - x_2,\ s_2 = 16,\ s_3 = 10 - x_2. \tag{6.20}$$

Thus, the highest possible value of x_2 is $x = 10$ when $s_1 = s_3 = 0$. If x_2 increases further, both s_1 and s_3 will become negative, thus it is no longer a basic feasible solution.

The next step is to use either by setting $x_1 = 0$ and $s_1 = 0$ as non-basic variables or by setting $x_1 = 0$ and $s_3 = 0$. Both cases correspond to the point A in our example, so we simply choose $x_1 = 0$ and $s_3 = 0$ as non-basic variables, and the basic variables are thus x_2, s_1 and s_2. Now we have to do some pivot operations so that s_3 will be replaced by x_2 as a new basic variable. Each constraint equation has only a single basic variable in the new canonical form. This means that each column corresponding to each basic variable should have only a single non-zero entry (usually 1). In addition, the right hand sides of all the constraints are non-negative and increase the value of the objective function at the same time. In order to convert the third column for x_2 to the form with only a single non-zero entry 1 (all other coefficients in the column should be zero), we first multiply the fourth row by 3 and add it to the first row, and the first row becomes

$$Z - 2x_1 + 0x_2 + 0s_1 + 0s_2 + 3s_3 = 30. \tag{6.21}$$

Then, we multiply the fourth row by -1 and add it to the second row, we have

$$0Z + x_1 + 0x_2 + s_1 + 0s_2 - s_3 = 10. \tag{6.22}$$

So the new canonical form becomes

$$\begin{pmatrix} 1 & -2 & 0 & 0 & 0 & 3 \\ 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Z \\ x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 30 \\ 10 \\ 16 \\ 10 \end{pmatrix}, \tag{6.23}$$

where the third, fourth, and fifth columns (for x_2, s_1 and s_2, respectively) have only one non-zero coefficient. All the right hand sides are non-negative. From this canonical form, we can find the basic feasible solution by setting non-basic variables equal to zero. This is to set $x_1 = 0$ and $s_3 = 0$. We now have the basic feasible solution

$$x_2 = 10,\ s_1 = 10,\ s_2 = 16, \tag{6.24}$$

which corresponds to the corner point A. The objective $Z = 30$.

Now again the question is whether we can improve the objective by increasing the non-basic variables. As the objective function is

$$Z = 30 + 2x_1 - 3s_3, \tag{6.25}$$

Z will increase 2 units if we increase x_1 by 1, but Z will decrease -3 if we increase s_3. Thus, the best way to improve the objective is to increase x_1. The question is what the limit of x_1 is. To answer this question, we hold s_3 at 0, we have

$$s_1 = 10 - x_1,\ s_2 = 16 - x_1, x_2 = 10. \tag{6.26}$$

We can see if x_1 can increase up to $x_1 = 10$, after that s_1 becomes negative, and this occurs when $x_1 = 10$ and $s_1 = 0$. This also suggests us that the new adjacent basic feasible solution can be obtained by choosing s_1 and s_3 as the non-basic variables. Therefore, we have to replace s_1 with x_1 so that the new basic variables are x_1, x_2 and s_2.

Using these basic variables, we have to make sure that the second column (for x_1) has only a single non-zero entry. Thus, we multiply the second row by 2 and add it to the first row, and the first row becomes

$$Z + 0x_1 + 0x_2 + 2s_1 + 0s_2 + s_3 = 50. \tag{6.27}$$

We then multiply the second row by -1 and add it to the third row, we have

$$0Z + 0x_1 + 0x_2 - s_1 + s_2 + s_3 = 6. \tag{6.28}$$

Thus we have the following canonical form

$$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Z \\ x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 50 \\ 10 \\ 6 \\ 10 \end{pmatrix}, \tag{6.29}$$

whose basic feasible solution can be obtained by setting non-basic variables $s_1 = s_3 = 0$. We have

$$x_1 = 10,\ x_2 = 10,\ s_2 = 6, \tag{6.30}$$

which corresponds to the extreme point B in Fig. 6.1. The objective value is $Z = 50$ for this basic feasible solution. Let us see if we can improve the objective further. Since the objective becomes

$$Z = 50 - 2s_1 - s_3, \tag{6.31}$$

any increase of s_1 or s_3 from zero will decrease the objective value. Therefore, this basic feasible solution is optimal. Indeed, this is the same solution as that obtained from the graph method. We can see that a major advantage is that we have reached the optimal solution after searching a certain number of extreme points, there is no need to evaluate the rest of the extreme points. This is exactly why the simplex method is so efficient.

The case study we used here is relatively simple, but it is useful to show how the basic procedure works in linear programming. For more practical applications, there are well-established software packages which will do the work for you once you have set up the objective and constraints properly.

Chapter 7

Nonlinear Optimization

As most of the real world problems are nonlinear, nonlinear mathematical programming thus forms an important part of mathematical optimization methods. A broad class of nonlinear programming problems is about the minimization or maximization of $f(\boldsymbol{x})$ subject to no constraints, and another important class is the minimization of a quadratic objective function subject to nonlinear constraints. There are many other nonlinear programming problems as well.

Nonlinear programming problems are often classified according to the convexity of the defining functions. An interesting property of a convex function f is that the vanishing of the gradient $\nabla f(\boldsymbol{x}_*) = 0$ guarantees that the point x_* is a global minimum or maximum of f. If a function is not convex or concave, then it is much more difficult to find global minima or maxima.

7.1 Penalty Method

For the simple function optimization with equality and inequality constraints, a common method is the penalty method. For the optimization problem

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

$$\psi_j(\boldsymbol{x}) \leq 0, \quad (j = 1, ..., N), \tag{7.1}$$

the idea is to define a penalty function $\Pi(\boldsymbol{x}, \mu_i, \nu_j)$ so that the constrained problem can be transformed into an unconstrained problem. Now we define

$$\Pi(\boldsymbol{x}, \mu_i, \nu_j) = f(\boldsymbol{x}) + \sum_{i=1}^{M} \mu_i \phi_i^2(\boldsymbol{x}) + \sum_{j=1}^{N} \nu_j \psi_j^2(\boldsymbol{x}), \tag{7.2}$$

where $\mu_i \gg 1$ and $\nu_j \geq 0$.

For example, in order to solve the following problem of the Gill-Murray-Wright type

$$\underset{x \in \Re}{\text{minimize}} \; f(x) = 100(x-b)^2 + 1,$$

$$\text{subject to } g(x) = x - a \geq 0, \tag{7.3}$$

where $a > b$ is a given value, we can define a penalty function $\Pi(x)$ using a penalty parameter $\mu \gg 1$. We have

$$\Pi(x, \mu) = f(x) + \frac{\mu}{2} g(x)^T g(x)$$

$$= 100(x-b)^2 + 1 + \frac{\mu}{2}(x-a)^2, \tag{7.4}$$

where the typical value for μ is $2000 \sim 10000$.

This method essentially transforms a constrained problem into an unconstrained one. From the stationary condition $\Pi'(x) = 0$, we have

$$200(x_* - b) - \mu(x_* - a) = 0, \tag{7.5}$$

which gives

$$x_* = \frac{200b + \mu a}{200 + \mu}. \tag{7.6}$$

For $\mu \to \infty$, we have $x_* \to a$. For $\mu = 2000$, $a = 2$ and $b = 1$, we have $x_* \approx 1.9090$. This means the solution depends on the value of μ, and it is very difficult to use extremely large values without causing extra computational difficulties.

7.2 Lagrange Multipliers

Another powerful methods without the above limitation of using large μ is the method of Lagrange multipliers. If we want to minimize a function $f(\boldsymbol{x})$

$$\underset{\boldsymbol{x}\in\Re^n}{\text{minimize}} f(\boldsymbol{x}), \qquad \boldsymbol{x} = (x_1, ..., x_n)^T \in \Re^n, \tag{7.7}$$

subject to the following nonlinear equality constraint

$$g(\boldsymbol{x}) = 0, \tag{7.8}$$

then we can combine the objective function $f(\boldsymbol{x})$ with the equality to form a new function, called the Lagrangian

$$\Pi = f(\boldsymbol{x}) + \lambda g(\boldsymbol{x}), \tag{7.9}$$

where λ is the Lagrange multiplier, which is an unknown scalar to be determined.

This essentially converts the constrained problem into an unconstrained problem for $\Pi(\boldsymbol{x})$, which is exactly the beauty of this method. If we have M equalities,

$$g_j(\boldsymbol{x}) = 0, \qquad (j = 1, ..., M), \tag{7.10}$$

then we need M Lagrange multipliers $\lambda_j (j = 1, ..., M)$. We thus have

$$\Pi(x, \lambda_j) = f(\boldsymbol{x}) + \sum_{j=1}^{M} \lambda_j g_j(\boldsymbol{x}). \tag{7.11}$$

The requirement of stationary conditions leads to

$$\frac{\partial \Pi}{\partial x_i} = \frac{\partial f}{\partial x_i} + \sum_{j=1}^{M} \lambda_j \frac{\partial g_j}{\partial x_i}, \ \ (i = 1, ..., n), \tag{7.12}$$

and

$$\frac{\partial \Pi}{\partial \lambda_j} = g_j = 0, \ \ (j = 1, ..., M). \tag{7.13}$$

These $M + n$ equations will determine the n-component of $\boldsymbol{x}$ and M Lagrange multipliers. As $\frac{\partial \Pi}{\partial g_j} = \lambda_j$, we can consider λ_j

as the rate of the change of the quantity Π as a functional of g_j.

Example 7.1: To solve the optimization problem

$$\underset{(x,y)\in\Re^2}{\text{maximize}}\, f(x,y) = xy^2,$$

subject to the condition

$$g(x,y) = x^2 + y^2 - 1 = 0.$$

We define

$$\Pi = f(x,y) + \lambda g(x,y) = xy^2 + \lambda(x^2 + y^2 - 1).$$

The stationary conditions become

$$\frac{\partial \Pi}{\partial x} = y^2 + 2\lambda x = 0,$$

$$\frac{\partial \Pi}{\partial y} = 2xy + 2\lambda y = 0,$$

and

$$\frac{\partial \Pi}{\partial \lambda} = x^2 + y^2 - 1 = 0.$$

The condition $xy+\lambda y = 0$ implies that $y = 0$ or $\lambda = -x$. The case of $y = 0$ can be eliminated as it leads to $x = 0$ from $y^2+2\lambda x = 0$, which does not satisfy the last condition $x^2 + y^2 = 1$. Therefore, the only valid solution is

$$\lambda = -x.$$

From the first stationary condition, we have

$$y^2 - 2x^2 = 0, \quad \text{or} \quad y^2 = 2x^2.$$

Substituting this into the third stationary condition, we have

$$x^2 - 2x^2 - 1 = 0,$$

which gives

$$x = \pm 1.$$

So we have four stationary points

$$P_1(1,\sqrt{2}),\ \ P_2(1,-\sqrt{2}),\ \ P_3(-1,\sqrt{2}),\ \ P_4(-1,-\sqrt{(2)}).$$

The values of function $f(x,y)$ at these four points are

$$f(P_1)=2,\ \ f(P_2)=2,\ \ f(P_3)=-2,\ \ f(P_4)=-2.$$

Thus, the function reaches its maxima at $(1,\sqrt{2})$ and $(1,-\sqrt{2})$. The Lagrange multiplier for this case is $\lambda=-1$.

7.3 Kuhn-Tucker Conditions

There is a counterpart of the Lagrange multipliers for the nonlinear optimization with constraint inequalities. The Kuhn-Tucker conditions concern the requirement for a solution to be optimal in nonlinear programming.

For the nonlinear optimization problem

$$\begin{aligned} &\underset{\boldsymbol{x}\in\Re^n}{\text{minimize}}\, f(\boldsymbol{x}),\\ &\text{subject to } \phi_i(\boldsymbol{x})=0,\ (i=1,...,M),\\ &\qquad\qquad \psi_j(\boldsymbol{x})\le 0,\ \ (j=1,...,N). \end{aligned} \tag{7.14}$$

If all the functions are continuously differentiable, at a local minimum $\boldsymbol{x}_*$, there exist constants $\lambda_0,\lambda_1,...,\lambda_q$ and $\mu_1,...,\mu_p$ such that

$$\lambda_0\nabla f(\boldsymbol{x}_*)+\sum_{i=1}^{M}\mu_i\nabla\phi_i(\boldsymbol{x}_*)+\sum_{j=1}^{N}\lambda_j\nabla\psi_j(\boldsymbol{x}_*)=0, \tag{7.15}$$

and

$$\psi_j(\boldsymbol{x}_*)\le 0,\qquad \lambda_j\psi_j(\boldsymbol{x}_*)=0,\ \ (j=1,2,...,N), \tag{7.16}$$

where $\lambda_j\ge 0,(i=0,1,...,N)$. The constants satisfy the following condition

$$\sum_{j=0}^{N}\lambda_j+\sum_{i=1}^{M}|\mu_i|\ge 0. \tag{7.17}$$

This is essentially a generalised method of the Lagrange multipliers. However, there is a possibility of degeneracy when $\lambda_0=0$ under certain conditions.

7.4 No Free Lunch Theorems

The methods used to solve a particular problem depend largely on the type and characteristics of the optimization problem itself. There is no universal method that works for all problems, and there is generally no guarantee to find the optimal solution in global optimization problems. In fact, there are several so-called Wolpert and Macready's 'No Free Lunch Theorems' (NLF theorems) which state that if any algorithm A outperforms another algorithm B in the search for an extremum of a cost function, then algorithm B will outperform A over other cost functions. NFL theorems apply to the scenario (either deterministic or stochastic) where a set of continuous (or discrete or mixed) parameter θ maps the cost functions into a finite set. Let n_θ be the number of values of θ (either due to discrete values or the finite machine precisions), and n_f be the number of values of the cost function. Then, the number of all possible combinations of cost functions is $N = n_f^{n_\theta}$ which is finite, but usually huge. The NFL theorems prove that the average performance over all possible cost functions is the same for all search algorithms.

Mathematically, if $P(s_m^y|f, m, A)$ denotes the performance, based on probability theory, of an algorithm A iterated m times on a cost function f over the sample s_m, then we have the averaged performance for two algorithms

$$\sum_f P(s_m^y|f, m, A) = \sum_f P(s_m^y|f, m, B), \tag{7.18}$$

where $s_m = \{(s_m^x(1), s_m^y(1)), ..., (s_m^x(m), s_m^y(m))\}$ is a time-ordered set of m distinct visited points with a sample of size m. The interesting thing is that the performance is independent of algorithm A itself. That is to say, all algorithms for optimization will give the same performance when averaged over *all possible* functions. This means that the universally best method does not exist.

Well, you might say, there is no need to formulate new algorithms because all algorithms will perform equally well. The

truth is that the performance is measured in the statistical sense and over *all possible* functions. This does not mean all algorithms perform equally well over some *specific* functions. The reality is that no optimization problems require averaged performance over all possible functions. Even though, the NFL theorems are valid mathematically, their influence on parameter search and optimization is limited. For any specific set of functions, some algorithms perform much better than others. In fact, for any specific problem with specific functions, there usually exist some algorithms that are more efficient than others if we do not need to measure their *average* performance. The main problem is probably how to find these better algorithms for a given particular type of problem.

On the other hand, we have to emphasize on the best estimate or sub-optimal solutions under the given conditions if the best optimality is not achievable. The knowledge about the particular problem concerned is always helpful for the appropriate choice of the best or most efficient methods for the optimization procedure.

The optimization algorithms we discussed so far are conventional methods and deterministic as there is no randomness in these algorithms. The main advantage of these algorithms is that they are well-established and benchmarked, but the disadvantages are that they could be trapped in a local optimum and there is no guarantee that they will find the global optimum even if you run the algorithms for infinite long because the diversity of the solutions is limited.

Modern optimization algorithms such as genetic algorithms and simulated annealing often involve a certain degree of randomness so as to increase the diversity of the solutions and also to avoid being trapped in a local optimum. The randomness makes sure that it can 'jump out' any local optimum. Statistically speaking it can find the global optima as the number of iterations approaches infinite. The employment of randomness is everywhere, especially in the metaheuristic methods such as particle swarm optimization and simulated annealing. We will study these methods in detail in the rest chapters of this book.

Part III

Metaheuristic Methods

Chapter 8

Tabu Search

Most of the algorithms discussed so far, especially, the gradient-based search methods, are local search methods. The search usually starts with a guess, and tries to improve the quality of the solutions. For unimodal functions, the convexity guarantees the final optimal solution is also a global optimum. For multimodal objectives, it is likely the search will get stuck at a local optimum. In order to get out a local optimum, certain variations with randomness should be used. A typical example is the genetic algorithm which is a global search algorithm. Another excellent example is the simulated annealing which is a global search method and guarantees to reach the global optimality as computing time approaches infinity. In reality, it finds the global optimality very efficiently.

The algorithms we will discuss in the rest of the book are metaheuristic methods, here *meta-* means 'beyond' or 'higher level' and *heuristic* means 'to find' or 'to discover by trial and error'. These metaheuristic methods include the Tabu search, ant colony optimization, particle swarm optimization, and simulated annealing. First, let us study the Tabu search.

8.1 Tabu Search

Tabu search, developed by Fred Glover in 1970s, is the search strategy that uses memory and search history as its integrated

component. As most of the successful search algorithms such as gradient-based methods do not use memory, it seems that at first it is difficult to see how memory will improve the search efficiency. Thus, the use of memory poses subtleties that may not be immediately visible on the surface. This is because memory could introduce too many degrees of freedom, especially for the adaptive memory use, which makes it almost impossible to use the rigorous theorems-and-proof approach to establish the convergence and efficiency of such algorithms. Therefore, even though Tabu search works so well for certain problems, it is difficult to analyse mathematically why it is so. Consequently, Tabu search remains a heuristic approach. Two important issues that are still under active research are how to use the memory effectively and how to combine with other algorithms to create more superior new generation algorithms.

Tabu search is an intensive local search algorithm and the use of memory avoids the the potential cycling of local solutions so as to increase the search efficiency. The recent tried or visited solutions are recorded and put into a Tabu list and new solutions should avoid those in the Tabu list.

The Tabu list is an important concept in Tabu search, and it records the search moves as the recent history, and any new search move should avoid this list of previous moves, this will inevitably save time as previous moves are not repeated. Over a large number of iterations, this Tabu list could save tremendous amount of computing time and thus increase the search efficiency significantly.

Let us illustrate the concept of the Tabu list through an example. Suppose we want to go through each route (straight-line) once and only once without repetition (see Fig. 8.1).

Example 8.1: If we start the travel from, say, point E, we first go along EA and reach point A. Now put the route EA in the Tabu list which is

$$\text{Tabu list} = \{\text{EA}\} ,$$

where we do not distinguish between EA and AE, though such difference is important for directed graphs. When we reach point

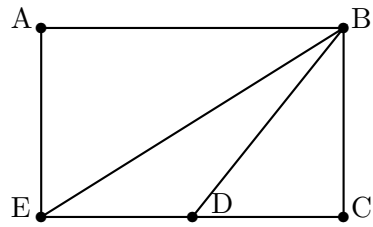


Figure 8.1: Passing through each line once and only once.

A, the next move can only be AB because EA is in the Tabu list so going back is not allowed. We now update the Tabu list as

$$\text{Tabu list} = \{\text{EA, AB}\} .$$

At point B, there are four possible routes, but only three (BE, BD, BC) are permissible routes because AB is now in the updated Tabu list. We see that the length of the Tabu list is increasing and flexible. Suppose we randomly choose the route BD to go to point D. The Tabu list now becomes

$$\text{Tabu list} = \{\text{EA, AB, BD}\} .$$

Now at point D, we can either go along BC or along DE, but we cannot go back along BD. So we randomly choose DE to reach point E.

$$\text{Tabu list} = \{\text{EA, AB, BD, DE}\} .$$

At point E, we have three possible routes, but only EB is permissible as the other two routes are forbidden. So we have to choose route EB to go back to point B, and the current Tabu list becomes

$$\text{Tabu list} = \{\text{EA, AB, BD, DE, EB}\} .$$

Again at point B, there are four routes, but only BC is permissible and all other three routes are in the Tabu list and thus not allowed. So we have to go along BC to reach point C. The updated Tabu list is

$$\text{Tabu list} = \{\text{EA, AB, BD, DE, EB, BC}\} .$$

Now the only permissible route at point C is CD to go to point D. Here we complete the task of going through each branch once and only once. If we now update the Tabu list, we get

$$\text{Tabu list} = \{\text{EA, AB, BD, DE, EB, BC, CD}\} ,$$

which is the complete route map for this task.

Here we have used the whole history as the Tabu list. There no reason why we must use the whole history, we can use it if the computer memory is not an issue. More often, we only use part of the history as an ongoing Tabu list, usually with a fixed length. If we use a fixed length $n = 5$ in our example, then the last two steps, the Tabu list should be replaced by

$$\text{Tabu list} = \{\text{AB, BD, DE, EB, BC}\} ,$$

and

$$\text{Tabu list} = \{\text{BD, DE, EB, BC, CD}\} .$$

We can see that the Tabu list serves as a way of mimicking the way we humans tackle the problem by 'trial and error' (heuristically) and by memorizing the steps we have tried. This is probably why the Tabu search is much more efficient than memoryless search methods. Furthermore, the advantage of a fixed length Tabu list is that it saves memory and is easy to implement, but it has the disadvantage of incomplete information. Effective usage of memory in Tabu search is still an area of active research. In general, a Tabu list should be flexible with enough information and should be easy for implementation in any programming language.

Tabu search was originally developed together with the integer programming, a special class of linear programming. The Tabu list in combination with integer programmming can reduce the computing time by at least two orders for a given problem, comparing the solely standard integer programming. The Tabu search can also be used along with many other algorithms.

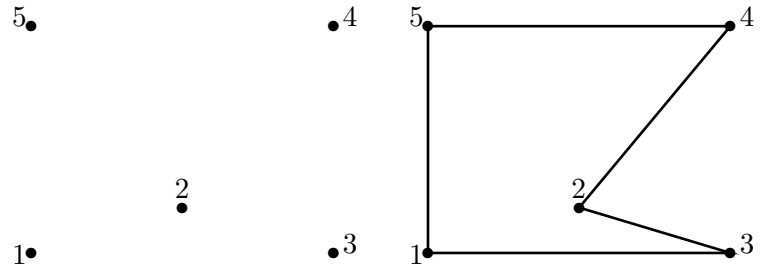


Figure 8.2: Travelling salesman problem for five cities and a possible route (not optimal).

8.2 Travelling Salesman Problem

Travelling salesman problem (TSP) is an optimization problem for a given number of cities (say, n) and their locations. The cities are represented as nodes in a graph and the distances between two cities are represented as the weight of a directed edge or route between the two cities. The aim is to find a path that visits each city once, and returning to the starting city, minimizing the total distance travelled. This is a very difficult problem as its only known solution to find the shortest path requires a solution time that grows exponentially with the problem size n. In fact, the travelling salesman problem is an NP-hard problem, which means that there are no known algorithms (of polynomial time) that exist.

For simplicity, we now look at the travelling salesman problem with only five cities: city 1 at (0,0), city 2 at (5,2), city 3 at (10,0), city 4 at (10,10), and city 5 at (0,10). Here the unit is 1000 km. There are many possible routes. Is the route 2-3-1-5-4-2 optimal? If not, how to prove this?

Let us first try an exhaustive search. The Euclidean distance between any two cities i and j is given by

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}), \qquad (i, j = 1, 2, ..., 5), \quad (8.1)$$

where (x_i, y_i) are the Cartesian coordinates of city i. This will

give the following distance matrix d

$$\boldsymbol{d} = \begin{pmatrix} 0 & 5.39 & 10.00 & 14.14 & 10.00 \\ 5.39 & 0 & 5.39 & 9.43 & 9.43 \\ 10.00 & 5.39 & 0 & 10.00 & 14.14 \\ 14.14 & 9.43 & 10.00 & 0 & 10.00 \\ 10.00 & 9.43 & 14.14 & 10.00 & 0 \end{pmatrix}, \tag{8.2}$$

where the diagonal elements are zero because the distance of a city to itself is zero.

In order to avoid any potential problem in finding the shortest distance, we now set the distance $d_{ii} = \infty$ for $i = 1, 2, .., 5$. This leads to

$$\boldsymbol{d} = \begin{pmatrix} \infty & 5.39 & 10.00 & 14.14 & 10.00 \\ 5.39 & \infty & 5.39 & 9.43 & 9.43 \\ 10.00 & 5.39 & \infty & 10.00 & 14.14 \\ 14.14 & 9.43 & 10.00 & \infty & 10.00 \\ 10.00 & 9.43 & 14.14 & 10.00 & \infty \end{pmatrix}. \tag{8.3}$$

As there are $n = 5$ cities, there are $5! = 120$ ways of visiting these cities such as 2-5-3-1-4-2, 5-4-1-2-3-5 (see Fig. 8.3) and so on and so forth. For example, the route 2-5-3-1-4-2 has a total distance

$$9.43 + 14.14 + 10.00 + 14.14 + 9.43 \approx 57.14, \tag{8.4}$$

while the route 5-4-1-2-3-5 has a total distance

$$10.00 + 14.14 + 5.39 + 5.39 + 14.14 \approx 49.06. \tag{8.5}$$

By searching over all the 120 combinations and calculating the total distance for each case, we can finally find the shortest route is 1-2-3-4-5-1 (or their ordered permutations such as 2-3-4-5-1-2 and 4-5-1-2-3-4) with a distance $5.39 + 5.39 + 10.00 + 10.00 + 10.00 \approx 40.78$.

For $n = 5$, we can use the exhaustive search because $n! = 120$ is small. Now suppose $n = 50$, the number of possible combinations is $50! \approx 3.04 \times 10^{64}$ while for $n = 100$ cities, $100! \approx 9.3 \times 10^{157}$. Even with all modern computers working

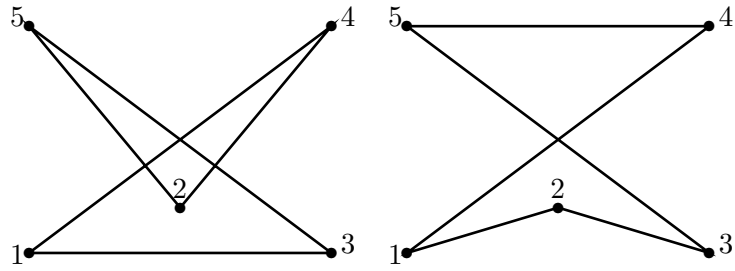


Figure 8.3: Two different routes: 2-5-3-1-4-2 and 5-4-1-2-3-5.

in parallel, it would takes the time much longer than the age of the universe to do such a huge number of calculations. So we have to use other methods.

8.3 Tabu Search for TSP

Now let us use Tabu search for the travelling salesman problem. As there is no known efficient method for solving such problems, we use a method of systematic swapping. For example, when $n = 3$ cities, any order is an optimal 1-2-3 or 2-3-1 and others. For $n =$ four cities, suppose we know that optimal route is 1-2-3-4-1 (its ordered permutation 2-3-4-1-2 and others are the same route). However, initially we start randomly from, say, 2-4-3-1-2. A simple swap between cities 3 and 4 leads to 2-3-4-1-2 which is the optimal solution. Of course, we do not know which two cities should be swapped, so we need a systematic approach by swapping any two cities i and j. Furthermore, in order to swap any two cities, say, 2 and 4, we can either simple swap them or we can use a more systematical approach by swapping 2 and 3, then 3 and 4. The latter version takes more steps but it does provide a systematic way of implementing the algorithms. Thus, we will use this latter approach.

For simplicity, we still use the same example of the five cities. Suppose we start from a random route 2-5-3-1-4-2 (see Fig. 8.3). In order to swap any two cities among these five cities, we use the following swap indices by swapping two ad-

jacent cities

$$\text{swap} = \begin{pmatrix} 2 & 1 & 3 & 4 & 5 \\ 1 & 3 & 2 & 4 & 5 \\ 1 & 2 & 4 & 3 & 5 \\ 1 & 2 & 3 & 5 & 4 \\ 5 & 2 & 3 & 4 & 1 \end{pmatrix}, \tag{8.6}$$

where the first row swaps the first two cities and the fifth row swap the first and last (fifth) cities. In order to avoid repetition of recent swaps, we use a Tabu list for the above index matrix

$$\text{Tabu list} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{8.7}$$

where the first element corresponds to the first row of the swap index matrix. Now we also use a fixed length memory, say, $m = 2$ which means that we only record two steps in recent history.

From the initial route= [2 5 3 1 4] with an initial distance $d = 57.14$, let us try the new route by swapping the first two cities or route = route[swap(1)]. We have

$$\text{route} = [5 \;\; 2 \;\; 3 \;\; 1 \;\; 4], \tag{8.8}$$

whose total distance is 48.96 which is shorter than $d = 57.14$ so we update the new shortest distance as $d = 48.96$. Here we can update the Tabu list, but it is better to update it when we find a better route after going through all the five swaps.

Now we generate the new route by swapping the second two pair or route = route [swap(2)]. We now have

$$\text{route} = [2 \;\; 3 \;\; 5 \;\; 1 \;\; 4], \tag{8.9}$$

whose the total distance is 53.10 which is larger than d so discard this attempt and still use $d = 48.96$.

Similarly, try the next swap, we get

$$\text{route} = [2 \;\; 5 \;\; 1 \;\; 3 \;\; 4], \tag{8.10}$$

with a distance 48.87, which is shorter than $d = 48.96$, so we update $d = 48.87$. The next swap leads to route=[2 5 3 4 1]

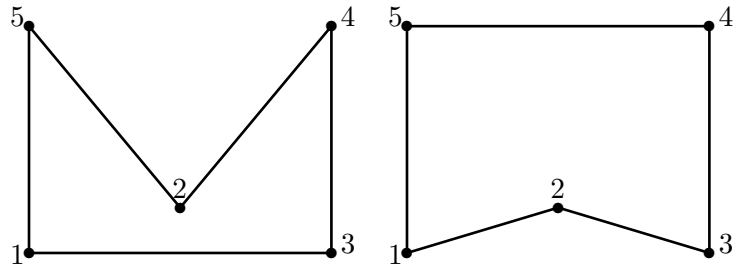


Figure 8.4: Current best route after first generation of swaps and the final optimal route.

with a distance 53.10, so we discard it. The fifth swap gives route=[4 5 3 1 2] with a distance 48.96 which is longer than $d = 48.87$, so we also discard.

After running through all the possible five swaps, the current best route is

$$\text{route} = [2 \;\; 5 \;\; 1 \;\; 3 \;\; 4], \tag{8.11}$$

with $d = 48.87$. Now we have to update the Tabu list and it becomes

$$\text{Tabu list} = \begin{pmatrix} 0 & 0 & m & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 \end{pmatrix}, \tag{8.12}$$

here the entry corresponding to the third swap is recorded.

Now we have to start all the five sequential swaps again from this current best route=[2 5 1 3 4]. The first swap leads to route=[5 2 1 3 4] with a distance of 44.82 which is shorter than 48.87. So the new distance is now $d = 44.82$. The second swap, we got the new route

$$\text{route} = [2 \;\; 1 \;\; 5 \;\; 3 \;\; 4], \tag{8.13}$$

whose distance is 48.96 so we discard it. As the next entry in the Tabu list is 2, we will not carry out this swap. Similarly, the next swap leads to route=[2 5 1 4 3] with a distance of 48.96 so we discard it. Finally, the fifth swap gives the route=[4 5 1 3 2] with a distance of 44.82 which is the same as the first swap,

so there is no need to update. After all the four permissible swaps, the current best route

$$\text{route} = [5 \;\; 2 \;\; 1 \;\; 3 \;\; 4], \tag{8.14}$$

and the new Tabu list is

$$\text{Tabu list} = \begin{pmatrix} 2 & 0 & 1 & 0 & 0 \end{pmatrix}. \tag{8.15}$$

where the third element is reduced from 2 to 1, which means the history is running and we can use the swap again later when this entry becomes zero.

Starting yet again from the current best route=[5 2 1 3 4] after the two generation swaps (see Fig. 8.4), the first swap is not allowed because the first entry in the Tabu list is 2. So we start the second swap which leads to

$$\text{route} = [5 \;\; 1 \;\; 2 \;\; 3 \;\; 4], \tag{8.16}$$

which has a total distance $d = 40.78$ and this is the global optimal route for this five cities (see Fig. 8.4).

The next swap leads to route=[5 2 1 4 3] with a distance of 53.10. As we continue, there is no further improvement in the solution quality. As we do not know whether the current optimal solution is the global optimality or not, we will continue the search until a prescribed number of iterations or after certain number of iterations with no further improvement in solution quality. In this present case, the global optimal solution is the route =[5 1 2 3 4] or 5-1-2-3-4-5 or its ordered permutations such as 2-3-4-5-1-2 and others.

We have seen that the Tabu search only takes about 10 steps to reach the global optimal route, and this is much more efficient comparing with the 120 steps by the exhaustive method. In addition, the Tabu list avoided at least 3 possible repetitions, saving about 1/3 of the moves. In general, for problems with large n, the computing time saved using Tabu search will be more significant.

Chapter 9

Ant Colony Optimization

From the Tabu search, we know that we can improve the search efficiency by using memory. Another way of improving the efficiency is to use the combination of randomness and memory. The randomness will increase the diversity of the solutions so as to avoid being trapped in local optima. The memory does not means to use simple history records. In fact, there are other forms of memory using chemical messenger such as pheromone which is commonly used by ants, honeybees, and many other insects. In this chapter, we will discuss the nature-inspired ant colony optimization (ACO), which is a metaheuristic method.

9.1 Behaviour of Ants

Ants are social insects in habit and they live together in organized colonies whose population size can range from about 2 to 25 millions. When foraging, a swarm of ants or mobile agents interact or communicate in their local environment. Each ant can lay scent chemicals or pheromone so as to communicate with others, and each ant is also able to follow the route marked with pheromone laid by other ants. When ants find a food source, they will mark it with pheromone and also

mark the trails to and from it. From the initial random foraging route, the pheromone concentration varies and the ants follow the route with higher pheromone concentration, and the pheromone is enhanced by increasing number of ants. As more and more ants follow the same route, it becomes the favoured path. Thus, some favourite route (often the shortest or more efficient) emerges. This is actually a positive feedback mechanism.

Emerging behaviour exists in an ant colony and such emergence arises from simple interactions among individual ants. Individual ants act according to simple and local information (such as pheromone concentration) to carry out their activities. Although there is no master ant overseeing the entire colony and broadcasting instructions to the individual ants, organized behaviour still emerges automatically. Therefore, such emergent behaviour is similar to other self-organized phenomena which occur in many processes in nature such as the pattern formation in animal skins (tiger and zebra skins).

The foraging pattern of some ant species (such as the army ants) can show extraordinary regularity. Army ants search for food along some regular routes with an angle of about 123° apart. We do not know how they manage to follow such regularity, but studies show that they could move in an area and build a bivouac and start foraging. On the first day, they forage in a random direction, say, the north and travel a few hundred meters, then branch to cover a large area. The next day, they will choose a different direction, which is about 123° from the direction on the previous day and cover a large area. On the following day, they again choose a different direction about 123° from the second day's direction. In this way, they cover the whole area over about 2 weeks and they move out to a different location to build a bivouac and forage again.

The interesting thing is that they do not use the angle of $360°/3 = 120°$ (this would mean that on the fourth day, they will search on the empty area already foraged on the first day). The beauty of this 123° angle is that it leaves an angle of about 10° from the direction on the first day. This means they cover

Ant Colony Optimization

begin
Objective function $f(\boldsymbol{x})$, $\boldsymbol{x} = (x_1, ..., x_n)^T$
[or $f(\boldsymbol{x}_{ij})$ *for routing problem where* $(i, j) \in \{1, 2, ..., n\}$*]*
Define pheromone evaporate rate γ
while *(criterion)*
for *loop over all n dimensions (or nodes)*
Generate new solutions
Evaluate the new solutions
Mark better locations/routes with pheromone $\delta\phi_{ij}$
Update pheromone: $\phi_{ij} \leftarrow (1 - \gamma)\phi_{ij} + \delta\phi_{ij}$
end for
Daemon actions such as finding the current best
end while
Output the best results and pheromone distribution
end

Figure 9.1: Pseudo code of ant colony optimization.

the whole circle in 14 days without repeating (or covering an previously-foraged area). This is an amazing phenomenon.

9.2 Ant Colony Optimization

Based on these characteristics of ant behaviour, scientists have developed a number of powerful ant colony algorithms with important progress made in recent years. Marco Dorigo pioneered the research in this area in 1992. In fact, we only use some of the nature or the behaviour of ants and add some new characteristics, we can devise a class of new algorithms.

The basin steps of the ant colony optimization (ACO) can be summarized as the pseudo code shown in Fig. 9.1.

Two important issues here are: the probability of choosing a route, and the evaporation rate of pheromone. There are a few ways of solving these problems although it is still an area of active research. Here we introduce the current best methods. For a network routing problem, the probability of ants at a

particular node i to choose the route from node i to node j is given by

$$p_{ij} = \frac{\phi_{ij}^{\alpha} d_{ij}^{\beta}}{\sum_{i,j=1}^{n} \phi_{ij}^{\alpha} d_{ij}^{\beta}}, \tag{9.1}$$

where $\alpha > 0$ and $\beta > 0$ are the influence parameters, and their typical values are $\alpha \approx \beta \approx 2$. ϕ_{ij} is the pheromone concentration on the route between i and j, and d_{ij} the desirability of the same route. Some *a priori* knowledge about the route such as the distance s_{ij} is often used so that $d_{ij} \propto 1/s_{ij}$, which implies that shorter routes will be selected due to the travelling time is shorter, and thus the pheromone concentration is higher.

This probability formula reflects the fact that ants would normally follow the paths with higher pheromone concentration. In the simpler case when $\alpha = \beta = 1$, the probability of choosing a path by ants is proportional to the pheromone concentration on the path. The denominator normalizes the probability so that it is in the range between 0 and 1.

The pheromone concentration can change with time due to the evaporation of pheromone. Furthermore, the advantage of pheromone evaporation is that it could avoid the system being trapped in local optima. If there is no evaporation, then the path randomly chosen by the first ants will become the preferred path as the attraction of other ants by their pheromone. For a constant rate γ of pheromone decay or evaporation, the pheromone concentration usually varies with time exponentially

$$\phi(t) = \phi_0 e^{-\gamma t}, \tag{9.2}$$

where ϕ_0 is the initial concentration of pheromone and t is the time. If $\gamma t \ll 1$, then we have $\phi(t) \approx (1 - \gamma t)\phi_0$. For the unitary time increment $\Delta t = 1$, the evaporation can be approximated by $\phi^{t+1} \leftarrow (1 - \gamma)\phi^t$. Therefore, we have the simplified pheromone update formula:

$$\phi_{ij}^{t+1} = (1 - \gamma)\phi_{ij}^{t} + \delta\phi_{ij}^{t}, \tag{9.3}$$

where $\gamma \in [0, 1]$ is the rate of pheromone evaporation. The increment $\delta\phi_{ij}^{t}$ is the amount of pheromone deposited at time

t along route i to j when an ant travels a distance L. Usually $\delta\phi_{ij}^t \propto 1/L$. If there are no ants on a route, then the pheromone deposit is zero.

There are other variations to these basic procedures. A possible acceleration scheme is to use some bounds of the pheromone concentration and only the ants with the current global best solution(s) are allowed to deposit pheromone. In addition, certain ranking of solution fitness can also be used. These are hot topics of current research.

9.3 Double Bridge Problem

A standard test problem for ant colony optimization is the simplest double bridge problem with two branches (see Fig. 9.2) where route (2) is shorter than route (1). The angles of these two routes are equal at both point A and point B so that the ants have equal chance (or 50-50 probability) of choosing each route randomly at the initial stage at point A.

Initially, fifty percent of the ants would go along the longer route (1) and the pheromone evaporates at a constant rate, but the pheromone concentration will become smaller as route (1) is longer and thus takes more time to travel through. Conversely, the pheromone concentration on the shorter route will increase steadily. After some iterations, almost all the ants will move along the shorter routes. Figure 9.3 shows the initial snapshot of 10 ants (5 on each route initially) and the snapshot after 5 iterations (or equivalent to 50 ants have moved along this section). Well, there are 11 ants, and one has not decided which route to follow as it just comes near to the entrance. Almost all the ants (well, about 90% in this case) move along the shorter route.

Here we only use two routes at the node, it is straightforward to extend it to the multiple routes at a node. It is expected that only the shortest route will be chosen ultimately. As any complex network system is always made of individual nodes, this algorithms can be extended to solve complex routing problems reasonably efficiently. In fact, the ant colony al-

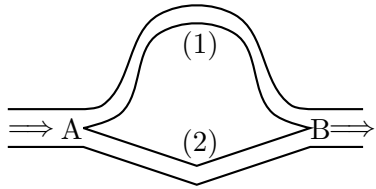


Figure 9.2: Test problem for routing performance: route (2) is shorter than route (1).

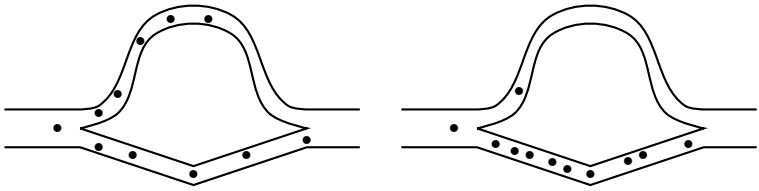


Figure 9.3: Route selection via ACO: (a) initially, ants choose each route with 50-50 probability, and (b) almost all ants move along the shorter route after 5 iterations.

gorithms have been successfully applied to the Internet routing problem, travelling salesman problem, combinatorial optimization problem, and other NP-hard problems.

9.4 Multi-Peak Functions

As we know that ant colony optimization has successfully solved NP-hard problems such as travelling salesman problems, it can also be extended to solve the standard optimization problems of multimodal functions. The only problem now is to figure out how the ants will move on a n-dimensional hyper-surface. For simplicity, we now make the discussion using the 2-D case which can easily be extended to higher dimensions. On a 2D landscape, ants can move in any direction or $0^\circ \sim 360^\circ$, but this will cause some problems. How to update the pheromone

at a particular point as there are infinite number of points. One solution is to track the history of each ant moves and record the locations consecutively, and the other approach is to use a moving neighbourhood or window. The ants 'smell' the pheromone concentration of their neighbourhood at any particular location.

In addition, we can limit the number of directions the ants can move. For example, ants are only allowed to move left and right, and up and down (only 4 directions). We will use this quantized approach here, which will make the implementation much simpler. Furthermore, the objective function or landscape can be encoded into virtual food so that ants will move the best locations where the best food sources are. This will make the search process even more simpler. This simplified algorithm is called Virtual Ant Algorithm (VAA) developed by Xin-She Yang and his colleagues in 2006, which has been successfully applied to optimization problems in engineering.

The following Keane function with multiple peaks is a standard test function

$$f(x,y) = \frac{\sin^2(x-y)\sin^2(x+y)}{\sqrt{x^2+y^2}}, \tag{9.4}$$

where

$$(x,y) \in [0,10] \times [0,10].$$

This function without any constraint is symmetric and has two highest peaks at $(0, 1.39325)$ and $(1.39325, 0)$. To make the problem harder, it is usually optimized under two constraints:

$$x + y \le 15, \qquad xy \ge \frac{3}{4}. \tag{9.5}$$

This makes the optimization difficult because it is now nearly symmetric about $x = y$ and the peaks occur in pairs where one is higher than the other. In addition, the true maximum is $f(1.593, 0.471) \approx 0.365$, which is defined by a constraint boundary.

Figure 9.4 shows the surface variations of the multi-peaked function. If we use 50 roaming ants and let them move around

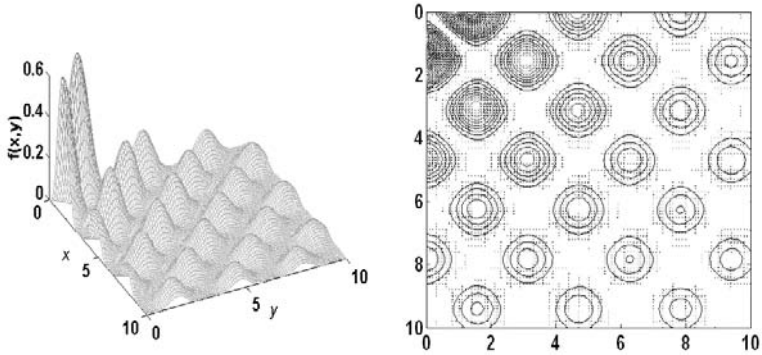


Figure 9.4: Surface variations of the multi-peak function.

for 25 iterations, then the pheromone concentrations (also equivalent to the paths of ants) are displayed in Fig. 9.4.

It is worth pointing out that ant colony algorithms are the right tool for combinatorial and discrete optimization. They have the advantages over other stochastic algorithms such as genetic algorithms and simulated annealing in dealing with dynamical network routing problems.

For continuous decision variables, its performance is still under active research. For the present example, it took about 1500 evaluations of the objective function so as to find the global optima. This is not as efficient as other metaheuristic methods, especially comparing with particle swarm optimization. This is partly because the handling of the pheromone takes time. Is it possible to eliminate the pheromone and just use the roaming ants? The answer is yes. Particle swarm optimization is just the right kind of algorithm for such further modifications. This is the topic of the next chapter.

Chapter 10

Particle Swarm Optimization

10.1 Swarm Intelligence

Particle swarm optimization (PSO) was developed by Kennedy and Eberhart in 1995, based on the swarm behaviour such as fish and bird schooling in nature. Many algorithms (such as ant colony algorithms and virtual ant algorithms) use the behaviour of the so-called swarm intelligence. Though particle swarm optimization has many similarities with genetic algorithms and virtual ant algorithms, but it is much simpler because it does not use mutation/crossover operators or pheromone. Instead, it uses the real-number randomness and the global communication among the swarm particles. In this sense, it is also easier to implement as there is no encoding or decoding of the parameters into binary strings as those in genetic algorithms.

This algorithm searches a space of an objective function by adjusting the trajectories of individual agents, called particles, as the piecewise path formed by positional vectors in a quasi-stochastic manner. The particle movement has two major components: a stochastic component and a deterministic component. The particle is attracted toward the position of the current global best while at the same time it has a tendency to move randomly. When a particle finds a location that

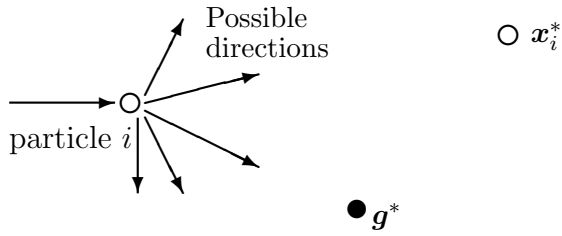


Figure 10.1: Schematic representation of the motion of a particle in PSO, moving towards the global best $\boldsymbol{g}^*$ and the current best $\boldsymbol{x}_i^*$ for each particle i.

is better than any previously found locations, then it updates it as the new current best for particle i. There is a current best for all n particles. The aim is to find the global best among all the current best until the objective no longer improves or after a certain number of iterations.

10.2 PSO algorithms

The essential steps of the particle swarm optimization can be summarized as the pseudo code shown in Fig. 10.2. The particle movement is schematically represented in Fig. 10.1 where $\boldsymbol{x}_i^*$ is the current best for particle i, and $\boldsymbol{g}^* \approx \min / \max\{f(\boldsymbol{x}_i)\}$, $(i = 1, 2, ..., n)$ is the current global best.

Let $\boldsymbol{x}_i$ and $\boldsymbol{v}_i$ be the position vector and velocity for particle i, respectively. The new velocity vector is determined by

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{v}_i^t + \alpha\boldsymbol{\epsilon}_1 \odot [\boldsymbol{g}^* - \boldsymbol{x}_i^t] + \beta\boldsymbol{\epsilon}_2 \odot [\boldsymbol{x}_i^* - \boldsymbol{x}_i^t]. \qquad (10.1)$$

where $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$ are two random vectors, and each entry taking the value between 0 and 1. The Hadamard product of two matrices $\boldsymbol{u} \odot \boldsymbol{v}$ is defined as the entrywise product, that is $[\boldsymbol{u} \odot \boldsymbol{v}]_{ij} = u_{ij}v_{ij}$.

The initial values of $\boldsymbol{x}(i, j, t = 0)$ can be taken as the boundary values $[a = \min(x_j), b = \max(x_j)]$ and $\boldsymbol{v}_i^{t=0} = 0$.

Particle Swarm Optimization

begin
Objective function $f(\boldsymbol{x})$, $\boldsymbol{x} = (x_1, ..., x_p)^T$
Initialize locations $\boldsymbol{x}_i$ *and velocity* $\boldsymbol{v}_i$ *of* n *particles.*
Initial minimum $f_{\min}^{t=0} = \min\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_n)\}$ *(at* $t = 0$*)*
while *(criterion)*
$t = t + 1$
for *loop over all* n *particles and all* p *dimensions*
Generate new velocity $\boldsymbol{v}_i^{t+1}$ *using equation (10.1)*
Calculate new locations $\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1}$
Evaluate objective functions at new locations $\boldsymbol{x}_i^{t+1}$
Find the current minimum $f_{\min}^{t+1}$
end for
Find the current best $\boldsymbol{x}_i^*$ *and current global best* $\boldsymbol{g}^*$
end while
Output the results $\boldsymbol{x}_i^*$ *and* $\boldsymbol{g}^*$
end

Figure 10.2: Pseudo code of particle swarm optimization.

The parameters α and β are the learning parameters or acceleration constants, which can typically be taken as, say, $\alpha \approx \beta \approx 2$. The new position can then be updated by

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1}. \tag{10.2}$$

Although $\boldsymbol{v}$ can be any values, it is usually bounded in some range $[0, \boldsymbol{v}_{max}]$.

10.3 Accelerated PSO

There are many variations which extend the standard algorithm, and the most noticeably the algorithms use inertia function $\theta(t)$ so that $\boldsymbol{v}_i^t$ is replaced by $\theta(t)\boldsymbol{v}_i^t$ where θ takes the values between 0 and 1. In the simplest case, the inertia function can be take as a constant, say, $\theta \approx 0.5 \sim 0.9$. This is equivalent to introduce a virtual mass to stabilize the motion of the particles, and thus the algorithm is expected to converge more quickly.

The standard particle swarm optimization uses both the current global best $\boldsymbol{g}^*$ and the individual best $\boldsymbol{x}_i^*$. The reason of using the individual best is primarily to increase the diversity in the quality solution, however, this diversity can be simulated using the randomness. Subsequently, there is no compelling reason for using the individual best. A simplified version which could accelerate the convergence of the algorithm is to use the global best only. Thus, in the accelerated particle swarm optimization, the velocity vector is generated by

$$\boldsymbol{v}_i^{t+1} = \boldsymbol{v}_i^t + \alpha(\boldsymbol{\epsilon} - 0.5) + \beta(\boldsymbol{g}^* - \boldsymbol{x}_i), \tag{10.3}$$

where $\boldsymbol{\epsilon}$ is a random variable with values from 0 to 1. The update of the position is simply

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \boldsymbol{v}_i^{t+1}. \tag{10.4}$$

In order to increase the convergence even further, we can also write the update of the location in a single step

$$\boldsymbol{x}_i^{t+1} = (1-\beta)\boldsymbol{x}_i + \beta\boldsymbol{g}^* + \alpha(\boldsymbol{\epsilon} - 0.5). \tag{10.5}$$

This simpler version will give the same order of convergence. The typical values for this accelerated PSO are $\alpha \approx 0.1 \sim 0.4$ and $\beta \approx 0.1 \sim 0.7$, though $\alpha \approx 0.2$ and $\beta \approx 0.5$ are recommended for most unimodal objective functions.

A further accelerated PSO is to reduce the randomness as iterations proceed. This mean that we can use a monotonically decreasing function such as

$$\alpha = \alpha_0 e^{-\gamma t}, \qquad \text{or} \qquad \alpha = \alpha_0 \gamma^t, \qquad (\gamma < 1), \tag{10.6}$$

where $\alpha_0 \approx 0.5 \sim 1$ is the initial value of the randomness parameter. t is the number of iterations or time steps. $\gamma < 1$ is a control parameter. For example, in our implementation given later in this chapter, we will use

$$\alpha = 0.7^t, \tag{10.7}$$

where $t \in [0, 10]$. The implementation of the accelerated PSO in Matlab and Octave is given later. Various studies show

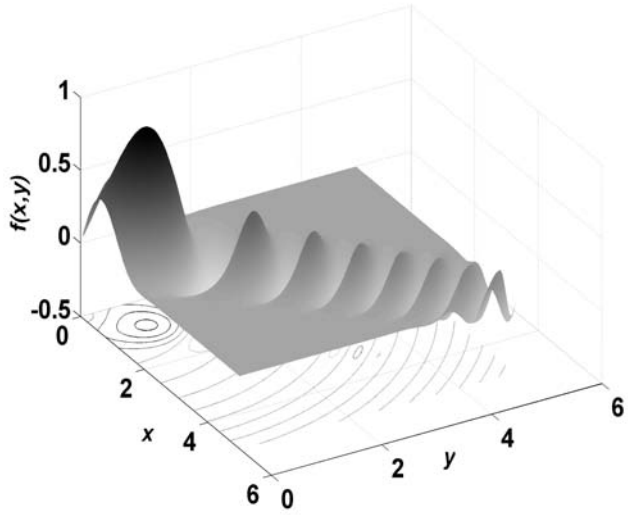


Figure 10.3: Multiple peaks with a global maximum at $(0.7634, 0.7634)$. We have used $\lambda = 1$.

that PSO algorithms can outperform genetic algorithms and other conventional algorithms for solving many optimization problems. This is partially due to that fact that the broadcasting ability of current best estimates gives a better and quicker convergence towards the optimality. However, this algorithm is almost memoryless since it does not record the movement paths of each particle, and it is expected that it can be further improved using short-term memory in the similar fashion as that in Tabu search.

10.4 Multimodal Functions

Multiple peak functions are often used to validate new algorithms. We can construct the following function with multiple peaks,

$$f(x,y) = \frac{\sin(x^2+y^2)}{\sqrt{x^2+y^2}} e^{-\lambda(x-y)^2}, \qquad \lambda > 0, \tag{10.8}$$

where $(x, y) \in [0, 5] \times [0, 5]$. Obviously, for a minimization problem, we can write it as

$$f(x, y) = -\frac{\sin(x^2 + y^2)}{\sqrt{x^2 + y^2}} e^{-\lambda(x-y)^2}, \qquad \lambda > 0. \tag{10.9}$$

Here we have designed our own test function, thought it is a standard practice to benchmark new algorithms against the test functions in the literature. In fact, various studies have already benchmarked the PSO algorithms against standard test functions. Later, we will also use the Michaelewicz function in our implementation as a simple demo.

The reason we devised our own function is that this multiple peak function has a global maximum in the domain and its location is independent of $\lambda > 0$. If λ is very small, say, $\lambda = 0.01$, then it is very difficult to find the optimality for many algorithms. Rosenbrock's banana function has similar properties.

The first derivatives of f are

$$\frac{\partial f}{\partial x} = \frac{e^{-\lambda(x-y)^2}}{\sqrt{x^2 + y^2}} \{2x \cos(x^2 + y^2)$$

$$-[\frac{x}{x^2 + y^2} + 2\lambda(x - y)] \tan(x^2 + y^2)\}, \tag{10.10}$$

$$\frac{\partial f}{\partial y} = \frac{e^{-\lambda(x-y)^2}}{\sqrt{x^2 + y^2}} \{2y \cos(x^2 + y^2)$$

$$-[\frac{y}{x^2 + y^2} + 2\lambda(x - y)] \tan(x^2 + y^2)\}. \tag{10.11}$$

As the function is symmetric in terms of x and y, the maxima must be on the line of $x = y$ and are determined by

$$\frac{\partial f}{\partial x} = 0, \qquad \frac{\partial f}{\partial y} = 0. \tag{10.12}$$

By using $x = y$ and $x > 0$, we have

$$2x \cos(2x^2) - \frac{1}{2x} \sin(2x^2) = 0, \tag{10.13}$$

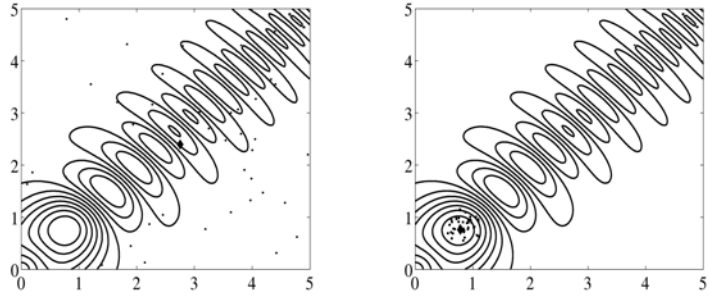


Figure 10.4: Initial locations and final locations of 40 particles after 10 iterations. The point marked with $\diamond$ corresponds to the best estimate (x_*, y_*).

or

$$\sin(2x^2) = 4x^2 \cos(2x^2), \tag{10.14}$$

which has multiple solutions. The global maximum occurs at

$$x_* = y_* \approx 0.7634. \tag{10.15}$$

It is worth pointing out that the solution $x_* = y_*$ is independent of λ as long as $\lambda > 0$.

If we use 40 particles, the new locations of these particles after 10 iterations (generations) are shown in Figure 10.4. The final optimal solution at $t = 10$ is shown on the right (with the best location marked with $\diamond$).

10.5 Implementation

The accelerated particle swarm optimization has been implemented using both Matlab and Octave. If you type the following program and save it as, say, pso_simpledemo.m, then launch Matlab or Octave and change to the directory where the file was saved. After typing in >pso_simpledemo, it will find the global optimal solution in less a minute on most modern personal computers.

```
% The Particle Swarm Optimization
% (written by X S Yang, Cambridge University)
% Usage: pso(number_of_particles,Num_iterations)
%  eg:   best=pso_demo(20,10);
%  where best=[xbest ybest zbest]  %an n by 3 matrix
%        xbest(i)/ybest(i) are the best at ith iteration

function [best]=pso_simpledemo(n,Num_iterations)
% n=number of particles
% Num_iterations=total number of iterations
if nargin<2,   Num_iterations=10;  end
if nargin<1,   n=20;          end
% Michaelewicz Function f*=-1.801 at [2.20319, 1.57049]
% Splitting two parts to avoid a long line for printing
str1='-sin(x)*(sin(x^2/3.14159))^20';
str2='-sin(y)*(sin(2*y^2/3.14159))^20';
funstr=strcat(str1,str2);
% Converting to an inline function and vectorization
f=vectorize(inline(funstr));
% range=[xmin xmax ymin ymax];
range=[0 4 0 4];
% ---------------------------------------------------
% Setting the parameters: alpha, beta
% Random amplitude of roaming particles alpha=[0,1]
% alpha=gamma^t=0.7^t;
% Speed of convergence (0->1)=(slow->fast)
beta=0.5;
% ---------------------------------------------------
% Grid values of the objective function
% These values are used for visualization only
Ngrid=100;
dx=(range(2)-range(1))/Ngrid;
dy=(range(4)-range(3))/Ngrid;
xgrid=range(1):dx:range(2);
ygrid=range(3):dy:range(4);
[x,y]=meshgrid(xgrid,ygrid);
z=f(x,y);
% Display the shape of the function to be optimized
figure(1);
surfc(x,y,z);
% ---------------------------------------------------
best=zeros(Num_iterations,3);   % initialize history
```

```
% ----- Start Particle Swarm Optimization -----------
% generating the initial locations of n particles
[xn,yn]=init_pso(n,range);
% Display the paths of particles in a figure
% with a contour of the objective function
 figure(2);
% Start iterations
for i=1:Num_iterations,
% Show the contour of the function
  contour(x,y,z,15); hold on;
% Find the current best location (xo,yo)
zn=f(xn,yn);
zn_min=min(zn);
xo=min(xn(zn==zn_min));
yo=min(yn(zn==zn_min));
zo=min(zn(zn==zn_min));
% Trace the paths of all roaming particles
% Display these roaming particles
plot(xn,yn,'.',xo,yo,'*'); axis(range);
% The accelerated PSO with alpha=gamma^t
  gamma=0.7; alpha=gamma.^i;
% Move all the particles to new locations
[xn,yn]=pso_move(xn,yn,xo,yo,alpha,beta,range);
drawnow;
% Use "hold on" to display paths of particles
hold off;
% History
best(i,1)=xo; best(i,2)=yo; best(i,3)=zo;
end   %%%%% end of iterations
% ----- All subfunctions are listed here -----
% Intial locations of n particles
function [xn,yn]=init_pso(n,range)
xrange=range(2)-range(1);
yrange=range(4)-range(3);
xn=rand(1,n)*xrange+range(1);
yn=rand(1,n)*yrange+range(3);
% Move all the particles toward (xo,yo)
function [xn,yn]=pso_move(xn,yn,xo,yo,a,b,range)
nn=size(yn,2);  %a=alpha, b=beta
xn=xn.*(1-b)+xo.*b+a.*(rand(1,nn)-0.5);
yn=yn.*(1-b)+yo.*b+a.*(rand(1,nn)-0.5);
[xn,yn]=findrange(xn,yn,range);
```

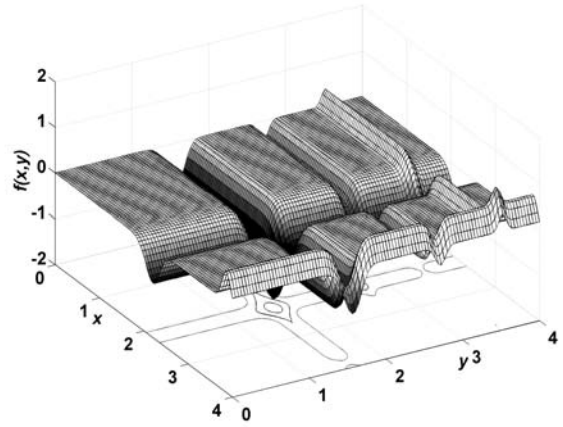


Figure 10.5: Michaelewicz function with a global minimum at about $(2.202, 1.571)$.

```
% Make sure the particles are within the range
function [xn,yn]=findrange(xn,yn,range)
nn=length(yn);
for i=1:nn,
   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
```

Let us look at the Michaelewicz function (for two independent variables)

$$f(x,y) = -\{\sin(x)[\sin(\frac{x^2}{\pi})]^{2m} + \sin(y)[\sin(\frac{2y^2}{\pi})]^{2m}\}, \quad (10.16)$$

where $m = 10$. The stationary conditions $f_x = f_y = 0$ require that

$$-\frac{4m}{\pi} x \sin(x) \cos(\frac{x^2}{\pi}) - \cos(x) \sin(\frac{x^2}{\pi}) = 0, \quad (10.17)$$

and

$$-\frac{8m}{\pi} y \sin(x) \cos(\frac{2y^2}{\pi}) - \cos(y) \sin(\frac{2y^2}{\pi}) = 0. \quad (10.18)$$

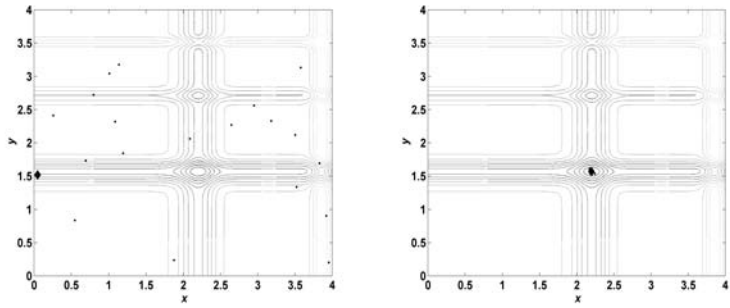


Figure 10.6: Initial locations and final locations of 20 particles after 10 iterations. The point marked with $\diamond$ corresponds to the best estimate (x_*, y_*).

The solution at $(0, 0)$ is trivial, and the minimum $f^* \approx -1.801$ occurs at about (2.202,1.571) (see Fig. 10.5).

If we run the program, we will get the global optimum after about 200 evaluations of the objective function (for 20 particles and 10 iterations). The results are shown in Fig. 10.6.

10.6 Constraints

The implementation we discussed in the previous section is for unstrained problems. For constrained optimization, there are many ways to implement the constraint equalities and inequalities. However, we will only discussed two approaches: direct implementation and transform to unconstrained optimization.

The simplest direct implementation is to check all the new particle locations to see if they satisfy all the constraints. The new locations are discarded if the constraints are not met, and new locations are replaced by newly generated locations until all the constraints are met. Then, the new solutions are evaluated using the standard PSO procedure. In this way, all the new locations should be in the feasible region, and all infeasible solutions are not selected. For example, in order to maximize $f(\boldsymbol{x})$ subjected to a constraint $g(\boldsymbol{x}) \leq 0$, the standard PSO

as discussed earlier is used, however, the new locations $\boldsymbol{x}_i$ of the n particles are checked at each iteration so that they must satisfy $g(\boldsymbol{x}_i) \leq 0$. If any $\boldsymbol{x}_i$ does not satisfy the constraint, it is then replaced by a new (different) location $\tilde{\boldsymbol{x}}_i$ which satisfies the constraint.

Alternatively, we can transform it to an unconstrained problem by using penalty method or Lagrange multipliers as discussed in Chapter 7. Using a penalty parameter $\nu \gg 1$ in our simple example here, we have the following penalty function

$$\Pi(\boldsymbol{x}, \nu) = f(\boldsymbol{x}) + \nu g(x)^2. \tag{10.19}$$

For any fixed value of ν which will determine the accuracy of the corresponding solutions, we can then optimize Π as a standard unstrained optimization problem.

There are other variations of particle swarm optimization, and PSO algorithms are often combined with other existing algorithms to produce new hybrid algorithms. In fact, it is still an active area of research with many new studies published each year.

Chapter 11

Simulated Annealing

11.1 Fundamental Concepts

Simulated annealing (SA) is a random search technique for global optimization problems, and it mimics the annealing process in material processing when a metal cools and freezes into a crystalline state with 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 cooling rate (often called annealing schedule).

The application of simulated annealing into optimization problems was pioneered by Kirkpatrick, Gelatt and Vecchi in 1983. Since then, there have been extensive studies. Unlike the gradient-based methods and other deterministic search methods which have the disadvantage of be trapped into local minima, the main advantage of the simulated annealing is its ability to avoid being trapped in local minima. In fact, it has been proved that the simulated annealing will converge to its global optimality if enough randomness is used in combination with very slow cooling.

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

lowest locations, hence the global minimum will be reached.

The basic idea of the simulated annealing algorithm is to use random search 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 cost (or 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

$$p = e^{-\frac{\delta E}{kT}}, \tag{11.1}$$

where k is the Boltzmann's constant, and 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 physics. The simplest way to link δE with the change of the objective function δf is to use

$$\delta E = \gamma \delta f, \tag{11.2}$$

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

$$p(\delta f, T) = e^{-\frac{\delta f}{T}}. \tag{11.3}$$

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

$$p = e^{-\frac{\delta f}{T}} > r, \tag{11.4}$$

it is accepted.

11.2 Choice of Parameters

Here the choice of the right 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 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 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 potential global minima. So the proper 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 cooling. For a linear cooling process, we have $T = T_0 - \beta t$ 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 as way that $T \to 0$ when $t \to t_f$ (maximum number of iterations), this usually gives $\beta = T_0/t_f$.

The geometric cooling essentially decreases the temperature by a cooling factor $0 < \alpha < 1$ so that T is replaced by αT or

$$T(t) = T_0 \alpha^t, \qquad t = 1, 2, ..., t_f. \tag{11.5}$$

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 t_f. For this reason, we will use this geometric cooling schedule. The cooling process should be slow enough to allow the system to stabilise easily. In practise, $\alpha = 0.7 \sim 0.9$ 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 stabilise and subse-

Simulated Annealing Algorithm

begin
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: $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + randn$
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 $p = \exp[-\delta f/kT] > r$
end if
Update the best $\boldsymbol{x}_*$ *and* f_*
end while
end

Figure 11.1: Simulated annealing algorithm.

quently 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 balance of 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.

11.3 SA Algorithm

The pseudo code of the simulated annealing algorithm is shown in Fig. 11.1. In order to find a suitable starting temperature T_0, we can use any information about the objective function. If we

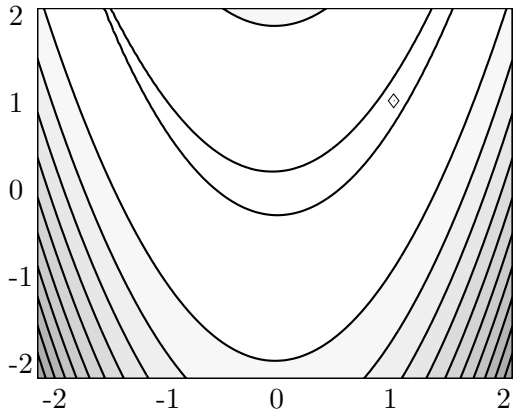


Figure 11.2: Contour of Rosenbrock function with a global minimum $f_* = 0$ at $(1, 1)$, and locations of the final 10 particles at the end of the simulated annealing.

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

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.

11.4 Implementation

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

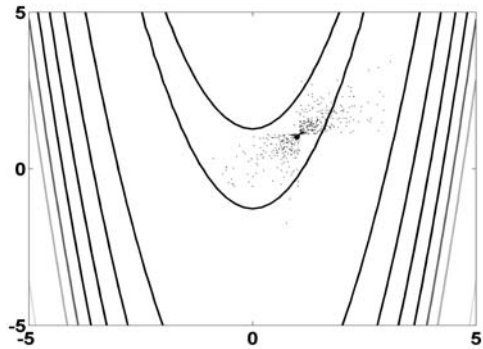


Figure 11.3: 500 evaluations during the simulated annealing. The final global best is marked with $\diamond$.

annealing using both Matlab and Octave. The implemented Matlab and Octave program is given below. Some of the simulations of the related test functions are explained in detail in the examples.

Example 11.1: For Rosenbrock's 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. 11.2). This is a standard test function and quite tough for most algorithms. However, using the program given below, we can find this global minimum easily and the 500 evaluations during the simulated annealing are shown in Fig. 11.3.

```
% Find the minimum of a function by Simulated Annealing
% Programmed by X S Yang (Cambridge University)
% Usage: sa_simpledemo
disp('Simulating ... it will take a minute or so!');
% Rosenbrock test function with f*=0 at (1,1)
fstr='(1-x)^2+100*(y-x^2)^2';
% Convert into an inline function
f=vectorize(inline(fstr));
% Show the topography of the objective function
range=[-2 2 -2 2];
```

```
xgrid=range(1):0.1:range(2);
ygrid=range(3):0.1:range(4);
[x,y]=meshgrid(xgrid,ygrid);
surfc(x,y,f(x,y));
% Initializing parameters and settings
T_init = 1.0;       % initial temperature
T_min =  1e-10;     % finial stopping temperature
                    % (eg., T_min=1e-10)
F_min = -1e+100;    % Min value of the function
max_rej=500;        % Maximum number of rejections
max_run=100;        % Maximum number of runs
max_accept = 15;    % Maximum number of accept
k = 1;              % Boltzmann constant
alpha=0.9;          % Cooling factor
Enorm=1e-5;         % Energy norm (eg, Enorm=1e-8)
guess=[2 2];        % Initial guess
% Initializing the counters i,j etc
i= 0; j = 0;
accept = 0; totaleval = 0;
% Initializing various values
T = T_init;
E_init = f(guess(1),guess(2));
E_old = E_init; E_new=E_old;
best=guess;  % initially guessed values
% Starting the simulated annealling
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)
    % Cooling according to a cooling schedule
        T = alpha*T;
        totaleval = totaleval + i;
        % reset the counters
        i = 1;  accept = 1;
    end
    % Function evaluations at new locations
      ns=guess+rand(1,2)*randn;
      E_new = f(ns(1),ns(2));
    % Decide to accept the new solution
    DeltaE=E_new-E_old;
    % Accept if improved
    if (-DeltaE > Enorm)
```

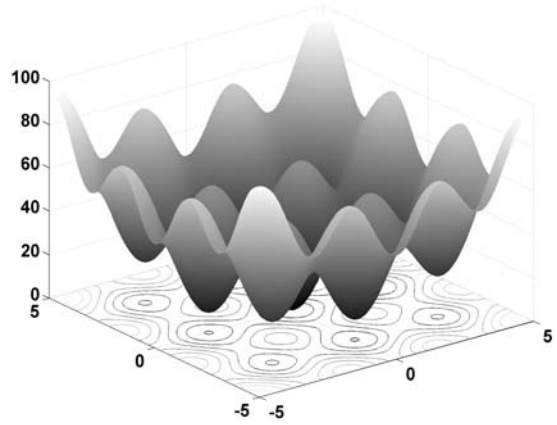


Figure 11.4: Egg crate function with a global minimum $f_* = 0$ at $(0, 0)$.

```
        best = ns; E_old = E_new;
        accept=accept+1;   j = 0;
    end
    % Accept with a small probability if not improved
    if (DeltaE<=Enorm & 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
% Display the final results
disp(strcat('Obj function :',fstr));
disp(strcat('Evaluations  :', num2str(totaleval)));
disp(strcat('Best location:', num2str(best)));
disp(strcat('Best estimate:', num2str(f_opt)));
```

Example 11.2: Using the above program, we can also simulate a more complicated function, often called the egg crate function, whose global minimum $f_* = 0$ is at $(0, 0)$. The objective function

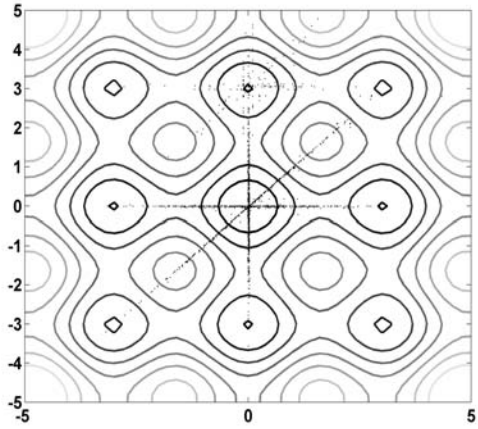


Figure 11.5: The paths of particles during simulated annealing.

for two variables is

$$f(x, y) = x^2 + y^2 + 25[\sin^2(x) + \sin^2(y)],$$

in the domain $(x, y) \in [-5, 5] \times [-5, 5]$. The landscape of the egg crate function is shown in Fig. 11.4, and the paths of particles during simulated annealing are shown in Fig. 11.5. It is worth pointing that the random generator we used in the program $\mathbf{rand(1,2)<1/2}$ leads to discrete motion along several major directions, which may improve the convergence for certain class of functions. However, for the Rosenbrock test function, this discrete approach does not work well. For continuous movement in all directions, simply use the random function $\mathbf{rand(1,2)}$ that is given in this simple demo program. It would takes about 2500 evaluations to get an accuracy with three decimal places.

Chapter 12

Multiobjective Optimization

All the optimization problems we discussed so far have only a single objective. In reality, we often have to optimize multiple objectives simultaneously. For example, we may want to improve the performance of a product while trying to minimize the cost at the time. In this case, we are dealing with multiobjective optimization problems. Many new concepts have to be introduced for multiobjective optimization.

12.1 Pareto Optimality

Any multiobjective optimization problem can generally be written as

$$\begin{aligned} &\underset{\boldsymbol{x}\in\Re^n}{\text{minimize/maximize}} && \boldsymbol{f}(\boldsymbol{x}) = [f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), ..., f_p(\boldsymbol{x})], \\ &\text{subject to} && g_j(\boldsymbol{x}) \geq 0, \quad j = 1, 2, ..., M, \qquad (12.1) \\ &&& h_k(\boldsymbol{x}) = 0, \quad k = 1, 2, ..., N, \qquad (12.2) \end{aligned}$$

where $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T$ is the vector of decision variables. In some formulations used in the literature, inequalities $g_j (j = 1, ..., N)$ can also include any equalities because an equality $\phi(\boldsymbol{x}) = 0$ can be converted into two inequalities $\phi(\boldsymbol{x}) \leq 0$ and

$\phi(\boldsymbol{x}) \geq 0$. However, for clarity, we list here the equalities and inequalities separately.

The space $\mathcal{F} = \Re^n$ spanned by the vectors of decision variables $\boldsymbol{x}$ is called the search space. The space $\mathcal{S} = \Re^p$ formed by all the possible values of objective functions is called the solution space or objective space. Comparing with the single objective function whose solution space is (at most) $\Re$, the solution space for multiobjective optimization is considerably much larger. In addition, as we know that we are dealing with multiobjectives $\boldsymbol{f}(\boldsymbol{x}) = [f_i]$, for simplicity, we can write f_i as $f(\boldsymbol{x})$ without causing any confusion.

Multiobjective optimization problems, unlike a single objective optimization problem, do not necessarily have an optimal solution that minimizes all the multiobjective functions simultaneously. Often, different objectives may conflict each other and the optimal parameters of some objectives usually do not lead to optimality of other objectives (sometimes make them worse). For example, we want first-class quality service on our holidays and at the same time we want to pay as little as possible. The high-quality service (one objective) will inevitably cost much more, and this is in conflict with the other objective (minimize cost).

Therefore, among these often conflicting objectives, we have to choose some tradeoff or a certain balance of objectives. If none of these are possible, we must choose a list of preferences so that which objectives should be achieved first. More importantly, we have to compare different objectives and make a compromise. This usually requires a formulation of a new evaluation modelling problem, and one of the most popular approaches to such modelling is to find a scalar-valued function that represents weighted combinations or preference order of all objectives. Such a scalar function is often referred to as the preference function or utility function. A simple way to construct this scalar function is to use the weighted sum

$$u(f_1(\boldsymbol{x}), ..., f_p(\boldsymbol{x})) = \sum_{i=1}^{p} \alpha_i f_i(\boldsymbol{x}), \tag{12.3}$$

where α_i are the weighting coefficients.

A point $\boldsymbol{x}_* \in \Re^n$ is called a Pareto optimal solution or non-inferior solution to the optimization problem if there is no $\boldsymbol{x} \in \Re^n$ satisfying $f_i(\boldsymbol{x}) \leq f_i(\boldsymbol{x}_*), (i = 1, 2, ..., p)$. In other words, $\boldsymbol{x}_*$ is Pareto optimal if there exists no feasible vector (of decision variables in the search space) which would decrease some objectives without causing an increase in at least one other objective simultaneously.

Unlike the single objective optimization with a single optimal solution, multiobjective optimization will lead to a set of solutions, called the Pareto optimal set $\mathcal{P}^*$, and the decision vectors $\boldsymbol{x}_*$ for the set solutions are thus called non-dominated. The set $\boldsymbol{x}_*$ in the search space that corresponds to the Pareto optimal solutions is also an efficient set in literature. The set (or plot) of the objective functions of these non-dominated decision vectors in the Pareto optimal set forms the so-called Pareto front $\mathcal{P}$ (or Pareto frontier). Now let us define the dominance of a vector $\boldsymbol{x}$ (or any vectors $\boldsymbol{u}$ and $\boldsymbol{v}$).

A vector $\boldsymbol{u} = (u_1, .., u_n)^T \in \mathcal{F}$ is said to dominate vector $\boldsymbol{v} = (v_1, ..., v_n)^T$ if and only if $u_i \leq v_i$ for $\forall i \in \{1, ..., n\}$ and $\exists i \in \{1, ..., n\} : u_i < v_i$. This 'partial less' relationship is denoted by

$$\boldsymbol{u} \prec \boldsymbol{v}, \tag{12.4}$$

which is equivalent to

$$\forall i \in \{1, ..., n\} : u_i \leq v_i \wedge \exists i \in \{1, ..., n\} : u_i < v_i. \tag{12.5}$$

Similarly, we can define another dominance relationship $\preceq$

$$\boldsymbol{u} \preceq \boldsymbol{v} \Longleftrightarrow \boldsymbol{u} \prec \boldsymbol{v} \vee \boldsymbol{u} = \boldsymbol{v}. \tag{12.6}$$

Using these notations, the Pareto front $\mathcal{P}$ can be defined as the set of non-dominated solutions so that

$$\mathcal{P} = \{\boldsymbol{s} \in \mathcal{S} \Big| \nexists\, \boldsymbol{s'} \in \mathcal{S} : \boldsymbol{s'} \prec \boldsymbol{s}\}, \tag{12.7}$$

or in term of the Pareto optimal set in the search space

$$\mathcal{P}^* = \{\boldsymbol{x} \in \mathcal{F} \Big| \nexists\, \boldsymbol{x'} \in \mathcal{F} : \boldsymbol{f(x')} \prec \boldsymbol{f(x)}\}. \tag{12.8}$$

Furthermore, a point $\boldsymbol{x}_* \in \mathcal{F}$ is called a non-dominated solution if no solution can be found that dominates it. A vector is called ideal if it contains the decision variables that correspond to the optima of objectives when each objective is considered separately.

The identification of the Pareto front is not an easy task, and it often requires a parametric analysis, say, by treating all but one objective, say, f_i in a p-objective optimization problem so that f_i is a function of f_1, ..., f_{i-1}, f_{i+1}, ..., and f_p. By maximizing the f_i when varying the values of the other $p-1$ objectives so that the solutions will trace out the Pareto front.

Example 12.1: For example, we have four Internet service providers A, B, C and D. We have two objectives to choose their service 1) as cheap as possible, and 2) higher bandwidth. They are listed below:

IP provider	Cost (£)	Bandwidth (Mb)
A	20	12
B	25	16
C	30	8
D	40	16

From the table, we know that option C is dominated by A and B because both objectives are improved (low cost and faster). Option D is dominated by B. Thus, solution C is an inferior solution, so is D. Both solutions A and B are non-inferior solutions or non-dominated solutions. However, which solution (A or B) to choose is not easy as provider A outperforms B on the first objective (cheaper) while B outperforms A on another objective (faster). In this case, we say these two solutions are incomparable. The set of the non-dominated solutions A and B forms the Pareto front which is a mutually incomparable set.

For a minimization problem with two objectives, the basic concepts of non-dominated set, Pareto front, and ideal vectors are shown in Fig. 12.1.

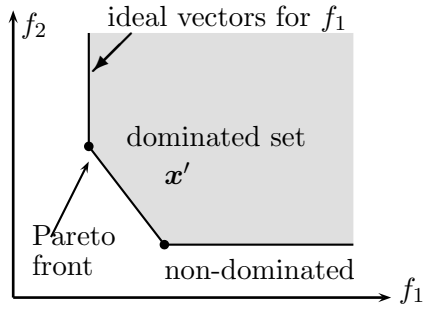


Figure 12.1: Non-dominated set, Pareto front and ideal vectors in a minimization problem with two objectives f_1 and f_2.

12.2 Weighted Sum Method

The classical three-objective functions are commonly used for testing multi-objective optimization algorithms. These functions are

$$f_1(x, y) = x^2 + (y - 1)^2, \tag{12.9}$$

$$f_2(x, y) = (x - 1)^2 + y^2 + 2, \tag{12.10}$$

$$f_3(x, y) = x^2 + (y + 1)^2 + 1, \tag{12.11}$$

where $(x, y) \in [-2, 2] \times [-2, 2]$.

Many solution algorithms intend to combine the multi-objective functions into one scalar objective using the weighted sum of objective functions

$$F(\boldsymbol{x}) = \alpha f_1(\boldsymbol{x}) + \beta f_2(\boldsymbol{x}) + ... + \gamma f_N(\boldsymbol{x}). \tag{12.12}$$

The important issue arises in assigning the weighting coefficients $(\alpha, \beta, ..., \gamma)$ because the solution is strongly dependent on the chosen weighting coefficients.

If we combine all the three functions into one $f(x, y)$ using weighted sums, we have

$$f(x, y) = \alpha f_1 + \beta f_2 + \gamma f_3, \qquad \alpha + \beta + \gamma = 1. \tag{12.13}$$

The stationary point is determined by

$$\frac{\partial f}{\partial x} = 0, \qquad \frac{\partial f}{\partial y} = 0, \tag{12.14}$$

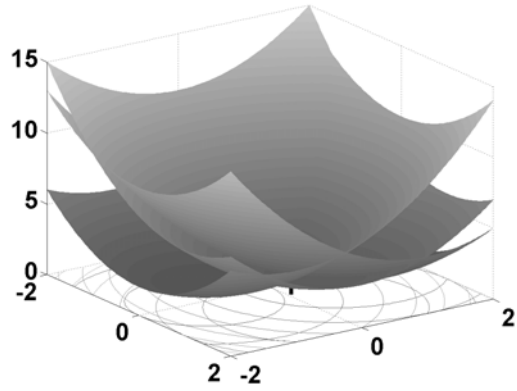


Figure 12.2: Three-objective functions with the global minimum at $x_* = \beta, y_* = \alpha - \gamma$.

which lead to

$$2\alpha + 2\beta(x-1) + 2\gamma = 0, \tag{12.15}$$

and

$$2\alpha(y-1) + 2\beta y + 2\gamma(y+1) = 0. \tag{12.16}$$

The solutions are

$$x_* = \beta, \qquad y_* = \alpha - \gamma. \tag{12.17}$$

This implies that $x_* \in [0,1]$ and $y_* \in [-1,1]$. Consequently, $f_1 \in [0,5], f_2 \in [2,4]$ and $f_3 \in [1,6]$. In addition, the solution or the optimal location varies with the weighting coefficients α, β and γ. In the simplest case $\alpha = \beta = \gamma = 1/3$, we have

$$x_* = \frac{1}{3}, \qquad y_* = 0. \tag{12.18}$$

This location is marked with a solid bar in Figure 12.2.

Now the original multiobjective optimization problem has been transformed into a single objective optimization problem. Thus, the solution methods for solving single objective problems are all valid. For example, we can use the particle swarm

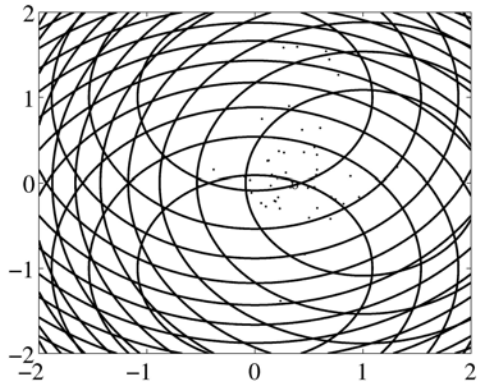


Figure 12.3: Final locations of 40 particles after 5 iterations. The optimal point is at $(1/3, 0)$ marked with $\circ$.

to find the optimal for given parameters α, β and γ. Figure 12.3 shows the final locations of 40 particles at $t = 5$ iterations. We can see that the particles converge towards the true optimal location marked with $\circ$.

However, there is an important question to be answered. The combined weighted sum transforms the optimization problem into a single objective, this is not necessarily equivalent to the original multiobjective problem because the extra weighting coefficients could be arbitrary, while the final solutions still depend on these coefficients. Furthermore, there are so many ways to construct the weighted sum function and there is not any easy guideline to choose which form is the best for a given problem. When there is no rule to follow, the simplest choice obviously is to use the linear form. But there is no reason why the weighted sum should be linear. In fact, we can use other combinations such as the following quadratic weighted sum

$$\Pi(\boldsymbol{x}) = \sum_{i=1}^{N} \alpha_i f_i^2(\boldsymbol{x}) = \alpha_1 f_1^2(\boldsymbol{x}) + ... + \alpha_N f_N^2(\boldsymbol{x}), \qquad (12.19)$$

and the others.

Another important issue is that how to choose the weighting coefficients as the solutions depend on these coefficients. The choice of weighting coefficients is essentially to assign a preference order by the decision maker to the multiobjectives. This leads to a more general concept of utility function (or preference function) which reflects the preference of the decision maker(s).

12.3 Utility Method

The weighted sum method is essentially a deterministic value method if we consider the weighting coefficients as the ranking coefficients. This implicitly assumes that the consequence of each ranking alternative can be characterized with certainty. This method can be used to explore the implications of alternative value judgement. Utility method, on the other hand, considers uncertainty in the criteria values for each alternative, which is a more realistic method because there is always some degree of uncertainty about the outcome of a particular alternative.

Utility (or preference) function can be associated with risk attitude or preference. For example, if you are offered a choice between a guaranteed £500 and a 50/50 chance of zero and £1000. How much are you willing to pay to take the gamble? The expected payoff of each choice is £500 and thus it is fair to pay $0.5 \times 1000 + (1 - 0.5) \times 0 = £500$ for such a gamble. A risk-seeking decision maker would risk a lower payoff in order to have a chance to win a higher prize, while a risk-averse decision maker would be happy with the safe choice of £500.

For a risk-neutral decision maker, the choice is indifferent between a guaranteed £500 and the 50/50 gamble since both choices have the same expected value of £500. In reality, the risk preference can vary from person to person and may depend on the type of problem. The utility function can have many forms, and one of the simplest is the exponential utility (of

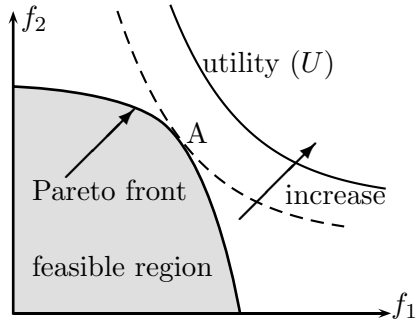


Figure 12.4: Finding the Pareto solution with maximum utility in a maximization problem with two objectives.

representing preference)

$$u(x) = \frac{1 - e^{-(x-x_a)/\rho}}{1 - e^{-(x_b-x_a)/\rho}}, \tag{12.20}$$

where x_a and x_b are the lowest and highest level of x, and ρ is called the risk tolerance of the decision maker.

The utility function defines combinations of objective values $f_1, ..., f_p$ which a decision maker finds equally acceptable or indifference. So the contours of the constant utility are referred to as the indifference curves. The optimization now becomes the maximization of the utility. For a maximization problem with two objectives f_1 and f_2, the idea of the utility contours (indifference curves), Pareto front and the Pareto solution with maximum utility (point A) are shown in Fig. 12.4. When the utility function touches the Pareto front in the feasible region, it then provides a maximum utility Pareto solution (marked with A).

For two objectives f_1 and f_2, the utility function can be constructed in different ways. For example, the combined product takes the following form

$$U(f_1, f_2) = k f_1^{\alpha} f_2^{\beta}, \tag{12.21}$$

where α and β are non-negative exponents and k a scaling

factor. The aggregated utility function is given by

$$U(f_1, f_2) = \alpha f_1 + \beta f_2 + [1 - (\alpha + \beta)] f_1 f_2. \tag{12.22}$$

There are many other forms. The aim of utility function constructed by the decision maker is to form a mapping $U : \Re^p \mapsto \Re$ so that the total utility function has a monotonic and/or convexity properties for easy analysis. It will also improve the quality of the Pareto solution(s) with maximum utility. Let us look at a simple example.

Example 12.2: For the simple two-objective optimization problem:

$$\underset{(x,y)\in\Re^2}{\text{maximize}}\, f_1(x, y) = x + y, \qquad f_2(x, y) = x,$$

subject to

$$x + \alpha y \leq 5, \qquad x \geq 0, \;\; y \geq 0,$$

where $0 < \alpha < 1$. Let us use the simple utility function

$$U = f_1 f_2,$$

which combines the two objectives. The line connecting the two corner points $(5, 0)$ and $(0, 5/\alpha)$ forms the Pareto front (see Fig. 12.5). It is easy to check that the Pareto solution with maximum utility is $U = 25$ at $A(5, 0)$ when the utility contours touch the Pareto front with the maximum possible utility.

The complexity of multiobjective optimization makes the construction of the utility function a difficult task as it can be constructed in many ways. A general and yet widely-used utility function is often written in the following additive form

$$\underset{\boldsymbol{x}\in\Re^n}{\text{maximize}} U(f_k(\boldsymbol{x})) = \sum_{j=1}^{K} \pi_j \sum_{i=1}^{p} \alpha_i u_{ijk}, \tag{12.23}$$

where $U(f_k)$ is the expected utility of alternative k. p is the number of objectives and K is the number of possible scenarios. π_j is the probability assigned to scenario j. u_{ijk} is the value of a single criterion utility function for objective i, scenario j and alternative k.

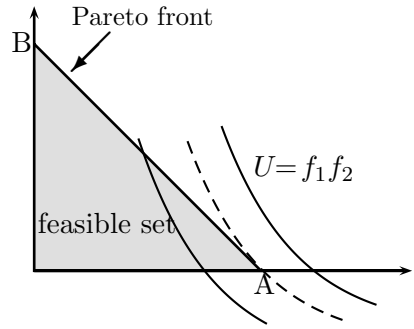


Figure 12.5: Pareto front is the line connecting A$(5, 0)$ and B$(0, 5/\alpha)$. The Pareto solution with maximum utility is $U_* = 25$ at point A.

12.4 Metaheuristic Search

So far, we have seen that finding solutions of multiobjective optimization is usually difficult, even by the simple weighted sum method and utility function. However, there are other promising methods that work well for multiobjective optimization problems, especially the metaheuristic methods such as simulated annealing and particle swarm optimization.

Here we will use the particle swarm optimization to find solutions for multiobjective optimization problems. The basic idea is to modify the standard PSO algorithms so that they can deal with the multiobjectives at the same time. This is better demonstrated by an example.

From the multimodal test function (10.8), we know that it has a single global maximum. If we extend its domain to a large set, then it will have two global maxima. Now we have

$$f(x, y) = \frac{\sin(x^2 + y^2)}{\sqrt{x^2 + y^2}} e^{-\lambda(x-y)^2}, \ (x, y) \in [-5, 5] \times [-5, 5],$$

where $\lambda > 0$. For $\lambda = 0.05$, the function is shown in Fig. 12.6. This function has two equal global maxima at $(0.7634, 0.7634)$ and $(-0.7634, -0.7634)$. If we run the standard PSO, it will only find one of the two global optima. In order to find all

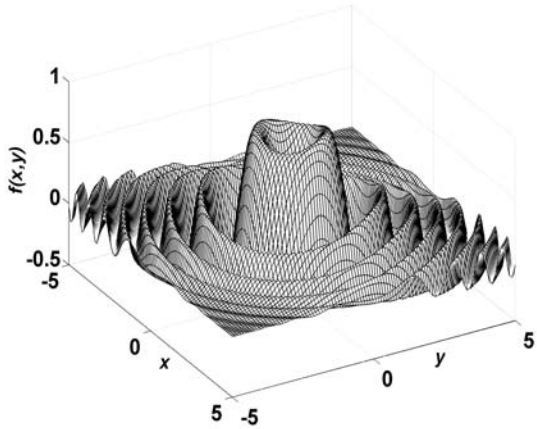


Figure 12.6: Multimodal function with two equal global maxima.

the (two) global optima, we have to modify the PSO. In fact, there are many ways to do this and one of the simplest way to achieve this is to use the recursive method.

In the recursive method, we call the standard PSO many times. As the initial starting point is a random guess, it will find one optimum each time we call it. If we call the PSO recursively, statistically speaking, all the optima will be founded. For our test function here, we called the accelerated PSO for 1000 times. It found one peak at (0.7634,0.7634) in 497 runs, the other peak in 498 runs, and trapped at local peaks in 5 runs. Two snapshots of the 1000 runs were shown in Fig. 12.7 so that all optima are found statistically.

The implementation of this recursive PSO using Matlab and Octave is given below. Since it is a recursive process, it will take a few minutes (depending on the speed of the computer).

```
% The Recursive PSO for Multiobjective Optimization
% (written by X S Yang, Cambridge University)
% Usage: pso_multi(number_of_runs)
%  eg:   best=pso_multi(100);
```

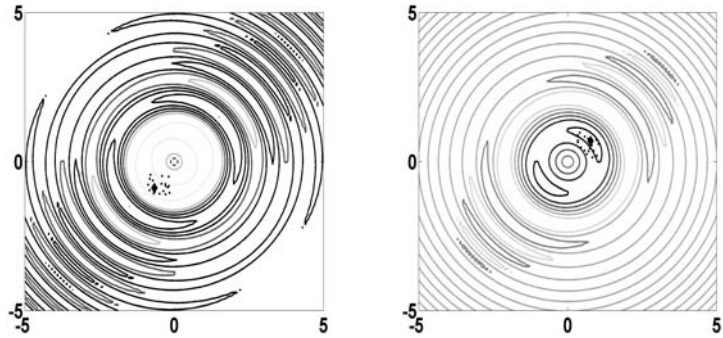


Figure 12.7: Recursive PSO runs converge to two global peaks.

```
%        where best <=> locations of global optima

function [best]=pso_multi(Num_run)
% Num_run=number of recursive iterations (=10 default)
if nargin<1,   Num_run=10;     end
disp('Running PSO recursively!');
disp('Please wait for a few minutes ...');
n=20;           % number of particles;
Num_steps=10;   % number of pseudo time steps
% This function has two global optima f*=0.851
% at (0.7634,0.7634) and (-0.7634,-0.7634).
fstr='sin(x^2+y^2)/sqrt(x^2+y^2)*exp(-0.05*(x-y)^2)'
% Converting to an inline function
f=vectorize(inline(fstr));
% range=[xmin xmax ymin ymax];
range=[-5 5 -5 5];
% ------------------------------------------------
% Grid values of the objective function
% These values are used for visualization only
Ngrid=100;
dx=(range(2)-range(1))/Ngrid;
dy=(range(4)-range(3))/Ngrid;
xgrid=range(1):dx:range(2);
```

```
ygrid=range(3):dy:range(4);
[x,y]=meshgrid(xgrid,ygrid);
z=f(x,y);
% Display the shape of the function to be optimized
surfc(x,y,z); drawnow;
% Run the PSO recursively for, say, 10 times
for i=1:Num_run,
    best(i,:)=pso(f,range,n,Num_steps);
end
% ------------------------------------------------
% Standard Accelerated PSO (for finding maxima)
function [best]=pso(f,range,n,Num_steps)
% here  best=[xbest ybest fbest]
% Speed of convergence (0->1)=(slow->fast)
beta=0.5;
% ----- Start Particle Swarm Optimization --------
% generating the initial locations of n particles
[xn,yn]=init_pso(n,range);
% Iterations as pseudo time
for i=1:Num_steps,
% Find the current best location (xo,yo)
  zn=f(xn,yn);
  zn_max=max(zn);
  xo=max(xn(zn==zn_max));
  yo=max(yn(zn==zn_max));
  zo=max(zn(zn==zn_max));
% Accelerated PSO with randomness: alpha=gamma^t
  gamma=0.7; alpha=gamma.^i;
% Move all particle to new locations
  [xn,yn]=pso_move(xn,yn,xo,yo,alpha,beta,range);
end   %%%%% end of iterations
% Return the finding as the current best
best(1)=xo; best(2)=yo; best(3)=zo;
% ------------------------------------------------
% All subfunctions are listed here
% Intial locations of particles
function [xn,yn]=init_pso(n,range)
```

```
  xrange=range(2)-range(1);
  yrange=range(4)-range(3);
  xn=rand(1,n)*xrange+range(1);
  yn=rand(1,n)*yrange+range(3);
% Move all the particles toward to (xo,yo)
function [xn,yn]=pso_move(xn,yn,xo,yo,a,b,range)
  nn=size(yn,2);   %%%%% a=alpha, b=beta
  xn=xn.*(1-b)+xo.*b+a.*(rand(1,nn)-0.5);
  yn=yn.*(1-b)+yo.*b+a.*(rand(1,nn)-0.5);
  [xn,yn]=findrange(xn,yn,range);
% Make sure the particles are inside the range
function [xn,yn]=findrange(xn,yn,range)
nn=length(yn);
for i=1:nn,
   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
```

12.5 Other Algorithms

There are other powerful algorithms that we have not addressed in this book. These include genetic algorithms, virtual bee algorithms, harmony search, random-restart hill climbing, dynamic programming, stochastic optimization, evolution strategy and many other evolutionary algorithms. For example, genetic algorithms (and their stochastic various variants) have been used to solve many practical optimization problems such as scheduling problems and engineering design problems. Readers interested in these modern techniques can refer to more advanced literature.

The optimization algorithms we have discussed in this book are mainly for the optimization problems with explicit objective functions. However, in reality, it is often difficult to quantify what we want to achieve, but we still try to optimize certain things such as the degree of enjoyment of a quality service on

our holidays. In other cases, it is not possible and/or there is no explicit form of objective function. For example, we want to design an aircraft wing so that it is most aerodynamically efficient. If we know nothing about the modern aircraft design, suppose we have to start from scratch, what shape it should be and how to quantify the efficiency (lift or some ratio of power to lift)? What is the explicit form of the objective function? This is a very difficult task. One approach is to try various shapes by carrying out experimental tests of these various shapes. Alternatively, we can use computer simulations (*e.g.*, computational fluid dynamics) for a given shape. Starting from an initial shape (say, an ellipsoid), what is the new shape we should generate? We usually have to run many iterations of computer simulations, changing design (both computer models and physical models) and evaluating their performance (objectives) again and again.

Whatever the objectives are, we have to evaluate the objectives many times. In most cases, the evaluations of the objective functions consume a lot of computational power (which costs money) and design time. Any efficient algorithm that can reduce the number of objective evaluations will save both time and money. Although, we have mainly focused on the optimization algorithms for objectives which are explicitly known, however, these algorithms will still be applicable to the cases where the objectives are not known explicitly. In most cases, certain modifications are required to suit a particular application. This is an exciting area of active research, and more publications are emerging each year.

Bibliography

[1] Abramowitz M. and Stegun I. A., *Handbook of Mathematical Functions*, Dover Publication, (1965).

[2] Adamatzky A., Teuscher C., *From Utopian to Genuine Unconventional Computers*, Luniver Press, (2006).

[3] Arfken G., *Mathematical Methods for Physicists*, Academic Press, (1985).

[4] Atluri S. N., *Methods of Computer Modeling in Engineering and the Sciences*, Vol. I, Tech Science Press, (2005).

[5] Bonabeau E., Dorigo M., Theraulaz G., *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, (1999)

[6] Bonabeau E. and Theraulaz G., Swarm smarts, *Scientific Americans*. March, 73-79 (2000).

[7] Bridges D. S., *Computatability*, Springer, New York, (1994).

[8] Chatterjee A. and Siarry P., Nonlinear inertia variation for dynamic adapation in particle swarm optimization, *Comp. Oper. Research*, **33**, 859-871 (2006).

[9] Coello C. A., Use of a self-adaptive penalty approach for engineering optimization problems, Computers in Industry, **41**, 113-127 (2000).

[10] Courant R. and Hilbert, D., *Methods of Mathematical Physics*, 2 volumes, Wiley-Interscience, New York, (1962).

[11] Davis M., *Computability and Unsolvability*, Dover, (1982).

[12] De Jong K., *Analysis of the Behaviour of a Class of Genetic Adaptive Systems*, PhD thesis, University of Michigan, Ann Arbor, (1975).

[13] Deb K., An efficient constraint handling method for genetic algorithms, Comput. Methods Appl. Mech. Engrg., **186**, 311-338 (2000).

[14] Deb. K., *Optimization for Engineering Design*: Algorithms and Examples, Prentice-Hall, New Delhi, (1995).

[15] Dorigo M., *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, Italy, (1992).

[16] Dorigo M. and Stützle T., *Ant Colony Optimization*, MIT Press, Cambridge, (2004).

[17] El-Beltagy M. A., Keane A. J., A comparison of various optimization algorithms on a multilevel problem, *Engin. Appl. Art. Intell.*, **12**, 639-654 (1999).

[18] Engelbrecht A. P., *Fundamentals of Computational Swarm Intelligence*, Wiley, (2005).

[19] Flake G. W., *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*, Cambridge, Mass.: MIT Press, (1998).

[20] Forsyth A. R., *Calculus of Variations*, Dover (1960).

[21] Fowler A. C., *Mathematical Models in the Applied Sciences*, Cambridge University Press, (1997).

[22] Gardiner C. W., *Handbook of Stochastic Methods*, Springer, (2004).

[23] Gershenfeld N., *The Nature of Mathematical Modeling*, Cambridge University Press, (1998).

[24] Gill P. E., Murray W., and Wright M. H., *Practical optimization*, Academic Press Inc, (1981).

[25] Glover F., Heuristics for Integer Programming Using Surrogate Constraints, *Decision Sciences*, **8**, 156-166 (1977).

[26] Glover F., Tabu Search - Wellsprings and Challenges, *Euro. J. Operational Research*, **106**, 221-225 (1998).

[27] Glover F. and Laguna M., *Tabu Search*, Kluwer Academic Publishers, (1997).

[28] Goldberg D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, Mass.: Addison Wesley (1989).

[29] Goodman R., *Teach Yourself Statistics*, London, (1957).

[30] Holland J., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Anbor, (1975).

[31] Jaeggi D., Parks G. T., Kipouros T., Clarkson P. J.: A multi-objective Tabu search algorithm for constrained optimization problem, *3rd Int. Conf. Evolutionary Multi-Criterion Optimization*, Mexico, **3410**, 490-504 (2005).

[32] Jeffrey A., *Advanced Engineering Mathematics*, Academic Press, (2002).

[33] John F., *Partial Differential Equations*, Springer, New York, (1971).

[34] Keane A. J., Genetic algorithm optimization of multi-peak problems: studies in convergence and robustness, *Artificial Intelligence in Engineering*, **9**, 75-83 (1995).

[35] Kennedy J. and Eberhart R. C.: Particle swarm optimization. *Proc. of IEEE International Conference on Neural Networks*, Piscataway, NJ. pp. 1942-1948 (1995).

[36] Kennedy J., Eberhart R., Shi Y.: *Swarm intelligence*, Academic Press, (2001).

[37] Korn G. A. and Korn T. M., *Mathematical Handbook for Scientists and Engineers*, Dover Publication, (1961).

[38] Kirkpatrick S., Gelatt C. D., and Vecchi M. P., Optimization by simulated annealing, *Science*, **220**, No. 4598, 671-680 (1983).

[39] Kreyszig E., *Advanced Engineering Mathematics*, 6th Edition, Wiley & Sons, New York, (1988).

[40] Kuhn H. W. and Tucker A. W., Nonlinear programming, *Proc. 2nd Berkeley Symposium*, pp. 481-492, University of California Press, (1951).

[41] Marco N., Lanteri S., Desideri J. A., Périaux J.: A parallel genetic algorithm for multi-objective optimization in CFD, in: *Evolutionary Algorithms in Engineering and Computer Science* (eds Miettinen K. et al), Wiley & Sons, (1999).

[42] Matlab info, http://www.mathworks.com

[43] Michaelewicz, Z., *Genetic Algorithm + Data Structure = Evolution Progamming*, New York, Springer, (1996).

[44] Mitchell, M., *An Introduction to Genetic Algorithms*, Cambridge, Mass: MIT Press, (1996).

[45] Octave info, http://www.octave.org

[46] Pearson C. E., *Handbook of Applied Mathematics*, 2nd Ed, Van Nostrand Reinhold, New York, (1983).

[47] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P., *Numerical Recipe in C++: The Art of Scientific Computing*, Cambridge University Press, (2002).

[48] Riley K. F., Hobson M. P., and Bence S. J., *Mathematical Methods for Physics and Engineering*, 3rd Edition, Cambridge University Press (2006).

[49] Sawaragi Y., Nakayama H., Tanino T., *Theory of Multiobjective Optimization*, Academic Press, (1985).

[50] Sirisalee P., Ashby M. F., Parks G. T., and Clarkson P. J.: Multi-criteria material selection in engineering design, *Adv. Eng. Mater.*, **6**, 84-92 (2004).

[51] Selby S. M., *Standard Mathematical Tables*, CRC Press, (1974).

[52] Smith D. R., *Variation Methods in Optimization*, New York, Dover, (1998).

[53] Spall J. C., *Introduction to Stochastic Search and optimization: Estimation, Simulation, and Control*, Wiley, Hoboken, NJ, (2003).

[54] Swarm intelligence, http://www.swarmintelligence.org

[55] Weisstein E. W., http://mathworld.wolfram.com

[56] Wikipedia, http://en.wikipedia.com

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

[58] Wylie C. R., *Advanced Engineering Mathematics*, Tokyo, (1972).

[59] Yang X. S., Engineering optimization via nature-inspired virtual bee algorithms, Lecture Notes in Computer Science, **3562**, 317-323 (2005).

[60] Yang X. S., Biology-derived algorithms in engineering optimization (Chapter 32), in *Handbook of Bioinspired Algorithms*, edited by Olarius S. & Zomaya A., Chapman & Hall / CRC, (2005).

[61] Yang X. S., Lees J. M., Morley C. T.: Application of virtual ant algorithms in the otpimization of CFRP shear strengthened precracked structures, *Lecture Notes in Computer Sciences*, **3991**, 834-837 (2006).

[62] Yang X. S., *Applied Engineering Mathematics*, Cambridge International Science Publishing, (2007).

Index