Some Topics on Monte Carlo and Quasi-Monte Carlo Methods

Ph.D. dissertation by Guiyuan Lei

Adviser: Prof. Wang Xinghua

Computational Mathematics

Zhejiang University

Thank you!

I'd like to thank my adviser Prof. Wang Xinghua. He conducted me into the field of Monte Carlo and quasi-Monte Carlo methods. Through my study career in the field, he gave me much help.

I also thank the following members of the group of computational mathematics. I learn much knowledge from the seminars among the group.

Wang Xinghua, Li Chong, Jiang Jinsheng, Zheng Shiming, Cheng Xiaoliang, Han Dangfu, Wu Qingbiao, Huang Zhengda, Chen Mingfei, Ye Xingde, Zhu Jianxin,Sun Fangyu, Yang Shijun, Liang Kewei, Li Daming, Liang Xianhong, Wang Heyu, Mi Xiangjiang, Jin Zhongqiu, Cui Feng and Xie Congcong

There are many other individuals help me apply the methods to practical problems. Jin Jianqiu introduced the work chance of prediction programming to me. Prof. Anders Karlsson discussed the inverse problem of light transport in tissue with me.

Lastly, I thank my family and friends. They put in a big effort to help me realize the dream of becoming a researcher. My husband Dr. Jiangping He encourages me to work hard. We have stayed up late together when I prepared my first paper. That was a wonderful time. He also helps me prepare some of the figures for this dissertation.

Preface

Monte Carlo methods are powerful tools for evaluating the properties of complex, many-body systems, as well as nondeterministic processes. J. Dongarra and F. Sullivan[1][2] published a list of "The top ten algorithms of the century" in "Computing in Science & Engineering". The Monte Carlo method is listed in the 10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century.

The Monte Carlo method or Metropolis algorithm is devised by J. von Neumann, S. Ulam, and N. Metropolis at the end of the Second World War to study the diffusion of neutrons in fissionable material. The name "Monte Carlo", chosen because of the extensive use of random numbers in the calculation, was coined by Metropolis in 1947 and used in the title of a paper[3] by N. Metropolis and S. Ulam in 1949. The method began as a technique for attacking specific problems in numerical simulations of physical systems, and interest in it grew slowly at first. But later, the subject exploded as the scope of applications broadened in many surprising directions, including function minimization, computational geometry, and combinatorial counting. Today, topics related to the Metropolis algorithm constitute an entire field of computational science supported by a deep theory and having applications ranging from physical simulations to the foundations of computational complexity. By introducing the quasi-Monte Carlo sequence, the Monte Carlo method has already been developed to quasi-Monte Carlo method.

We are often confronted with problems that have an enormous number of dimensions or a process that involves a path with many possible branch points, each of which is governed by some fundamental probability occurrence. The solutions are not exact in a rigorous way, because we randomly sample the problem. However, it is possible to achieve nearly exact results using a relatively small number of samples compared to the problem's dimensions. Indeed, Monte Carlo and quasi-Monte Carlo methods are the only practical choice for evaluating problems of high dimensions though it may be time consuming. Luckily, with the development of modern computer technology at very fast speed, we can calculate more complex systems. So

the Monte Carlo and quasi-Monte Carlo methods will be more promising in the 21th century. So I choose Monte Carlo and quasi-Monte Carlo as my study field.

If you have any advice, please let me know.

Guiyuan Lei <guiyuanlei@hotmail.com>

Lund, March, 2003

Abstract

I have studied a couple of topics on Monte Carlo and quasi-Monte Carlo methods. This dissertation covers its applications in integration, optimization and simulation.

Chapter 1 and 2 are the basic knowledge to use Monte Carlo and quasi-Monte Carlo methods. Chapter 1 presents the error bounds of Monte Carlo and quasi-Monte Carlo integration methods. By comparing these two methods, we show the advantages of quasi-Monte Carlo method. We also introduce the standard quasi-Monte Carlo random search for optimization. The last but not least application is Metropolis algorithms which is the origin of Monte Carlo method. Because the random numbers generators are the key of Monte Carlo methods and quasi-Monte Carlo methods. Chapter 2 describes the pseudo-random number generators and quasi-random number generators. How to generate non-uniform random number from its distributed function is also introduced.

Chapter 3 introduces B-spline smoothed rejection sampling method. The standard rejection sampling method which is introduced in chapter 2 is closely related to the problem of quasi-Monte Carlo integration of characteristic functions, whose accuracy may be lost due to the discontinuity of the characteristic functions. We use B-spline smoothing technique to smooth the characteristic function without changing the integral quantity and get a differentiable weight function. The method considerably improves the quality of sampling points. We apply the B-spline smoothed rejection sampling method to importance sampling. Numerical experiments show that the error size $O(N^{-1})$ is regained by using the B-spline smoothed rejection method for quasi-Monte Carlo estimate. The error bound of Monte Carlo method using B-spline smoothed importance sampling is also better than that of the standard Monte Carlo method. So the B-spline smoothed rejection sampling method is indirectly proved to be superior to the standard rejection sampling method.

Chapter 4 is about the Monte Carlo integration. We get a theoretical error of the fine antithetic variables Monte Carlo(FAMC) method for

multidimensional integration. The error size of FAMC is $O(N^{-(\frac{1}{2}+\frac{2}{s})})$ for functions having second continuous derivative, where s is the dimension of the integrand. We also give the theoretical error result of antithetic variable Monte Carlo(AMC) method for multi-variable functions whose degree is no more than two. The constant before $O(N^{-\frac{1}{2}})$ is less than that of the MC method. We realize the parallel algorithm in C for the FAMC and AMC methods. The results of the numerical experiments coincide with the theoretical results very well.

Chapter 5 introduces adaptive monte carlo method(AQMC) for global optimization. AQMC algorithm progresses in the nondifferentiable optimization. First, we develop the local search such that the search direction, search radius and number of search points are adjusted according to the previous search result. Second, we introduce the ideas of population and generate new individuals according to population evolution degree. Because the search procedure will be adjusted according to the previous result, the method not only speeds up the random search but also balances the global and local demands (adaptive equalization).

Chapter 6 combines the genetic programming with AQMC optimization method to solve the prediction problems. There are many complex systems in real life. In order to analyze, design and predict the system, we often want to model the dynamic systems of ordinary differential equations according to the observed data. We use genetic programming to optimize the the right hand functions of the ordinary differential equations. Adaptive quasi-Monte Carlo optimization methods are used to optimize the coefficients of the functions. The program for the prediction of electrical power consumption of Hangzhou city shows that the hybrid method is powerful.

In Chapter 7, we combine the Monte Carlo simulation and optimization. We first introduce the Monte Carlo simulation of light transport in tissue, explain how to generate the random number according to the practical problems using the transform method introduced in chapter 2. Then use the AQMC method to solve the inverse problem of light transport. We further discuss how to balance the global and local search demands in practical problems.

The C codes of some programs are given in the appendix.

Contents

List of Figures

List of Tables

Chapter 1

Preliminary on Monte Carlo and quasi-Monte Carlo methods

Monte Carlo and quasi-Monte Carlo methods are versatile and widely used numerical methods. They have been studied for many years. The most common applications are for evaluating integrals. Monte Carlo and quasi-Monte Carlo can also be used in optimization and simulation. Because they are simple and direct, Monte Carlo and quasi-Monte Carlo are easy to use . They are also robust, since their accuracy depend on only the crudest measure of the complexity of the problems. This chapter presents some basic knowledge about Monte Carlo and quasi-Monte Carlo methods.

1.1 Monte Carlo integration

A review of Monte Carlo methods for integration problems was presented by Caflisch[4]. The integral of a Lebesgue integrable function $f(\mathbf{x})$ can be expressed as the average or *expectation* of the function f evaluated at a random location. Consider an integral on the unit cube $I^s = [0,1]^s$ in s dimensions. Then

$$I = \mathrm{E}[f(\mathbf{x})] = \int_{I^s} f(\mathbf{x})\mathrm{d}\mathbf{x} = \bar{f} \tag{1.1}$$

in which $\mathbf{x}$ is a uniformly distributed vector in the unit cube.

The crude Monte Carlo quadrature formula is based on the probabilistic interpretation of an integral. Consider a sequence ξ_n sampled from the uniform distribution. Then an empirical approximation to the expectation

is

$$M = \frac{1}{N}\sum_{k=1}^{N} f(\xi_k) \tag{1.2}$$

According to the Strong Law of Large Numbers[5], this approximation is convergent with probability one: that is,

$$\lim_{N\to\infty} M \to I. \tag{1.3}$$

In addition, it is unbiased, which means that the average of M is exactly I for any N; that is

$$\mathrm{E}[M] = I, \tag{1.4}$$

in which the average is over the choice of the points ξ_k.

In general, define the Monte Carlo integration error

$$\epsilon_N = I - M \tag{1.5}$$

so that the bias is $E[\epsilon_N]$ and the root mean square error($rmse$) is

$$\mathrm{E}[\epsilon_N^2]^{1/2}. \tag{1.6}$$

1.1.1 Error bounds

The Central Limit Theorem(CLT)[5] describes the size and statistical properties of Monte Carlo integration errors.

Theorem 1.1.1 *For N large,*

$$\epsilon_N \approx \sigma N^{-1/2}\nu \tag{1.7}$$

in which ν is a standard normal ($N(0,1)$) random variable and the constant $\sigma = \sigma[f]$ is the square root of the variance of f; that is,

$$\sigma[f] = \Big(\int_{I^s} (f(\mathbf{x}) - I)^2 \mathrm{d}\mathbf{x}\Big)^{1/2}. \tag{1.8}$$

A more precise statement is that

$$\begin{aligned} \lim_{N\to\infty} \mathrm{Prob}(a < \tfrac{\sqrt{N}}{\sigma}\epsilon_N < b) &= \mathrm{Prob}(a < \nu < b) \\ &= \int_a^b (2\pi)^{-1/2} \mathrm{e}^{-x^2/2} \mathrm{d}x. \end{aligned} \tag{1.9}$$

This says that the error in Monte Carlo integration is of size $O(N^{-1/2})$with a constant that just the variance of the integrand f. Moreover, the statistical distribution of the error is approximately a normal random variable. In contrast to the usual results of numerical analysis, this is a probabilistic result. It does not provide an absolute upper bound on the error; rather it says that the error is of a certain size with some probability. On the other hand, this result is an equality, so that the bounds it provides are tight.

1.1.2 Comparison to grid-based methods

Most people who see Monte Carlo for the first time are surprised that it is a viable method. How can a random array be better than a grid? There are several ways to answer this question. First, compare the convergence rate of Monte Carlo with that of a grid-based integration method such as Simpson's rule. The convergence rate for grid-based quadrature is $O(N^{-k/s})$ for an order k method in dimension s. On the other hand, the Monte Carlo convergence rate is $O(N^{-1/2})$ independent of dimension. So Monte Carlo beats a grid in high-dimension s, if

$$k/s < 1/2$$

However, for an analytic function on a periodic domain, the value of k is infinite, so that this simple explanation fails. A more realistic explanation for the robustness of Monte carlo is that it is practically impossible to lay down a grid in high dimension. The simplest cubic grid in s dimension requires at least 2^s points. For $s = 20$, which is not particularly large, this requires more than a million points. Moreover, it is practically impossible to refine a grid in a high dimension, since a refinement requires increasing the number of points by factor 2^s. In contrast to these difficulties for a grid in high dimension, the accuracy of Monte Carlo quadrature is nearly independent of dimension and each additional point added to the Monte Carlo quadrature formula provides an incremental improvements in its accuracy. To be sure, the value of N at which the $O(N^{-1/2})$ error estimate becomes valid is difficult to predict, but experience shows that, for problems of moderate complexity in moderate dimension(for instance $s = 20$), the $O(N^{-1/2})$ error size is typically attained for moderate values of N.

1.2 Quasi-Monte Carlo integration

We recall that Monte Carlo integration with N random nodes the absolute value of the error has the average order of magnitude $O(N^{-1/2})$. Clearly, there exist sets of N nodes for which the absolute value of the error is not larger than the average. If we could construct such sets of nodes explicitly,

this would already be a useful methodological advance. The quasi-Monte Carlo method for numerical integration aims much higher, as it seeks to construct sets of nodes that perform significantly better than average. The integration rules for the quasi-Monte Carlo method are taken from the appropriate Monte Carlo estimate. For instance, for the unit cube integration domain $I^s = [0,1]^s$, we have the quasi-Monte Carlo approximation

$$\int_{I^s} f(\mathbf{x})\mathrm{d}\mathbf{x} \approx \frac{1}{N}\sum_{k=1}^{N} f(\mathbf{x}_k) \tag{1.10}$$

which formally looks like the Monte Carlo estimate but is now used with deterministic nodes $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N \in I^s$. These nodes should be chosen judiciously so as to guarantee a small error in Eq. (1.10).

So the basic idea of a quasi-Monte Carlo method is to replace random samples in a Monte Carlo method by well-chosen deterministic points. The criterion for the choice of deterministic points depends on the numerical problem at hand. For the important problem of numerical integration, the selection criterion is easy to find and leads to the concepts of uniformly distributed sequence and discrepancy.

1.2.1 Discrepancy

The discrepancy can be viewed as a quantitative measure for the deviation from uniform distribution.

Let P be a point set consisting of $\mathbf{x}_1, \cdots, \mathbf{x}_N \in I^s$. For an arbitrary subset B of I_s, we define

$$A(B;P) = \sum_{n=1}^{N} \chi_B(\mathbf{x}_n), \tag{1.11}$$

where χ_B is the characteristic function of B. Thus $A(B;P)$is the counting function that indicates the number of n with $1 \le n \le N$ for which $\mathbf{x}_n \in B$. If $\mathcal{B}$ is a nonempty family of Lebesgue-measurable subsets of I^s, then a general notion of discrepancy of the point set P is given by

$$D_N(\mathcal{B};P) = \sup_{B\in\mathcal{B}} \left|\frac{A(B;P)}{N} - \lambda_s(B)\right|. \tag{1.12}$$

Note that $0 \le D_N(\mathcal{B};P) \le 1$ always and λ_s denotes the s-dimensional Lebesgue measure. By suitable specializations of the family $\mathcal{B}$, we obtain the three most important concepts of discrepancy. We put $\bar{I}^s = [0,1)^s$.

Definition 1.2.1 (star discrepancy) *The star discrepancy* $D_N^*(P) = D_N^*(\mathbf{x}_1, \cdots, \mathbf{x}_N)$ *of the point set* P *is defined by* $D_N^*(P) = D_N(\mathcal{J}^*; P)$*, where* $\mathcal{J}^*$ *is the family of all subintervals of* $\bar{I}^s$ *of the form* $\prod_{i=1}^s [0, u_i)$*.*

Definition 1.2.2 (extreme discrepancy) *The (extreme) discrepancy* $D_N(P) = D_N(\mathbf{x}_1, \cdots, \mathbf{x}_N)$ *of the point set* P *is defined by* $D_N(P) = D_N(\mathcal{J}; P)$*, where* $\mathcal{J}$ *is the family of all subintervals of* $\bar{I}^s$ *of the form* $\prod_{i=1}^s [u_i, v_i)$*.*

Definition 1.2.3 (isotropic discrepancy) *The isotropic discrepancy* $J_N(P) = J_N(\mathbf{x}_1, \cdots, \mathbf{x}_N)$ *of the point set* P *is defined by* $J_N(P) = D_N(\mathcal{C}; P)$*, where* $\mathcal{C}$ *is the family of all convex subets of* I^s*.*

The properties of the discrepancy are discussed in [6]. The following error bounds analysis is also from this book.

1.2.2 Koksma-Hlawka inequality

We discuss the most important error bounds for the quasi-Monte Carlo approximation Eq. (1.10). We start with the one-dimensional case, a classical result is the following inequality of Koksma.

Theorem 1.2.1 *If* f *has bounded variation* $V(f)$ *on* $[0, 1]$*, then, for any* $x_1, \cdots, x_N \in [0, 1]$*,we have*

$$|\frac{1}{N}\sum_{k=1}^{N} f(x_k) - \int_0^1 f(x)\mathrm{d}x| \leq V(f) D_N^*(x_1, \cdots, x_N). \tag{1.13}$$

Theorem 1.2.2 *If* f *is continuous on* $[0, 1]$*, then, for any* $x_1, \cdots, x_N \in [0, 1]$*,we have*

$$|\frac{1}{N}\sum_{k=1}^{N} f(x_k) - \int_0^1 f(x)\mathrm{d}x| \leq \omega(f, D_N^*(x_1, \cdots, x_N)). \tag{1.14}$$

where $\omega(f, \delta) = \sup_{\substack{\|t\| < \delta \\ x, x+t \in D}} |f(x+t) - f(x)|$ *is modulus of continuity.*

To extend Koksma's inequality to the multidimensional case, we have the following inequality of Hlawka, which is often called the Koksma-Hlawka inequality.

Theorem 1.2.3 *If f has bounded variation $V(f)$ on I^s in the sense of Hardy and Krause, then, for any $\mathbf{x}_1, \cdots, \mathbf{x}_N \in I^s$,we have*

$$|\frac{1}{N}\sum_{k=1}^{N} f(\mathbf{x}_k) - \int_{I^s} f(\mathbf{x})\mathrm{d}\mathbf{x}| \leq V(f)D_N^*(\mathbf{x}_1, \cdots, \mathbf{x}_N)). \tag{1.15}$$

All the proof of the above mentioned theorems can be found in [6].

It is widely believed that the star discrepancy of any N-element point set P is of order $O(N^{-1}(\log N)^{s-1})$, so the error bound of quasi-Monte Carlo integration is of order $O(N^{-1})$.

1.2.3 Advantages of the quasi-Monte Carlo method

The very nature of the quasi-Monte Carlo method, with its completely deterministic procedures, implies that we get deterministic and thus guaranteed error bounds. In principle, it is therefore always possible to determine in advance an integration rule that yields a prescribed level of accuracy. Moreover, with the same computational effort, i.e., with the same number of function evaluations (which are the costly operations in numerical integration), the quasi-Monte Carlo method achieves a significantly higher accuracy than the Monte Carlo method. Thus, on two crucial accounts — determinism and precision — the quasi-Monte Carlo method is superior to the Monte Carlo method.

1.3 Monte Carlo optimization

Another basic problem of numerical analysis to which quasi-Monte Carlo methods can be applied is global optimization[7]. Quasi-Monte Carlo random search was introduced by Niederreiter [8]. Let f be a bounded real-valued function defined on the bounded subset E of R^s, $s \geq 1$, and let $\mathbf{x}_1, \cdots, \mathbf{x}_N$ be points in E. Then

$$m_N = \max_{1 \leq n \leq N} f(\mathbf{x_n}) \tag{1.16}$$

is taken as an approximation for the correct value M of the supremum of f over E. Define

$$d_N = d_N(E) = \sup_{\mathbf{x} \in E} \min_{1 \leq n \leq N} d(\mathbf{x}, \mathbf{x}_n) \tag{1.17}$$

as the dispersion of $\mathbf{x}_1, \cdots, \mathbf{x}_N$ in E, where $d(\mathbf{y}, \mathbf{z}) = \max_{1 \leq j \leq s} |y_j - z_j|$ for $\mathbf{y} = (y_1, \cdots, y_s)$, $\mathbf{z} = (z_1, \cdots, z_s) \in R^s$. Niederreiter [8] proved that

$$M - m_N \leq \omega(d_N) \tag{1.18}$$

where $\omega(t) = \sup_{\substack{\mathbf{x},\mathbf{y}\in E \\ d(\mathbf{x},\mathbf{y})\leq t}} |f(\mathbf{x}) - f(\mathbf{y})|$, $t \geq 0$ is modulus of continuity.

If f is continuous on E, the method described above is convergent. In order to speed up the search and insure the search is global, we propose an adaptive random search in quasi-Monte Carlo methods for global optimization in Chapter 5. We will discuss how to balance the local and global search demands.

1.4 Monte Carlo simulation: Metropolis algorithm

Monte Carlo simulation are used very robust in physics related fields such as Monte Carlo method in liquid simulation [9], particle simulation of heat equation [10], Boltzmann equation solution [11], etc.

Metropolis algorithm[12] is the origin of Monte Carlo method, it's very robust in the physical simulation. So we introduce this algorithm here.

In the simulation of liquid, suppose the system is at state m, the whole energy is $\mathcal{V}_m$. The system try to transform to state n. First calculate the total energy $\mathcal{V}_n$ of state n, then compare the value of these two energy. If $\delta\mathcal{V}_{nm} = \mathcal{V}_n - \mathcal{V}_m \leq 0$, which means that the energy of state n is less than that of state m. Because the system tends to stay at the low energy state, the probability of staying at state n is greater than the probability of staring at state m, then the system transform to state n without doubt. If $\delta\mathcal{V}_{nm} = \mathcal{V}_n - \mathcal{V}_m > 0$, which means that the energy of state n is greater than that of state m. The system transform to state n with probability ρ_n/ρ_m. As for the canonical, or constant-NVT ensemble,

$$\frac{\rho_n}{\rho_m} = \exp(-\beta\delta\mathcal{V}_{nm})$$

where β is Boltzman factor. So generate a random number ξ, if $\xi < \exp(-\beta\delta\mathcal{V}_{nm})$, then the state transform to state n; else, the system stays at the current state. We can conclude that the probability of transforming from state m to state n is $\min(1, \exp(-\beta\delta\mathcal{V}_{nm}))$. From the initial state, the system repeat the following two step:

- Step 1: Try to walk from the current state to the next state, calculate the energy of next state.
- Step 2: Calculate the probability of transforming $\min(1, \exp(-\beta\delta\mathcal{V}_{nm}))$ and decide the real state of next step, transform to next state or stay at the current state.

This is a Markov Chain process, the system will come to steady state.

In fact the key of Metropolis algorithm is to calculate the probability of system transformming from current state to the next state. We will describe the Monte Carlo methods in light transport in Chapter 7. We will focus on the probability calculating.

1.5 Further reading

There are many books on Monte Carlo and quasi-Monte Carlo methods. Some books [13][14][15] are collections of conference which cover all the three applications, some[16] focus on the random number generation. Others[17] discuss the Monte Carlo methods in biology fields.

Chapter 2

Random number generator

Random sampling is at the heart of the Monte Carlo method. The success of a Monte Carlo calculation depends, of course, on the appropriateness of the underlying stochastic model, but also, to a large extent, on how well the random numbers used in the computation simulate the random variables in the model. This chapter focuses on the generator of random number.

It is common to make a distinction between uniform and nonuniform random numbers. Uniform random numbers are random numbers for which the target distribution is the uniform distribution U on I. Nonuniform random numbers are usually generated by starting from uniform random numbers and transforming them to fit a given target distribution $F \neq U$. We will introduce the generator of pseudorandom numbers and quasirandom numbers, then describe the transforming method.

2.1 Pseudorandom number

Early in the history of the Monte Carlo method, it already became clear that "truly random" numbers are fictitious from a practical point of view. Therefore users have resorted to pseudorandom numbers(PRN) that can be readily generated in the computer by deterministic algorithms with relatively few input parameters.

A classical and still very popular method for the generation of uniform PRN is the linear congruential method introduced by Lehmer[18]. As the parameters in this method, we choose a large positive integer M, an integer a with $1 \leq a < M$ and $\gcd(a, M) = 1$, and an integer $c \in Z_M = (0, 1, \cdots, M-1)$. Then we select an initial value $y_0 \in Z_M$ and generate a sequence $y_0, y_1, \cdots \in Z_M$ by the recursion

$$y_{n+1} \equiv a y_n + c \bmod M \qquad n = 0, 1, \cdots \tag{2.1}$$

From the sequence, we derive the linear congruential pseudornadom numbers

$$x_n = \frac{y_n}{M} \in \bar{I} = [0, 1) \qquad n = 0, 1, \cdots \tag{2.2}$$

In this context, M is referred to as the modulus and a as the multiplier. The choice of the modulus is customarily accorded with the word length of the machine, typical values being $M = 2^{32}$ or the Mersenne prime $M = 2^{32} - 1$. For high-precision calculations , larger values such as $M = 2^{48}$ have also been used.

2.2 Quasirandom number

For pseudorandom numbers generated by any of methods, the limit is taken over a cyclic finite set, and the supremum in the discrepancy is a maximum. It is useful to consider a deterministic sequence with low discrepancy. The objective is that any finite subsequence fill the space uniformly.

Several such sequence have been proposed, such as Van der Corput sequence, Halton sequences [19], Faure sequence [20], Sobol' sequences [21] and Niederreiter's (t, m, s)-nets and (t, s) sequences [22]. These sequences are called quasirandom sequences. Whereas pseudorandom sequences or pseudorandom generators attempt to simulate randomness, quasirandom sequence is for it to "look like" a sequence of realization of uniform random variables; but for a (finite) quasirandom sequence the objective is that it fill a unit hypercube as uniformly as possible. For general review of quasirandom sequences, see [4] and [23].

2.2.1 Halton sequence

Halton sequences are generalizations of Van der Corput sequences. A Halton sequence is formed by reversing the digits in the representation of some sequence of integers in a given base. Although this can be done somewhat arbitrarily, a straightforward way of forming a s-dimension Halton sequence $\mathbf{x}_1, \mathbf{x}_2, \cdots$, where $\mathbf{x}_i = (x_{i1}, x_{i2}, \cdots, x_{is})$ is first to choose s bases, $b_1, b_2, \cdots, b_s$, perhaps the first s primes. The j^{th} base will be used to form the j^{th} component of each vector in the sequence. Then begin with some integer m and

1. Choosing t_{mj} suitably large, represent m in each base:

$$m = \sum_{k=0}^{t_{mj}} a_{mk} b_j^k, \qquad j = 1, \cdots, s$$

2. Form
$$x_{ij} = \sum_{k=0}^{t_{mj}} a_{mk} b_j^{k-t_{mj}-1}, \qquad j = 1, \cdots, s$$

3. Set $m = m + 1$ and repeat

Suppose, for example, $s = 3$, $m = 15$, and we use the bases 2, 3, and 5. We form $15 = 1111_2$, $15 = 120_3$, and $15 = 30_5$, and deliver the first x as $(0.1111_2, 0.021_3, 0.03_5)$ or $(0.937500, 0.259259, 0.120000)$.

2.2.2 Sobol' sequence

A Sobol' sequence is based on a set of "direction numbers", v_i. The v_i are
$$v_i = \frac{m_i}{2^i},$$
where the m_i are odd positive integers less than 2^i; and the v_i are chosen so that they satisfy a recurrence relation using the coefficients of a primitive polynomial whose coefficients are either 0 or 1,
$$f(z) = z^p + c_1 z^{p-1} + \cdots + c_{p-1} z + c_p \tag{2.3}$$
For $i > p$, the recurrence relation is
$$v_i = c_1 v_{i-1} \oplus c_2 v_{i-2} \oplus \cdots \oplus c_p v_{i-p} \oplus \lfloor v_{i-p}/2^p \rfloor \tag{2.4}$$
where $\oplus$ denotes bitwise binary exclusive-or. An equivalent recurrence for the m_i is
$$m_i = 2c_1 m_{i-1} \oplus 2^2 c_2 m_{i-2} \oplus \cdots \oplus 2^p c_p m_{i-p} \oplus m_{i-p} \tag{2.5}$$
As an example, consider the primitive polynomial
$$x^4 + x + 1$$
The corresponding recurrence is
$$m_i = 8m_{i-3} \oplus 16m_{i-4} \oplus m_{i-4}$$
If we start with $m_1 = 1$, $m_2 = 1$, $m_3 = 3$, and $m_4 = 13$, for example, we get
$$\begin{aligned} m_5 &= 8 \oplus 16 \oplus 1 \\ &= 01000_2 \oplus 10000_2 \oplus 00001_2 \\ &= 11001_2 \\ &= 25 \end{aligned}$$

The i^{th} number in the Sobol' sequence is now formed as

$$x_i = b_1 v_1 \oplus b_2 v_2 \oplus b_3 v_3 \oplus \cdots ,$$

where $\cdots b_3 b_2 b_1$ is the binary representation of i.

Antonov and Saleev[24] show that equivalently the Sobol' sequence can be formed as

$$x_i = g_1 v_1 \oplus g_2 v_2 \oplus g_3 v_3 \oplus \cdots , \tag{2.6}$$

where $\cdots g_3 g_2 g_1$ is the binary representation of a particular Gray code evaluated at i. (A Gray code is a function, $G(i)$, on the nonnegative integers such that the binary representation of $G(i)$ and $G(i+1)$ differ in exactly one bit. namely in the position of the rightmost zero bit in the binary representation of i) The binary representation of the Gray code used by Antonbov and Saleev is

$$\cdots g_3 g_2 g_1 = \cdots b_3 b_2 b_1 \oplus b_4 b_3 b_2.$$

(This is the most commonly used Gray code, which yields function values $0, 1, 3, 2, 6, 7, 5, 4, \cdots$)The Sobol' sequence from Eq. (2.6) can be generated recursively by

$$x_i = x_{i-1} \oplus v_r$$

where r is determined so that b_r is the rightmost zero bit in the binary representation of $i - 1$.

Bratley and Fox[25] discuss criteria for starting values, $m_1, m_2, \cdots$ (The starting values used in the example with the primitive polynomial above satisfy those criteria.)

Sobol' sequence of two dimension is plotted in Fig. 2.1, it is compared with pseudo random sequence in Fig. 2.2. The $ran2$ is one of pseudorandom generator from [26]. The C codes of Sobol' sequence modified from [26] is presented in appendix.

2.2.3 Niderreiter's (t, m, s)-net and (t, s)-sequences

In this section we define point sets and sequences with a very regular distribution behavior. These will be called (t, m, s) nets and (t, s) sequences, respectively.

For the subsequent definitions, we fix the dimension $s \geq 1$ and an integer $b \geq 2$. A subinterval E of $\bar{I}^s$ of the form

$$E = \prod_{i=1}^{s} [a_i b^{-d_i}, (a_i + 1) b^{-d_i})$$

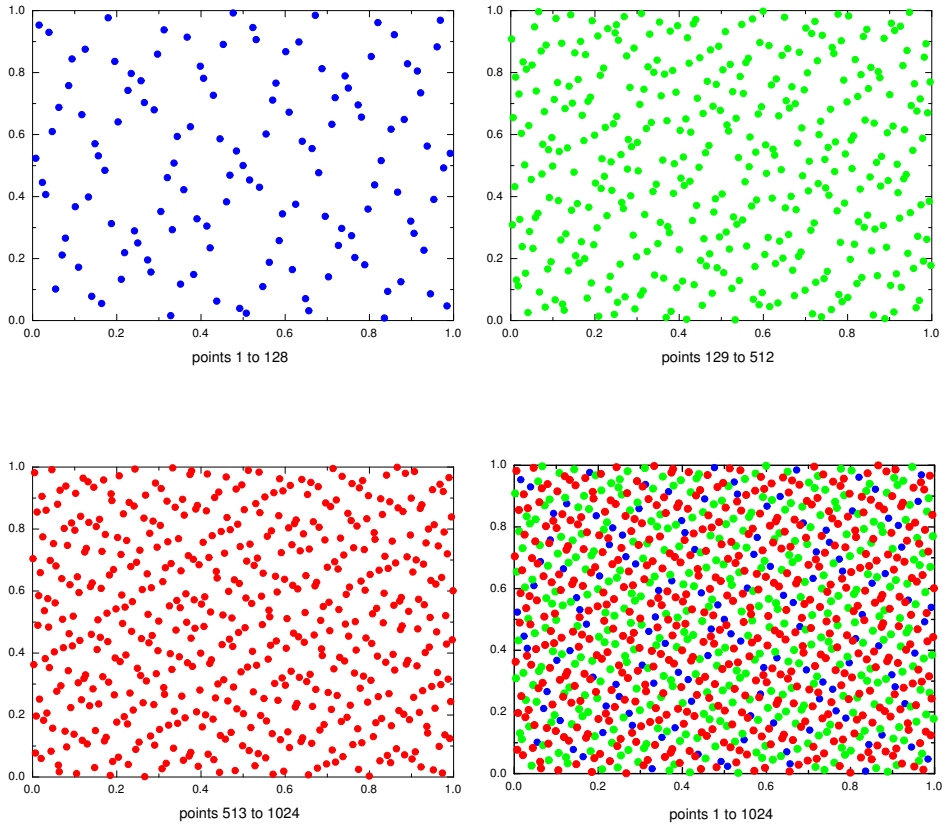

Figure 2.1: Sobol' points of two dimensions. The new generated successive points fill in the gaps in the previously generated points

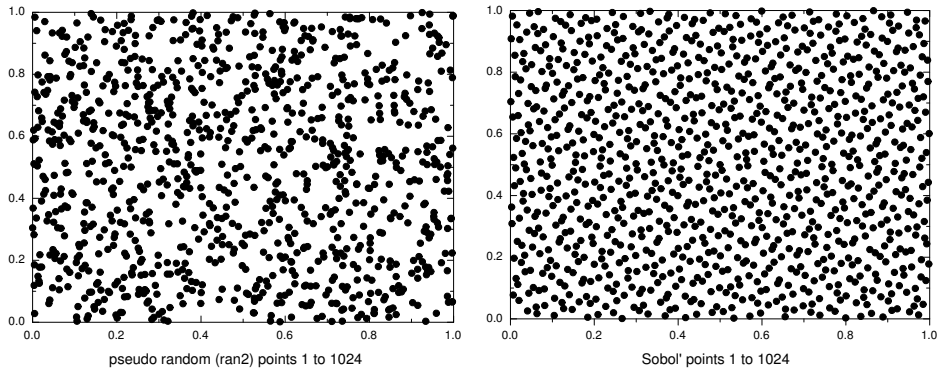

Figure 2.2: Quasirandom sequences have better uniformity properties alternative to pseudorandom sequences

with $a_i, d_i \in \mathbb{Z}, d_i \geq 0, 0 \leq a_i < b^{d_i}$ for $1 \leq i \leq s$ is called an elementary interval in base b.

Definition 2.2.1 $((t, m, s) - net)$ *Let $0 \leq t \leq m$ be integers. A (t, m, s)-net in base b is a point set P of b^m points in $\bar{I}^s$ such that $A(E, P) = b^t$ for every elementary interval E in base b with $\lambda_s(E) = b^{t-m}$.*

Definition 2.2.2 $((t, s) - sequence)$ *Let $t \geq 0$ be an integer. A sequence $\mathbf{x}_0, \mathbf{x}_1, \cdots$ of points in $\bar{I}^s$ is a (t, s)-sequence in base b if, for all integers $k \geq 0$ and $m > t$, the point set consisting of the $\mathbf{x}_n$ with $kb^m \leq n < (k+1)b^m$ is a (t, m, s)-net in base b.*

2.3 Transformations of uniform deviates: general methods

Sampling of random variates from a nonuniform distribution is usually done by transformation to uniform variates. The methods discussed in this section are "universal" in the sense that they apply to almost any distribution.

2.3.1 Inverse CDF method

If x is a scalar random variable with a continuous cumulative distribution function(CDF) P_x, then the random variable

$$\xi = P_x(x) = \int_a^x p(t)\mathrm{d}t \tag{2.7}$$

has a $U(0, 1)$ distribution. This fact provides a very simple relationship with a uniform random variable U and a random variable x with distribution function

$$x = P_x^{-1}(\xi).$$

Use of this straightforward transformation is called the inverse CDF technique. Whenever the inverse of the distribution function is easy to compute, the inverse CDF method is a good one. However, because it is relatively difficult to compute the inverse of some distribution functions of interest, the inverse CDF method is not as commonly used as its simplicity might suggest.

2.3.2 Rejection methods

Let $p(\mathbf{x})$ be a probability density function defined on I^s. The algorithm of standard rejection can be described as follows:

1. Select $\gamma \geq \sup_{\mathbf{x} \in I^s} \; p(\mathbf{x})$.
2. Repeat until N points have been accepted:
   (a) Sample $(\mathbf{x}_t, y_t)$ from $U([0,1]^{s+1})$.
   (b) If $y_t < \gamma^{-1} p(\mathbf{x}_t)$, accept trial point $\mathbf{x}_t$.
   otherwise, reject them.

Rejection methods, like any method for generating nonuniform random numbers, are dependent on a good source of uniforms.

Chapter 3

B-spline smoothed rejection sampling and its applications

The rejection sampling method is one of the most popular methods used in Monte Carlo methods. It turns out that the standard rejection method is closely related to the problem of Monte Carlo integration of characteristic functions, whose accuracy may be lost due to the discontinuity of the characteristic functions. We proposed a B-spline smoothed rejection sampling method[27], which smoothed the characteristic function by B-spline smoothing technique without changing the integral quantity. Numerical experiments showed that for quasi-Monte Carlo method, the convergence rate of nearly $O(N^{-1})$ is regained by using the B-spline smoothed rejection method in importance sampling. As for the Monte Carlo method, the error size using B-spline smoothed importance sampling is better than the standard error size $O(N^{-\frac{1}{2}})$. These numerical results prove that the B-spline smoothed rejection sampling is superior to the standard rejection sampling method.

3.1 Introduction

Standard rejection sampling method is of importance in practical quasi-Monte Carlo methods such as decision process and sampling from a density function. However it is not as efficient as theoretically expected.

Rejection sampling method can be interpreted as the integration of a characteristic function. Recall the quasi-Monte Carlo integration introduced in Chapter 1, the error can be described by Koksma-Hlawka inequality and in general the error order of $O(N^{-1})$ is expected. However, characteristic functions have infinite variation, the exception being rectangles with sides parallel to the coordinate axes. So the Koksma-Hlawka inequality cannot be used to derive an upper bound and theoretical error bounds of size $O(N^{-1})$

are often not observed.

We smoothed the characteristic function by B-spline technique, and regained the error bounds of size $O(N^{-1})$. B-spline smoothed rejection sampling method is introduced in section 3.2.3. We also developed the importance sampling by using B-spline smoothed rejection sampling method in section 3.3.2. Numerical experiments will be given in section 3.4.

3.2 Rejection method

3.2.1 Standard rejection method, interpreted as integration of characteristic function

We discuss the standard rejection method described in section 2.3.2. By Bayes' formula, the density function of accepted points $p_{accept}(\mathbf{x})$ can be interpreted as a Monte Carlo evaluation of the following integral:

$$p_{\,\text{accept}}(\mathbf{x}) = \frac{\int_0^1 \chi(y < \gamma^{-1}p(\mathbf{x}))\mathrm{d}y}{\int_{I^s}[\int_0^1 \chi(y < \gamma^{-1}p(\mathbf{x}))\mathrm{d}y]\mathrm{d}\mathbf{x}} = \frac{p(\mathbf{x})/\gamma}{1/\gamma} = p(\mathbf{x}).$$

where $\chi(y < \gamma^{-1}p(\mathbf{x}))$ is characteristic function defined as:

$$\chi(y < \gamma^{-1}p(\mathbf{x})) = \begin{cases} 1, & if\ y < \gamma^{-1}p(\mathbf{x}), \\ 0, & otherwise. \end{cases} \tag{3.1}$$

So this algorithm produces an infinite sequence P of accepted points in s dimensions distributed according to $p(\mathbf{x})$.

But how well is the quality of the first N elements of the sequence P? We introduce the more general concept of discrepancy[28].

Definition 3.2.1 (F-discrepancy) *Assume that $P_N = \{\mathbf{x}_i, , i = 1, \ldots, N\}$ is a set of points in I^s, and $F_N(\mathbf{x})$ is the empirical distribution of P_N, i.e.*

$$F_N(\mathbf{x}) = (1/N)\sum_{i=1}^{N} \chi\{\mathbf{x}_i \leq \mathbf{x}\}$$

The F-discrepancy of P_N with respect to cumulative distribution function $F(\mathbf{x})$ is defined by

$$D_F(P_N) = \sup_{\mathbf{x}\in I^s} |F_N(\mathbf{x}) - F(\mathbf{x})|. \tag{3.2}$$

The F-discrepancy is a measure for the quality of the representation of $F(\mathbf{x})$ by a point set P_N. As shown in the literature[29], The F-discrepancy is in fact the error of quasi-Monte Carlo integration of a characteristic function and the known theoretical bounds may only be $O(N^{-1/(s+1)})$, which is due to the discontinuity of the characteristic function. A smoothed rejection method was given[29] [30], which replaced the characteristic functions by continuous but non-differentiable functions. In section 3.2.3, we propose a B-spline smoothed rejection sampling method which introduced a differentiable weight function.

3.2.2 Smoothing characteristic function

We can smooth the discontinuous function by B-spline technique[31]. It's well known that for any integrable function $f(x)$, $x \in R$, we call

$$f_h(x) = \frac{1}{h}\int_{x-\frac{h}{2}}^{x+\frac{h}{2}} f(t)\mathrm{d}t \tag{3.3}$$

its average function. Denote $D^{-1}f(x) = \int_a^x f(t)\mathrm{d}t$, we have

$$f_h(x) = h^{-1}\delta_h D^{-1} f(x)$$

where $\delta_h F(x) = F(x+\frac{h}{2}) - F(x-\frac{h}{2})$. We apply the smoothing operator $h^{-1}\delta_h D^{-1}$ (h is smoothing width) to some simple and basic discontinuous functions. For example, if $f(x) = x_+$, then

$$\begin{aligned} f_h(x) &= \tfrac{1}{2h}[(x+\tfrac{h}{2})_+^2 - (x-\tfrac{h}{2})_+^2] \\ &= \begin{cases} 0, & if\ x \le -\frac{h}{2}, \\ (x+\frac{h}{2})^2/2h, & if\ -\frac{h}{2} < x \le \frac{h}{2}, \\ x, & if\ x > \frac{h}{2}. \end{cases} \end{aligned}$$

It is obvious that average function $f_h(x)$ is continuous function. When h is sufficiently small, $f_h(x)$ is the approximation function of $f(x)$. The difference between them is the function value over $[x-\frac{h}{2}, x+\frac{h}{2}]$.

Theorem 3.2.1 *If $f_h(x)$ is the average function of $f(x)$ defined by Eq.(3.3), then*

$$f_h(x) \approx f(x).$$

and $\lim_{h\to 0} f_h(x) = f(x)$.

We call the above described smoothing technique B-spline smoothing technique.

3.2.3 B-spline smoothed rejection sampling method, without changing the integral quantity

The characteristic function $\chi(y < \gamma^{-1}p(\mathbf{x}))$ defined by Eq. (3.1) can also be smoothed by B-spline smoothing technique. We rewrite $\chi(y < \gamma^{-1}p(\mathbf{x}))$ as

$$W_0(\mathbf{x}, y) = (\gamma^{-1}p(\mathbf{x}) - y)_+^0, \qquad 0 \le y \le 1.$$

We apply the smoothing operator $(2h)^{-1}\delta_{2h}D^{-1}$ to $W_0(\mathbf{x}, y)$, where smoothing width is $2h$.

$$\begin{aligned} W_\delta(\mathbf{x}, y) &= (2h)^{-1}\delta_{2h}D^{-1}W_0(\mathbf{x}, y) \\ &= (2h)^{-1}[(\gamma^{-1}p(\mathbf{x}) - y + h)_+ - ((\gamma^{-1}p(\mathbf{x}) - y - h)_+] \end{aligned}$$

Denote $f_1(y) = (\gamma^{-1}p(\mathbf{x}) - y + h)_+$ and $f_2(y) = (\gamma^{-1}p(\mathbf{x}) - y - h)_+$, applying the smoothing operator $h^{-1}\delta_h D^{-1}$ to $f_1(y)$ and $f_2(y)$ again, we get the differientable weight function.

$$W_{\delta\delta}(\mathbf{x}, y) = (2h)^{-1}[h^{-1}\delta_h D^{-1}f_1(y) - h^{-1}\delta_h D^{-1}f_2(y)]$$

$$= \begin{cases} 1 & , \ 0 \le y < \gamma^{-1}p(\mathbf{x}) - \frac{3h}{2} \\ \frac{[(\gamma^{-1}p(\mathbf{x}) - y + h) - \frac{(\gamma^{-1}p(\mathbf{x}) - y - \frac{h}{2})^2}{2h}]}{2h} & , \ \gamma^{-1}p(\mathbf{x}) - \frac{3h}{2} \le y < \gamma^{-1}p(\mathbf{x}) - \frac{h}{2}, \\ \frac{(\gamma^{-1}p(\mathbf{x}) - y + h)}{2h} & , \ \gamma^{-1}p(\mathbf{x}) - \frac{h}{2} \le y < \gamma^{-1}p(\mathbf{x}) + \frac{h}{2}, \\ \frac{(\gamma^{-1}p(\mathbf{x}) - y + \frac{3h}{2})^2}{4h^2} & , \ \gamma^{-1}p(\mathbf{x}) + \frac{h}{2} \le y < \gamma^{-1}p(\mathbf{x}) + \frac{3h}{2}, \\ 0 & , \ \gamma^{-1}p(\mathbf{x}) + \frac{3h}{2} \le y \le 1. \end{cases} \tag{3.4}$$

The function $W_{\delta\delta}(\mathbf{x}, y)$ is our weight function used to replace characteristic function $\chi(y < \gamma^{-1}p(\mathbf{x}))$. The modified B-spline smoothed rejection sampling method is described as follows:

1. Select $\gamma \ge \sup_{\mathbf{x}\in I^s} \ p(\mathbf{x})$.

2. Repeat until the sum of weight w_t is within one unit of N:

   (a) Sample $(\mathbf{x}_t, y_t)$ from $U([0, 1]^{s+1})$.

   (b) Set weight $w_t = W_{\delta\delta}(\mathbf{x}_t, y_t)$ to points $\mathbf{x}_t$, namely the accept probability of point $\mathbf{x}_t$ is w_t.

Theorem 3.2.2 *$W_{\delta\delta}(\mathbf{x}, y)$ is weight function defined by Eq. (3.4), $\chi(y < \gamma^{-1}p(\mathbf{x}))$ is characteristic function defined by Eq. (3.1). We have*

$$\int_0^1 W_{\delta\delta}(\mathbf{x}, y)\mathrm{d}y = \int_0^1 \chi(y < \gamma^{-1}p(\mathbf{x}))\mathrm{d}y$$

The sequence generated by B-spline smoothed rejection sampling method is distributed according to $p(\mathbf{x})$.

Proof: Set $t = \gamma^{-1}p(\mathbf{x}) - \frac{3h}{2}$, then

$$\begin{aligned}\int_0^1 W_{\delta\delta}(\mathbf{x}, y)\mathrm{d}y &= \int_0^t 1\mathrm{d}y + \int_t^{t+h} \frac{[(t-y+\frac{5h}{2}) - \frac{(t-y+h)^2}{2h}]}{2h}\mathrm{d}y + \int_{t+h}^{t+2h} \frac{(t-y+\frac{5h}{2})}{2h}\mathrm{d}y \\ &\quad + \int_{t+2h}^{t+3h} \frac{(t-y+3h)^2}{4h^2}\mathrm{d}y \\ &= t + \frac{11h}{12} + \frac{h}{2} + \frac{h}{12} = t + \frac{3h}{2} = \gamma^{-1}p(\mathbf{x})\end{aligned}$$

So $\int_0^1 W_{\delta\delta}(\mathbf{x}, y)\mathrm{d}y = \int_0^1 \chi(y < \gamma^{-1}p(\mathbf{x}))\mathrm{d}y$.

That is to say that the sequence generated by B-spline smoothed rejection sampling method is distributed according to $p(\mathbf{x})$.

We will apply the B-spline smoothed rejection sampling method to importance sampling in quasi-Monte Carlo integration in section 3.3.2.

3.3 Importance sampling

3.3.1 Standard importance sampling

Importance sampling is probably the most widely used variance reduction technique among Monte Carlo methods. Rewrite the integral $I(f)$ as

$$I(f) = \int_{I^s} f(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{I^s} \frac{f(\mathbf{x})}{p(\mathbf{x})}p(\mathbf{x})\mathrm{d}\mathbf{x},$$

where $p(\mathbf{x})$ is an importance function, which is chosen such that it mimics the behavior of $f(\mathbf{x})$ over I^s. The standard importance-sampled estimate is

$$I_N^{(\mathrm{I}S)} = \frac{1}{N}\sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)}, \tag{3.5}$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_N$ are samples from the density $p(\mathbf{x})$. Rejection sampling method is used robustly to sample points from $p(\mathbf{x})$. However, the improved performance for quasi-Monte Carlo method is not often observed, this degradation is due to the discontinuity of characteristic functions. We introduced the B-spline smoothed rejection sampling method into the importance sampling, and regained integration error of size $O(N^{-1})$.

3.3.2 Improving importance sampling with B-spline smoothed rejection sampling method

We rewrite the integral $I(f)$ as

$$\begin{aligned} I(f) = \textstyle\int_{I^s} f(\mathbf{x})\mathrm{d}\mathbf{x} &= \gamma \textstyle\int_{I^s} \frac{f(\mathbf{x})}{p(\mathbf{x})}\gamma^{-1}p(\mathbf{x})\mathrm{d}\mathbf{x} \\ &= \gamma \textstyle\int_{I^s} \frac{f(\mathbf{x})}{p(\mathbf{x})}[\int_0^1 \chi(y < \gamma^{-1}p(\mathbf{x})\mathrm{d}y]\mathrm{d}\mathbf{x} \\ &= \gamma \textstyle\int_{I^s} \frac{f(\mathbf{x})}{p(\mathbf{x})}[\int_0^1 W_{\delta\delta}(\mathbf{x}, y)\mathrm{d}y]\mathrm{d}\mathbf{x} \\ &\approx \frac{\gamma}{N^*}\sum_{i=1}^{N^*} W_{\delta\delta}(\mathbf{x}_i, y_i)\frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)} \end{aligned}$$

The improved importance-sampled estimate of quasi-Monte Carlo integration is defined as

$$I_N^{(\mathrm{B}IS)} = \frac{1}{N}\sum_{i=1}^{N^*} W_{\delta\delta}(\mathbf{x}_i, y_i)\frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)}, \tag{3.6}$$

where $W_{\delta\delta}(\mathbf{x}, y)$ is defined by Eq. (3.4) and N^* is chosen such that the sum of acceptance weights w_i is within one unit of N. It is obvious that

$$N \approx N^*/\gamma$$

Numerical experiments for improved importance sampling in Monte Carlo and quasi-Monte Carlo integration will be given in the following section.

3.4 Numerical experiments

In this section, the standard estimates, the standard rejection methods and B-spline smoothed rejection sampling methods for Monte Carlo and quasi-Monte Carlo integration will be compared on several classical functions. The following estimates will be computed by Eq. (1.2), (3.5) and (3.6):
Crude Monte Carlo: $Y_N^{(1)} = (1/N)\sum_{i=1}^N f(\mathbf{x}_i)$, $\mathbf{x}_i \sim U([0,1]^s)$;

Standard rejection method: $Y_N^{(2)} = (1/N)\sum_{i=1}^N f(\mathbf{x}_i)/p(\mathbf{x}_i)$, $\mathbf{x}_i \sim p(\mathbf{x}_i)$;

B-spline smooth rej.: $Y_N^{(3)} = (1/N)\sum_{i=1}^{N^*} W_{\delta\delta}(\mathbf{x}_i, y_i)f(\mathbf{x}_i)/p(\mathbf{x}_i)$.

For a given N, take m samples of these estimates, denoted by $Y_N^{(j)}(k)$ for $1 \le k \le m$(using successive points from a single sequence). The final approximation of integral $I(f)$ is given by $\hat{I}^{(j)} = (1/m)\sum_{k=1}^m Y_N^{(j)}(k)$. In all

cases the errors can be estimated by the empirical standard deviation(sd) and the empirical root mean square error($rmse$), defined respectively as:

$$sd(\hat{\sigma}^{(j)}) = \sqrt{\frac{1}{(m-1)}\sum_{k=1}^{m}[Y_N^{(j)}(k) - \hat{I}^{(j)}]^2}, j = 1, 2, 3. \tag{3.7}$$

$$rmse(\hat{\sigma}^{(j)}) = \sqrt{\frac{1}{m}\sum_{k=1}^{m}[Y_N^{(j)}(k) - I]^2}, j = 1, 2, 3. \tag{3.8}$$

We use Halton sequences[32] for quasirandom points and generate pseudorandom points using function *ran2* [26]. Set $m = 75$, that is to say that 75 runs for each estimate. Log-log plots are used so that slopes(which are presented parenthetically in the figure keys)correspond to convergence rates.

Example 1. This example is Monte Carlo and quasi-Monte Carlo integration over $I^7 = [0, 1]^7$ of the function

$$f_1(\mathbf{x}) = e^{1-(\sin^2(\frac{\Pi}{2}x_1)+\sin^2(\frac{\Pi}{2}x_2)+\sin^2(\frac{\Pi}{2}x_3))}\arcsin(\sin(1) + \frac{x_1 + \cdots + x_7}{200})$$

using the positive definite importance function:

$$p_1(\mathbf{x}) = \frac{1}{\eta}e^{1-(\sin^2(\frac{\Pi}{2}x_1)+\sin^2(\frac{\Pi}{2}x_2)+\sin^2(\frac{\Pi}{2}x_3))},$$

where $\eta = \int_{I^7} e^{1-(\sin^2(\frac{\Pi}{2}x_1)+\sin^2(\frac{\Pi}{2}x_2)+\sin^2(\frac{\Pi}{2}x_3))}\mathrm{d}\mathbf{x} = e \cdot (\int_0^1 e^{-(\sin^2(\frac{\Pi}{2}x)}\mathrm{d}x)^3$, which is easily approximated to high accuracy as a one-dimensional integral.

The resulting sd error for example 1 using pseudorandom and quasirandom points are presented in Fig. 3.1 and 3.2.

The computational examples show that:

- Quasi-Monte Carlo methods give much smaller errors than Monte Carlo methods with the same sample size.
- Both for quasi-Monte Carlo and Monte Carlo, the importance sampling with B-spline smoothed rejection sampling is better than that with standard rejection sampling.
- For Monte Carlo, the importance sampling with B-spline smoothed rejection sampling also improves error.

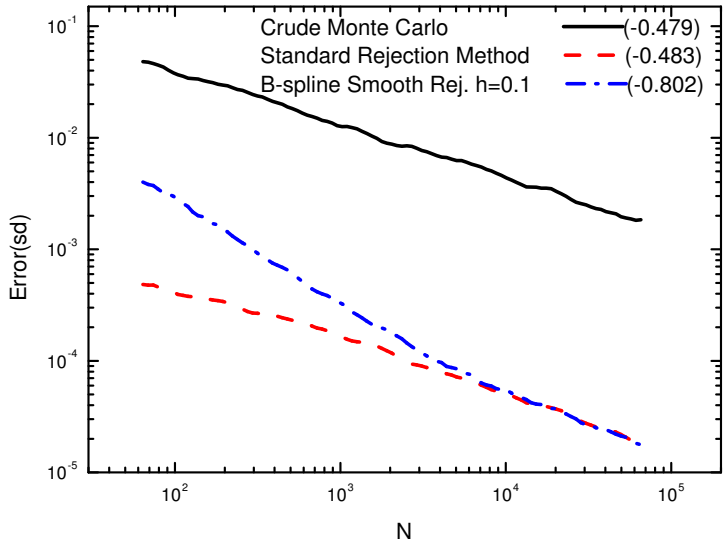

Figure 3.1: The resulting *sd* error for example 1 using pseudorandom points.

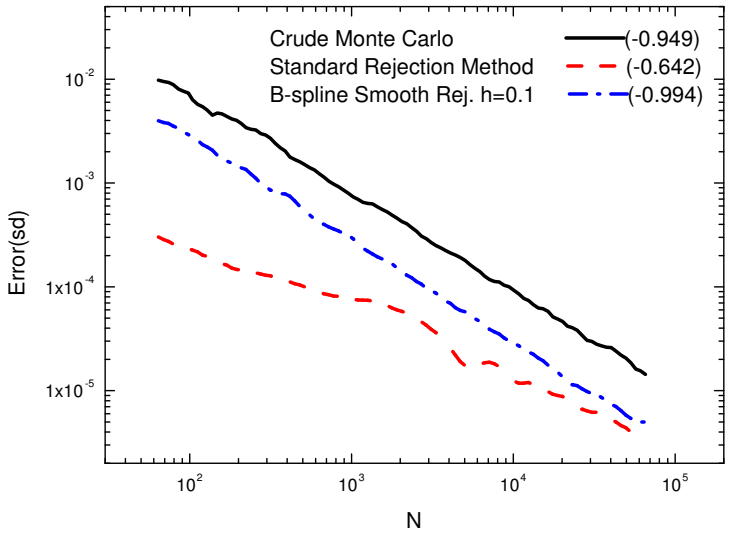

Figure 3.2: The resulting *sd* error for example 1 using pseudorandom points.

3.5 Conclusions

We can conclude that B-spline smoothed rejection sampling methods can improve the rejection method, and make the importance sampling more efficient in quasi-Monte Carlo methods. It can also be seen that use of modified differentiable weight functions in B-spline smoothed rejection sampling methods may improve Monte Carlo methods. As shown in the numerical example, the error size of $O(N^{-0.8})$ is much better than $O(N^{-0.5})$ of crude Monte Carlo. Though we can use quasi-Monte Carlo method to calculate the integral for the numerical example here, not necessary to use importance sampling and rejection sampling. It is necessary to use rejection sampling in decision process and the use of B-spline smoothed rejection sampling method will be much more efficient than standard rejection sampling.

Chapter 4

Fine antithetic variable method for Monte Carlo integration

We get a theoretical error of fine antithetic variables Monte Carlo(FAMC) method for multidimensional integration. The error size is $O(N^{-(\frac{1}{2}+\frac{2}{s})})$ for functions having second continuous derivative, where s is the dimension of the integrand. We also give the theoretical error result of antithetic variable Monte Carlo(AMC) method for multi-variable functions whose degree is no more than two. The constant before $O(N^{-\frac{1}{2}})$ is less than that of the MC method. We realize the parallel programming in C language. The results of the numerical experiments coincide with the theoretical results very well.

4.1 Introduction

Consider the problem of numerical estimation of an absolutely convergent multidimensional integral

$$I = \int_{I^s} f(\mathbf{x})\mathrm{d}\mathbf{x} \tag{4.1}$$

where $I^s = [0,1]^s$ is the s-dimensional unit cube. And a crude Monte Carlo(MC) estimator

$$M = \frac{1}{N}\sum_{k=1}^{N} f(\xi_k) \tag{4.2}$$

can be used for evaluation I. Here $\xi_1, \xi_2, \cdots, \xi_N$ are independent values of random number ξ uniformly distributed on I^s. The absolute value of the error of MC method of size $O(N^{-\frac{1}{2}})$ has been proved in [4][6]. Another estimator described here is antithetic variables Monte Carlo(AMC) given in

[33].

$$A = \frac{1}{2N}\sum_{k=1}^{N}[f(\xi_k) + f(2\mathbf{c} - \xi_k)] \tag{4.3}$$

where $\mathbf{c} = (\frac{1}{2}, \cdots, \frac{1}{2})^{\mathrm{T}}$ is the center of I^s. The name of antithetic variables was introduced by Hammersley and Morton [34]. Now we divide I^s into $N = n^s$ sub-cube D_k by uniformly grids, $\mathbf{c}_k$ is the center of D_k. Define $\mathbf{d}_k$ as the nearest point of D_k to $\mathbf{O} = (0, \cdots, 0)^{\mathrm{T}}$, set $\eta_k = \mathbf{d}_k + \xi_k/N^{1/s}$. And the fine antithetic variables Monte Carlo(FAMC) estimator we proposed is defined as:

$$F = \frac{1}{2N}\sum_{k=1}^{N}[f(\eta_k) + f(2\mathbf{c}_k - \eta_k)] \tag{4.4}$$

S.Haber proposed the same method in [35] and gave the theorem proving that error of FAMC method is of size $O(N^{-(\frac{1}{2}+\frac{2}{s})})$ for functions having second continuous derivative. We present the theorem in section 4.2 and give our proof. The corollary shows that the error of AMC method is the same size as that of the MC method while the constant before $O(N^{-\frac{1}{2}})$ of AMC is less than that of MC. Because number of calculated function values for antithetic variables Monte Carlo is related to the dimension s, the traditional serial program is time consuming. We realize the parallel programming in C language in section 4.3. The numerical results given in section 4.4 coincide with the theoretical results very well.

4.2 Theorems on fine antithetic variables Monte Carlo

Let $\|\mathbf{x}\| = (\sum_{i=1}^{s}(x_i)^2)^{\frac{1}{2}}$ for $\mathbf{x} = (x_1, \cdots, x_s)^{\mathrm{T}} \in R^s$. Suppose D is the convex domain in R^s and $\delta > 0$, define modulus of continuity $\omega(f, \delta)$ and second order modulus of continuity $\Omega(f, \delta)$ for $f : D \to R$ respectively as:

$$\omega(f, \delta) = \sup_{\substack{\|\mathbf{t}\|<\delta \\ \mathbf{x},\mathbf{x}+\mathbf{t}\in D}} |f(\mathbf{x}+\mathbf{t}) - f(\mathbf{x})| \tag{4.5}$$

$$\Omega(f, \delta) = \sup_{\substack{\|\mathbf{t}\|<\delta \\ \mathbf{x}\pm\mathbf{t}\in D}} |f(\mathbf{x}+\mathbf{t}) - 2f(\mathbf{x}) + f(\mathbf{x}-\mathbf{t})| \tag{4.6}$$

For $f : D \to R^s$, the modulus of continuity is defined as

$$\omega(f, \delta) = (\sum_{i=1}^{s}\omega(f_i, \delta)^2)^{\frac{1}{2}} \tag{4.7}$$

F is the FAMC estimator and A the AMC estimator of integral I described in the introduction. We have the following theoretical results.

Lemma 4.2.1 *Denote $C^1(D)$ the class of functions having first continuous derivative on convex domain D. Suppose $f \in C^1(D)$, then the second order modulus of continuity $\Omega(f,\delta)$ and the modulus of continuity $\omega(f,\delta)$ have the following inequality.*

$$\Omega(f,\delta) \le 2\delta\omega(f',\delta) \tag{4.8}$$

where $\delta > 0$.

Proof. For all $\mathbf{x} \pm \mathbf{t} \in D$, $\exists\ \theta_i \in (0,1), i = 1,2$, such that

$$\begin{aligned} f(\mathbf{x}+\mathbf{t}) - f(\mathbf{x}) &= \sum_{i=1}^{s} \frac{\partial f(\mathbf{x}+\theta_1\mathbf{t})}{\partial x_i} t_i, \\ f(\mathbf{x}-\mathbf{t}) - f(\mathbf{x}) &= -\sum_{i=1}^{s} \frac{\partial f(\mathbf{x}-\theta_2\mathbf{t})}{\partial x_i} t_i. \\ \text{So} \quad f(\mathbf{x}+\mathbf{t}) - 2f(\mathbf{x}) + f(\mathbf{x}-\mathbf{t}) &= \sum_{i=1}^{s} \left(\frac{\partial f(\mathbf{x}+\theta_1\mathbf{t})}{\partial x_i} - \frac{\partial f(\mathbf{x}-\theta_2\mathbf{t})}{\partial x_i}\right) t_i. \end{aligned}$$

For all $\|\mathbf{t}\| < \delta$, an application of Cauchy inequality gives

$$\begin{aligned} \Omega(f,\delta)^2 &\le \sum_{i=1}^{s} \left(\left|\frac{\partial f(\mathbf{x}+\theta_1\mathbf{t})}{\partial x_i} - \frac{\partial f(\mathbf{x})}{\partial x_i}\right| + \left|\frac{\partial f(\mathbf{x})}{\partial x_i} - \frac{\partial f(\mathbf{x}-\theta_2\mathbf{t})}{\partial x_i}\right| \right)^2 \sum_{i=1}^{s} (t_i)^2 \\ &\le \sum_{i=1}^{s} [2\omega(\frac{\partial f}{\partial x_i}, \delta)]^2 \cdot \delta^2 \end{aligned}$$

Therefore $\Omega(f,\delta) \le 2\delta\omega(f',\delta)$.

Theorem 4.2.2 *Denote $C(I^s)$ the space of continuous functions on $I^s = [0,1]^s$. Consider $f \in C(I^s)$, the variance of* FAMC *estimator F can be estimated by the second order modulus of continuity $\Omega(f,\delta)$ as*

$$\mathrm{E}(F-I)^2 < \frac{1}{4N}\Omega(f, \frac{1}{2N^{1/s}})^2 \tag{4.9}$$

where N is the number of random points.

Proof. Denote

$$\begin{aligned} F_k &= \frac{1}{2N}[f(\eta_k) + f(2\mathbf{c}_k - \eta_k)], \\ I_k &= \int_{D_k} f(\mathbf{x})\mathrm{d}\mathbf{x}. \end{aligned}$$

Introducing the change of variable $\xi_k = N^{1/s}(\eta_k - \mathbf{d}_k)$, we have

$$\begin{aligned} \mathrm{E}F_k &= \frac{1}{2N}\int_{I^s}[f(\eta_k) + f(2\mathbf{c}_k - \eta_k)]\mathrm{d}\xi_k \\ &= \frac{1}{2}\int_{D_k}[f(\eta_k) + f(2\mathbf{c}_k - \eta_k)]\mathrm{d}\eta_k \\ &= I_k. \end{aligned}$$

Substituting F_k and I_k into $\mathrm{E}(F_k - I_k)^2$, then adding and subtracting the term $2f(\mathbf{c}_k)$, we obtain the variance of F_k.

$$\begin{aligned} \mathrm{E}(F_k - I_k)^2 = &\quad N\int_{D_k}(F_k - I_k)^2\mathrm{d}\eta_k \\ = &\quad N\int_{D_k}\Big\{\Big[\frac{1}{2N}(f(\eta_k) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \eta_k))\Big] \\ &-\Big[\frac{1}{2}\int_{D_k}(f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x}))\mathrm{d}\mathbf{x}\Big]\Big\}^2\mathrm{d}\eta_k \end{aligned}$$

Expanding the square term, then regrouping the equation, we have

$$\begin{aligned} \mathrm{E}(F_k - I_k)^2 = &\quad \frac{1}{4N}\int_{D_k}[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]^2\mathrm{d}\mathbf{x} \\ &-\frac{1}{4}\Big\{\int_{D_k}[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]\mathrm{d}\mathbf{x}\Big\}^2. \\ < &\quad \frac{1}{4N}\int_{D_k}[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]^2\mathrm{d}\mathbf{x} \end{aligned}$$

Since $\mathbf{c}_k$ is the center of D_k, so $\|\mathbf{x} - \mathbf{c}_k\| < \dfrac{1}{2N^{1/s}}$ for $\mathbf{x} \in D_k$. According to the definition of $\Omega(f, \delta)$, we have $f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x}) \le \Omega(f, \frac{1}{2N^{1/s}})$. Therefore

$$\begin{aligned} \mathrm{E}(F_k - I_k)^2 \le &\quad \frac{1}{4N}\int_{D_k}\Omega(f, \frac{1}{2N^{1/s}})^2\mathrm{d}\mathbf{x} \\ = &\quad \frac{1}{4N^2}\Omega(f, \frac{1}{2N^{1/s}})^2 \end{aligned}$$

Since the random points are independent, so $\mathrm{Cov}(F_i - I_i)(F_k - I_k) = 0, i \neq k$. Summing of $\mathrm{E}(F_k - I_k)^2$ sub k, we have

$$\mathrm{E}(F - I)^2 = \sum_{k=1}^{N}\mathrm{E}(F_k - I_k)^2 < \frac{1}{4N}\Omega(f, \frac{1}{2N^{1/s}})^2$$

The proof is terminated.

Applying Lemma 4.2.1 to Theorem 4.2.2, we get

Corollary 4.2.3 *Denote $C^1(I^s)$ the class of functions having first continuous derivative on $I^s = [0,1]^s$. If $f \in C^1(I^s)$, then the variance of F can be estimated by the modulus of continuity $\omega(f,\delta)$ as*

$$\mathrm{E}(F-I)^2 < \frac{1}{4N^{1+2/s}}\omega(f', \frac{1}{2N^{1/s}})^2 \tag{4.10}$$

where N is the number of random points.

Following in a similar manner, when $f \in C^2(I^s)$, we can estimate $\mathrm{E}(F-I)^2$ of size $O(\frac{1}{N^{1+\frac{4}{s}}})$. Furthermore we have the following exact result.

Theorem 4.2.4 *Denote $C^2(I^s)$ the class of functions having second continuous derivative on $I^s = [0,1]^s$. Assume $f \in C^2(I^s)$, then the variance of F has the following exact estimation.*

$$\mathrm{E}(F-I)^2 = \frac{1}{288}\int_{I^s}\sum_{i=1}^{s}\sum_{j=1}^{s}(1-\frac{3}{5}\delta_{ij})\Big(\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\Big)^2 \mathrm{d}\mathbf{x}\cdot\frac{1}{N^{1+\frac{4}{s}}} + o(\frac{1}{N^{1+\frac{4}{s}}}), \tag{4.11}$$

where $\delta_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$ is the symbol of Kronecleer and N is the number of random points.

Proof. From the proof of Theorem 4.2.2, we have

$$\begin{aligned} \mathrm{E}(F_k - I_k)^2 = \; & \frac{1}{4N}\int_{D_k}[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]^2 \mathrm{d}\mathbf{x} \\ & -\frac{1}{4}\Big\{\int_{D_k}[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]\mathrm{d}\mathbf{x}\Big\}^2. \end{aligned} \tag{4.12}$$

Now

$$\begin{aligned} & f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x}) \\ = \; & (\mathbf{x}-\mathbf{c}_k)^{\mathrm{T}} f''(\mathbf{c}_k)(\mathbf{x}-\mathbf{c}_k) + o(\frac{1}{N^{\frac{2}{s}}}) \\ = \; & \sum_{i=1}^{s}\sum_{j=1}^{s}\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i \partial x_j}(x_i - c_{ki})(x_j - c_{kj}) + o(\frac{1}{N^{\frac{2}{s}}}) \end{aligned}$$

Since $\mathbf{c}_k$ is the center of D_k, so $\int_{D_k}(x_i - c_{ki})(x_j - c_{kj})\mathrm{d}\mathbf{x} = 0$, we get

$$\begin{aligned} & \int_{D_k}[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]\mathrm{d}\mathbf{x} \\ = & \sum_{i=1}^{s}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\int_{D_k}(x_i - c_{ki})^2\mathrm{d}x + o(\frac{1}{N^{1+\frac{2}{s}}}) \qquad (4.13) \\ = & \frac{1}{12N^{1+\frac{2}{s}}}\sum_{i=1}^{s}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2} + o(\frac{1}{N^{1+\frac{2}{s}}}) \end{aligned}$$

And the integration of $[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]^2$ on D_k is

$$\begin{aligned} & \int_{D_k}[f(\mathbf{x}) - 2f(\mathbf{c}_k) + f(2\mathbf{c}_k - \mathbf{x})]^2\mathrm{d}\mathbf{x} \\ = & \int_{D_k}\sum_{i=1}^{s}\sum_{j=1}^{s}\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i \partial x_j}(x_i - c_{ki})(x_j - c_{kj})\sum_{m=1}^{s}\sum_{n=1}^{s}\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_m \partial x_n}(x_m - c_{km})(x_n - c_{kn})\mathrm{d}\mathbf{x} \\ & + o(\frac{1}{N^{1+\frac{4}{s}}}) \end{aligned}$$

All the non-zero terms satisfy $\begin{cases} m \neq i \\ j = i \\ n = m \end{cases}$ or $\begin{cases} j \neq i \\ m = i \\ n = j \end{cases}$ or $\begin{cases} j \neq i \\ n = i \\ m = j \end{cases}$ or $n = m = j = i$.
So

$$
\begin{aligned}
&\int_{D_k}[f(\mathbf{x})-2f(\mathbf{c}_k)+f(2\mathbf{c}_k-\mathbf{x})]^2\mathrm{d}\mathbf{x}\\
=\;&\sum_{i=1}^{s}\sum_{m\neq i}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_m)^2}\int_{D_k}(x_i-c_{ki})^2(x_m-c_{km})^2\mathrm{d}\mathbf{x}\\
&+2\sum_{i=1}^{s}\sum_{j\neq i}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i\partial x_j}\Big)^2\int_{D_k}(x_i-c_{ki})^2(x_j-c_{kj})^2\mathrm{d}\mathbf{x}\\
&+\sum_{i=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\Big)^2\int_{D_k}(x_i-c_{ki})^4\mathrm{d}\mathbf{x}+o(\frac{1}{N^{1+\frac{4}{s}}})\\
=\;&\frac{1}{144N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\sum_{m\neq i}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_m)^2}+\frac{2}{144N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\sum_{j\neq i}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i\partial x_j}\Big)^2\\
&+\frac{1}{80N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\Big)^2+o(\frac{1}{N^{1+\frac{4}{s}}})
\end{aligned}
$$

By adding and subtracting the term $\Big(\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\Big)^2$ under the summation sub i, we get

$$
\begin{aligned}
&\int_{D_k}[f(\mathbf{x})-2f(\mathbf{c}_k)+f(2\mathbf{c}_k-\mathbf{x})]^2\mathrm{d}\mathbf{x}\\
=\;&\frac{1}{144N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\sum_{m=1}^{s}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_m)^2}+\frac{2}{144N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\sum_{j=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i\partial x_j}\Big)^2\\
&+(\frac{1}{80}-\frac{3}{144})\frac{1}{N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\Big)^2+o(\frac{1}{N^{1+\frac{4}{s}}})\\
=\;&\frac{1}{144N^{1+\frac{4}{s}}}\Big(\sum_{i=1}^{s}\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\Big)^2+\frac{1}{72N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\sum_{j=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i\partial x_j}\Big)^2\\
&-\frac{1}{120N^{1+\frac{4}{s}}}\sum_{i=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\Big)^2+o(\frac{1}{N^{1+\frac{4}{s}}})
\end{aligned}
\tag{4.14}
$$

Now substitution Eq. (4.14) and Eq. (4.13) into Eq. (4.12), we obtain

$$\mathrm{E}(F_k - I_k)^2 = \frac{1}{288N^{2+\frac{4}{s}}}\sum_{i=1}^{s}\sum_{j=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i \partial x_j}\Big)^2 - \frac{1}{480N^{2+\frac{4}{s}}}\sum_{i=1}^{s}\Big(\frac{\partial^2 f(\mathbf{c}_k)}{(\partial x_i)^2}\Big)^2 + o(\frac{1}{N^{2+\frac{4}{s}}})$$

Therefore

$$\begin{aligned}\mathrm{E}(F-I)^2 &= \sum_{k=1}^{N}\mathrm{E}(F_k - I_k)^2 \\ &= \frac{1}{288N^{2+\frac{4}{s}}}\sum_{k=1}^{N}\sum_{i=1}^{s}\sum_{j=1}^{s}(1-\frac{3}{5}\delta_{ij})\Big(\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i \partial x_j}\Big)^2 + o(\frac{1}{N^{1+\frac{4}{s}}}) \\ &= \frac{1}{288N^{1+\frac{4}{s}}}\lim_{N\to+\infty}\frac{1}{N}\sum_{k=1}^{N}\Big[\sum_{i=1}^{s}\sum_{j=1}^{s}(1-\frac{3}{5}\delta_{ij})\Big(\frac{\partial^2 f(\mathbf{c}_k)}{\partial x_i \partial x_j}\Big)^2\Big] + o(\frac{1}{N^{1+\frac{4}{s}}})\end{aligned}$$

Since f is function having second continuous derivative, so $\sum_{i=1}^{s}\sum_{j=1}^{s}(1-\frac{3}{5}\delta_{ij})\Big(\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\Big)^2$ is continuous function. Replacing summation sub k by integration, we have

$$\mathrm{E}(F-I)^2 = \frac{1}{288N^{1+\frac{4}{s}}}\int_{I^s}\sum_{i=1}^{s}\sum_{j=1}^{s}(1-\frac{3}{5}\delta_{ij})\Big(\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\Big)^2 \mathrm{d}\mathbf{x} + o(\frac{1}{N^{1+\frac{4}{s}}})$$

The proof is completed.

By suitable modification to the proof of Theorem 4.2.4, we can get

Corollary 4.2.5 *Denote $\sqcap_2(I^s)$ the class of multi-variable functions whose degree is no more than two on $I^s = [0,1]^s$. If $f \in \sqcap_2(I^s)$, then the variance of* AMC *estimator A has the following exact estimation*

$$\mathrm{E}(A-I)^2 = \frac{1}{288N}\sum_{i=1}^{s}\sum_{j=1}^{s}(1-\frac{3}{5}\delta_{ij})\Big(\frac{\partial^2 f(\mathbf{c})}{\partial x_i \partial x_j}\Big)^2 \tag{4.15}$$

where $\mathbf{c} = (\frac{1}{2},\cdots,\frac{1}{2})^{\mathrm{T}}$ and N is the number of random points.

From Theorem 4.2.4, we can see that the root mean square error of FAMC method is of size $O(N^{-(\frac{1}{2}+\frac{2}{s})})$. It's better than the result of MC method.

The Corollary 4.2.5 shows that the constant before $O(N^{-\frac{1}{2}})$ of AMC is less than that of MC and the variance is zero for the linear functions, in particular. We will give numerical experiments in section 4.4.

4.3 Parallel programming for antithetic variable Monte Carlo integration

We realize the MPI parallel programming in C language. The program run on the computer cluster "Large Scale Scientific Computing"(lssc.cc.ac.cn) of state key Laboratory of Scientific and Engineering Computing, Chinese Academy of Sciences.

In serial program, we calculate N function values on one computer, while the parallel program distributes these calculating on *noprocs* processors. In order to assure the independence of the random numbers, we use only one random number generator to generate one same sequence, so the seeds need to be transform among the processors. MPI program starts using the following codes:

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&nid);
MPI_Comm_size(MPI_COMM_WORLD,&noprocs);
```

here *nid* is the serial number for each processor, the value ranges from 0 to $noprocs-1$. Each processor once generates a successive sub_seq_len random numbers. The remain number of N that can not been divided by sub_seq_len is marked as $remainder$.

```
procs_step=sub_seq_len*noprocs;
remainder=N%sub_seq_len;
size=N-remainder;
```

After generates random numbers, each processor transforms the seeds of generator to the next processor, then calculates the function values of these recently generated sub_seq_len random numbers. The next processor(nid_after) of this processor(nid) repeat the same process while the this processor is calculating function values. That is, generates random numbers, transforms seeds of random number generator, calculates function values. So, except the time to generate sub_seq_len random numbers and transform the seeds of random number generator, all processors are calculating the function values simultaneously. Each processor need to know from which processor(nid_before) to receive the seeds of generator before call the generator.

Because the AMC and FAMC methods need to map the random number to every sub-cube, the serial number$(0-(N-1))$ of each random number

is important to be marked. We use *nid* to control this serial number. In the call of subroutine MC(), $k+m$ is the serial number of the random number.

```
for(k=nid*sub_seq_len;k<N;k+=procs_step)
 {...
if(k+sub_seq_len<=size)
   { for(m=0;m<sub_seq_len;m++)
          MC(k+m,n,random[m],interval,sum);
   }
else
   { for(m=0;m<remainder;m++)
          MC(k+m,n,random[m],interval,sum);
   }
...
}
```

It is obvious that the value of k for processor 0 starts from 0, so the function value of the random number whose serial number is 0(among the N random numbers) should be calculated on processor 0. And after finish every N function values calculating, the next N function values calculating should start on the processor 0. It is necessary to trace the processor(*last_nid*) who deal with the last random number(whose serial number is $N-1$) of every N random numbers. When this processor finishes generating *sub_seq_len* or *remainder* random numbers, it should transform the seeds of generator to the processor 0. The processor 0 need to decide whether from processor $noprocs-1$ or from the processor who deals with the $N-1$'th random number to receive the seeds of generator. We use the variable *last_nid* to mark the source processor of processor 0.

```
for(i=0;i<runs;i++) {for(j=1;j<=step;j++)
{ N=points_step[j];
  ...
  last_nid=noprocs-1;
  for(k=nid*sub_seq_len;k<N;k+=procs_step)
  {/*receive the random seeds*/
    if(nid)   /*nid_before=(nid+noprocs-1)%noprocs;*/
        nid_before=nid-1;
    else
        nid_before=last_nid;
    MPI_Irecv(seeds,NTAB+3,MPI_LONG,nid_before,10,MPI_COMM_WORLD,&
          req_recv_seeds);
    MPI_Wait(&req_recv_seeds,&status);
    ...
    nid_after=(nid+1)%noprocs;
    /*in general the (noprocs-1)'th process send the seeds to process 0*/
    last_nid=noprocs-1;
    /*the process who deal with the N'th random number will send the seeds to process 0
      then process 0 start the first random number of new successive N random points
    */
    if((k+sub_seq_len==N)||k+remainder==N)
    {    nid_after=0;
         last_nid=nid;
```

```
      /*boadcast so that process 0 know the change of last_nid*/
      MPI_Bcast(&last_nid,1,MPI_INT,nid,MPI_COMM_WORLD);
  }
  MPI_Isend(seeds,NTAB+3,MPI_LONG,nid_after,10,MPI_COMM_WORLD,&
        req_send_seeds);
  ...
 }
 ...
}/*end of j: step*/
...
}/*end of i :runs*/
```

When all the processors finish calculating N function values, the following code to gather all the sub-sums.

```
MPI_Reduce(sum,G_sum,4,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
```

The parallel program ends with

```
MPI_Finalize();
```

The full codes are in the appendix.

4.4 Numerical experiments

FAMC and AMC method have been compared with MC method for many classical functions. Denote $Y_N^{(1)} = M$, $Y_N^{(2)} = A$, $Y_N^{(3)} = F$. For the *sd* error and *rmse* defined in Eq. (3.7) and Eq. (3.8), we set $m = 75$ and use the pseudo-random generated from *ran2* in [26].

The examples given here are from [36][37].

Example 1.

$$I_1 = \int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{4x_1 x_3^2 \exp(2x_1 x_3)}{(1 + x_2 + x_4)^2} \mathrm{d}x_1 \mathrm{d}x_2 \mathrm{d}x_3 \mathrm{d}x_4 \tag{4.16}$$

Example 2.

$$I_2 = \int_0^1 \cdots \int_0^1 \prod_{i=1}^{s} \frac{1 + 3(x_i)^2}{2} \mathrm{d}x_1 \cdots \mathrm{d}x_s, s = 10 \tag{4.17}$$

Example 3.

$$I_3 = \int_0^1 \cdots \int_0^1 \exp \sum_{i=1}^{s} \frac{x_i}{i} \mathrm{d}x_1 \cdots \mathrm{d}x_s, s = 15 \tag{4.18}$$

The exact value $I_1 = 0.5753$, $I_2 = 1.0$, $I_3 = 5.610253495$

Table 4.1: *rmse* error of example 1, $s = 4$

	16	81	256	625	1296	2401	4096
MC	0.26816	0.12726	0.07522	0.04744	0.03278	0.02726	0.01828
AMC	0.19763	0.08531	0.04605	0.03064	0.02021	0.01497	0.01050
FAMC	0.09145	0.01912	0.00774	0.00302	0.00140	0.00082	0.00043

Table 4.2: *rmse* error of example 2, $s = 10$

	1024	59049	1048576	9765625	60466176	282475249
MC	0.070045	0.010355	0.002235	0.000735	0.000299	0.000127
AMC	0.042890	0.006604	0.001491	0.000522	0.000186	0.000085
FAMC	0.017122	0.001541	0.000209	0.000041	0.000013	0.000004

We denote the error of size $O(N^{\alpha})$. From theoretical results, $\alpha(\text{MC}) = \alpha(\text{AMC}) = -\frac{1}{2}$ and $\alpha(\text{FAMC}) = -(\frac{1}{2} + \frac{2}{s})$. For FAMC method, the results of *rmse* error are presented in tables 4.1–4.3. The first line of the tables is size of N, that is to say that $N = n^s$ random points have been used for estimating the integrations. Theoretical α(FAMC) and the corresponding slopes of linear fit of *rmse* error(log-log data are used) are presented in table 4.4. For AMC and MC method, The results of *sd* error are presented in Fig. 4.1–4.6. Log-log plots are used so that slopes correspond to α and intercept correspond to the constant.

We have the following results:

- From tables 4.1–4.3, we can see that the error of FAMC method is much less than that of the AMC and MC methods.
- Fig. 4.1–4.6 show that the constant before $O(N^{-\frac{1}{2}})$ of AMC is less than that of MC method.

Table 4.3: *rmse* error of example 3, $s = 15$

	32768	14348907	1073741824
MC	0.0093009	0.0004967	0.0000651
AMC	0.0023853	0.0001140	0.0000118
FAMC	0.0006000	0.0000134	0.0000008

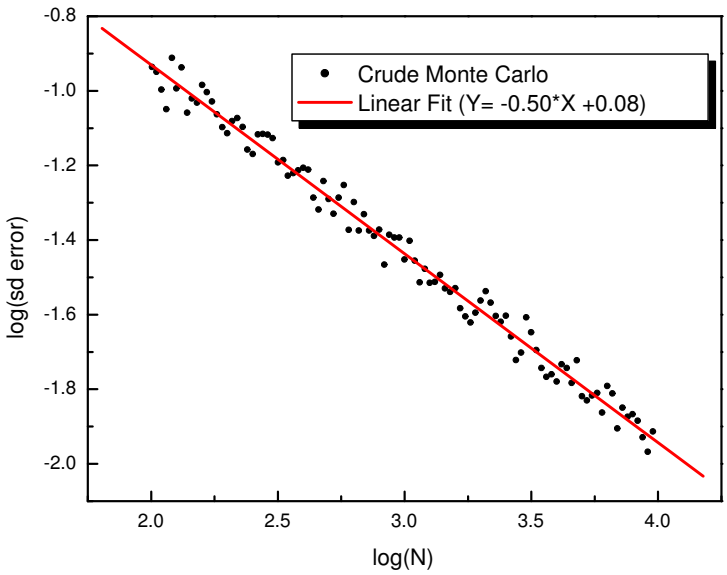

Figure 4.1: The *sd* error of example 1 for MC method.

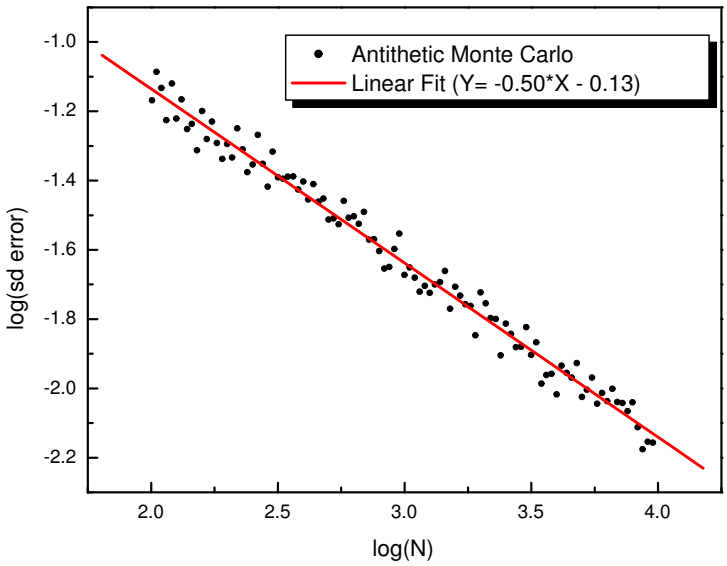

Figure 4.2: The *sd* error of example 1 for AMC method.

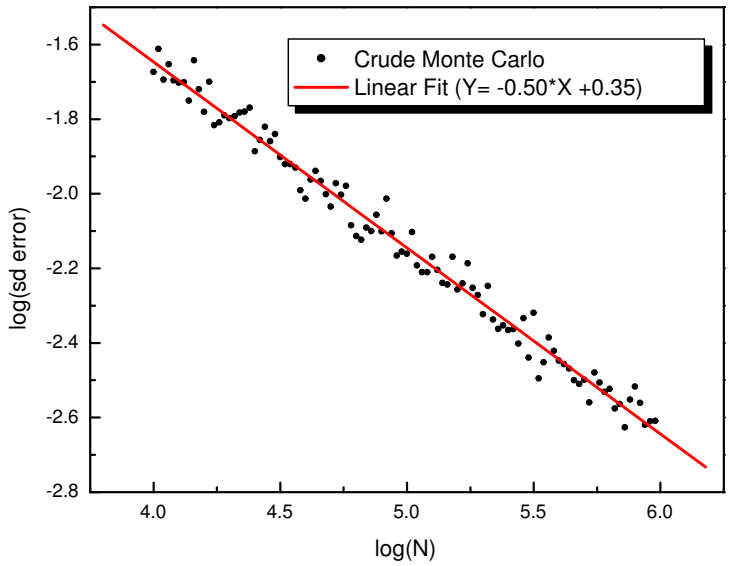

Figure 4.3: The *sd* error of example 2 for MC method.

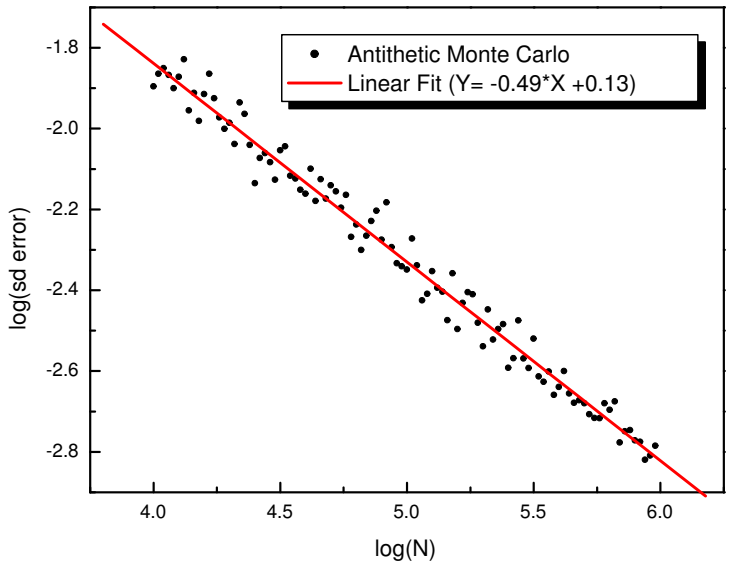

Figure 4.4: The *sd* error of example 2 for AMC method.

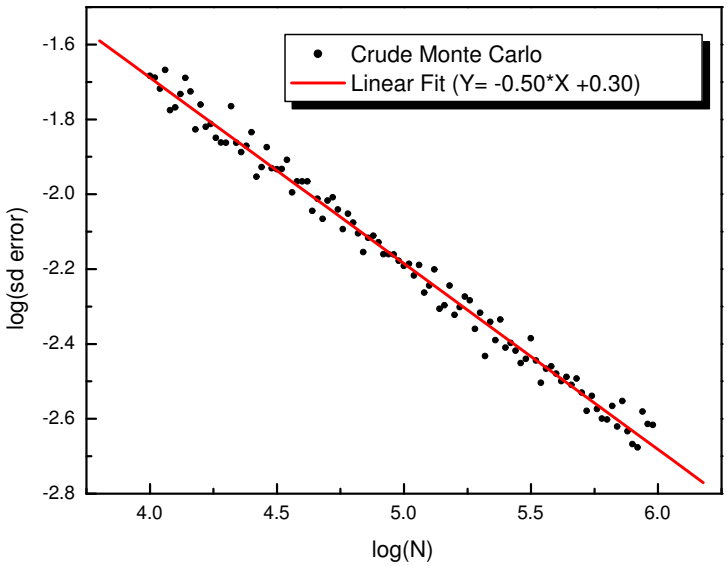

Figure 4.5: The *sd* error of example 3 for MC method.

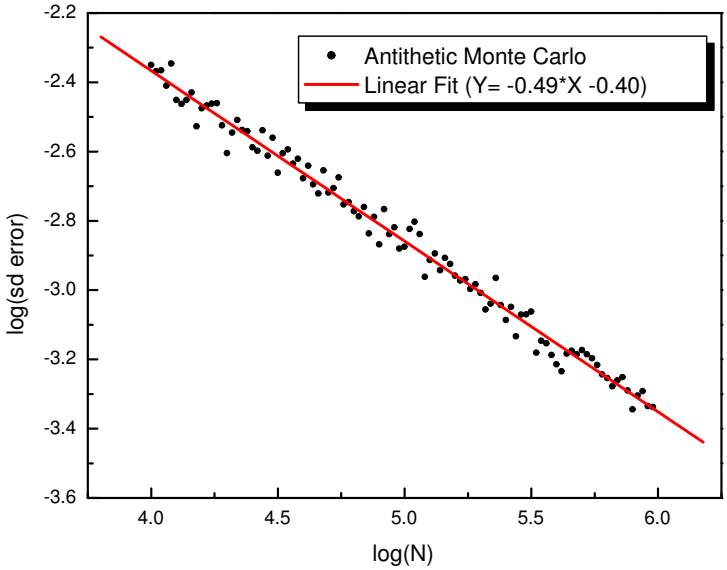

Figure 4.6: The *sd* error of example 3 for AMC method.

Table 4.4: Theoretical α(FAMC) and slope of linear fit of *rmse* error for FAMC method

	s	α(FAMC)	slope
Example 1	4	-1.00	-0.96
Example 2	10	-0.70	-0.67
Example 3	15	-0.63	-0.64

- Slopes of linear fit in table 4.4 fit with α(FAMC) very well. For example, when $s = 10$, $\alpha(\text{FAMC}) = -0.7$, the slope is -0.67.

4.5 Conclusions

Both from the theoretical and numerical experiment results, we can conclude that the FAMC method is superior to the AMC and MC method because of error size of $O(N^{-(\frac{1}{2}+\frac{2}{s})})$ for functions having second continuous derivative. The numerical experiments also coincide with the theoretical result on that the error of AMC method is the same order as that of the MC method, while the constant before $O(N^{-\frac{1}{2}})$ is less than that of MC method.

Chapter 5

Adaptive random search in quasi-Monte Carlo methods for global optimization

Quasi-Monte Carlo random search is useful in nondifferentiable optimization. We introduce an adaptive random search in Quasi-Monte Carlo Methods (AQMC) for global optimization. First, we use adaptive technique in the local search [38] such that it can head for local extremum points quickly because the search direction and search radius are adjusted according to the previous search result. Then, we develop the adaptive technique [39] by borrowing ideas of population evolution from Genetic Algorithms. New individuals will be imported into the population adaptively according to population evolution degree. For quasi-random sequences with low discrepancy, the new generated successive points fill in the gaps in the previously generated distribution, which ensures that the domain of function can be searched evenly and the global extremum can be found. In conclusion, the AQMC method not only speeds up the random search but also balances the global and local demands (adaptive equalization).

5.1 Introduction

The standard quasi-Monte Carlo optimization introduced in Chapter 1 is convergent. However, the rate of convergence is in general very slow. In order to speed up the method, Niederreiter and Peart [40] developed the quasi-random search by using "localization of search"(LQMC). Analogous to the method described in [40], in 1990, Y. Wang and K.T. Fang [41] introduced a sequential number-theoretic method for optimization (SNTO) and its applications in statistics.

The successes of LQMC and SNTO depend to great extent on the condition $d_N < \varepsilon$, where ε is contraction ratio that is a positive number less

than $\frac{1}{2}$. Since $d_N \geq \frac{1}{2}N^{-1/s}$ is an absolute low bound for the dispersion of any N points in $I^s = [0,1]^s$ according to [42], it follows that N must at least be of an order of magnitude ε^{-s}. In addition, if the function has many local maxima, in particular, the local maximum is much close to M, then "localization of search" could be lead into a "wrong track", that is to say, the global maximum could not be found.

In this chapter, we introduce an adaptive technique in local search and in the procedure of population evolution. We call the methods Adaptive Quasi-Monte Carlo methods for global optimum (AQMC). AQMC not only speeds up the random search methods considerably, but also balances the global and local demand (adaptive equalization). The algorithm is described in section 5.3, Numerical experiments will be given in section 5.4.

5.2 LQMC vs. LAQMC

Assume f be defined on rectangular region $E = [\mathbf{a}, \mathbf{b}], \mathbf{a}, \mathbf{b} \in R^s$. The Niederreiter's local search (LQMC) algorithms can be described as following:

- Step 1(initialization): generate N quasi-random points, find $\mathbf{x}_m$ such that $f(\mathbf{x}_m) = \max\limits_{1 \leq n \leq N} f(\mathbf{x}_n)$
- Step 2(mapping): map the N points to a s dimension cube with $\mathbf{x}_m$ as center, ε_i as radius. $g_C(\mathbf{x}) = \mathbf{x}_m + \varepsilon_i(2\mathbf{x} - (\mathbf{a} + \mathbf{b}))$.
- Step 3: find new $\mathbf{x}_m$, such that $f(\mathbf{x}_m) = \max(f(\mathbf{x}_m), \max\limits_{1 \leq n \leq N} f(g_C(\mathbf{x}_n)))$
- Step 4: repeat step 2 and 3 till the search radius is close to zero.

Where N is the order of $O(\varepsilon^{-s})$. ε_i is decreased in each mapping step, it is usually be set as $\varepsilon_i = \varepsilon^i, 0 < \varepsilon < 1/2$(see Fig. 5.3). We call each mapping a generation.

We develop local search in order to find the local extremum more quickly, which means we can use smaller number of points to obtain the extremum . We call it LAQMC method. LAQMC method also have the ability to jump out the local extremum to global extremum sometimes.

LAQMC is different from LQMC in three aspects. Here we mark the generation no. i as subscript. Search direction and search radius ε_{ik} are adjusted according to the previous search result. In addition, N_i is proportion to ε_{ik}.

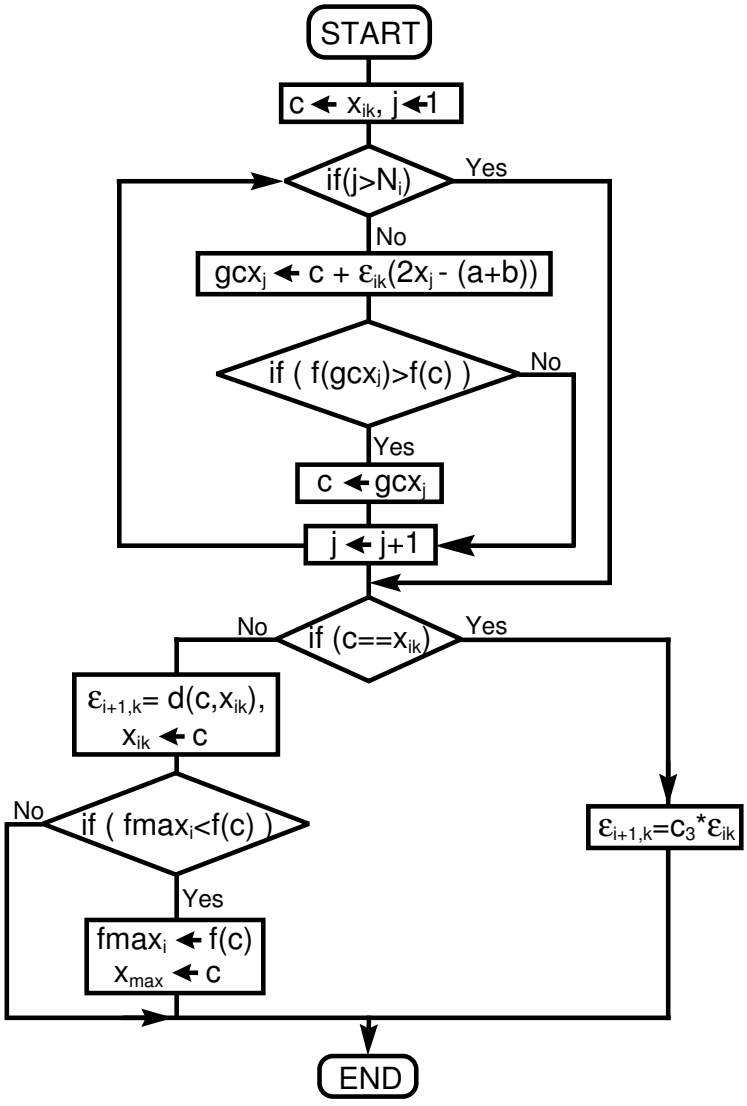

Figure 5.1: Flow chart of local search of AQMC(LAQMC)

LQMC

```
r=0.25;//search radius
a=x[m];//the center

for(i=1;i<=N;i++)
  {
   gcx[i]=a+r*(x[i]-1);
  }


r=r*r;
```

LAQMC

```
r=0.25;//search radius
a=x[m];//the center
Ni=int(c2*N*max(r,c1));
for(i=1;i<=Ni;i++)
  {gcx[i]=a+r*(x[i]-1);
   if(f(gcx[i])-f(a)>1.0E-8)
        a=gcx[i];
  }
tempx=fabs(x[m]-a);
if(tempx>1.0E-8)
    r=tempx;
else
    r=c3*r;//c3<1;
```

Figure 5.2: Codes of LQMC vs LAQMC

For the selected individual $\mathbf{x}_{ik}$, map the first N_i points of the segment $\mathbf{x}_1, \cdots, \mathbf{x}_N$ to the neighborhood of $\mathbf{x}_{ik}$ by $g_C : E \to C$.

$$1 \leq N_i = [c_2 \times N \times \max\{\varepsilon_{ik}, c_1\}] \leq N, \quad 0 < c_1 \leq 1, 0 < c_2 \leq 1 \tag{5.1}$$

$$g_C(\mathbf{x}) = \mathbf{c} + \varepsilon_{ik}(2\mathbf{x} - (\mathbf{a} + \mathbf{b})) \quad for \quad \mathbf{x} \in E \tag{5.2}$$

where $[x]$ denotes the greatest integer $< x$. $\mathbf{c}$ is initially set to be $\mathbf{x}_{ik}$, if $f(g_C(\mathbf{x}_j)) > f(\mathbf{c})$, then $\mathbf{c}$ is set to be $g_C(\mathbf{x}_j)$, $j = 1, \cdots, N_i$. As shown in flow chart (Fig. 5.1), $\varepsilon_{i+1,k}$, the next search radius of k'th individual, will be adjusted according to this search result. If function value bigger than $f(\mathbf{x}_{ik})$ is found, then $\varepsilon_{i+1,k} = d(\mathbf{c}, \mathbf{x}_{ik})$, and $\mathbf{x}_{ik}$ will be replaced by $\mathbf{c}$. Otherwise, we have $\varepsilon_{i+1,k} = c_3 \times \varepsilon_{ik}$, where $0 < c_3 < \varepsilon_0$. We suggest that $c_3 = \varepsilon_0^3$.

We give parts of C codes for LAQMC and LQMC method in Fig. 5.2. To be simplex, the dimension s is set to be 1.

We will test the LAQMC and LQMC method in section 5.4. The adaptive local search of AQMC algorithms is timesaving.

5.3 AQMC algorithms

But local search is not enough for global optimization. Borrowing ideas of population evolution from Genetic Algorithms [43][44][45][46], We introduce an adaptive quasi-Monte Carlo global optimization. We take the initial segment $\mathbf{x}_1, \cdots, \mathbf{x}_N \in E$ of infinite quasi-random sequence(N is relatively

small when s is large) as the initial population and each point is an individual. First calculate fitness for each individual. Then select one individual (selection probability is proportion to fitness) and perform adaptive local search. New individuals will be imported into the population adaptively according to population evolution degree Ed. If better individual is found, Ed will increase, and the probability of importing new individual into the population increases. As we have discussed in section 2.2, the new generated successive points of quasi-random sequence fill in the gaps in the previously generated distribution in E (the domain of function f), which ensures that E can be searched evenly and the global maximum can be found. Let $fmax_i = \max\limits_{\substack{1\le k\le i\\ 1\le j\le N}} f(\mathbf{x}_{kj})$ and $\mathbf{x}_{max}$ denote the corresponding approximate maximum point. The nondecreasing sequence $fmax_1, fmax_2, \cdots$ is taken as an approximation for the correct value M of the supremum of f described above.

Definition 5.3.1 (fitness) *Set $F_{ij} = f(\mathbf{x}_{ij}) - Cmin$, where $Cmin$ is the minimal function value of all individuals till current generation and $f(\mathbf{x}_{ij})$ is the function value of j'th individual of i'th generation. Then*

$$p_{ij} = F_{ij} / \sum_{k=1}^{N} F_{ik}$$

denotes the fitness of the j'th individual of i'th generation. Obviously

$$p_{ij} \ge 0, j = 1, \cdots, N \quad and \quad \sum_{j=1}^{N} p_{ij} = 1.$$

Definition 5.3.2 (evolution degree) *The mean function value of the i'th generation is denoted by $m_i = \sum_{j=1}^{N} f(\mathbf{x}_{ij})/N, i = 1, \cdots$. Initially set $m_0 = m_1$, once the worse individuals of the i'th population be replaced by new points from the sequence, set $m_0 = m_i$. Then*

$$Ed_i = |1 - m_i/m_0| \ (i = 1, \cdots)$$

denotes the population evolution degree of i'th generation.

The concept of evolution degree Ed represents the saturation of local search ability. As for the initial population, $Ed = 0$, the probability to generate new individual is zero, the local search overwhelm generating new individual. As the local search improves the approximation of maximum, the ability of local search to find better maximum points degrades. The evolution degree Ed increases and the probability to generate new individual increases too. When the new random points are generated to be as new

individuals, the population is renewed, the evolution degree Ed is reset to be 0, the evolution restarts. We can see that the AQMC method focus on local search, it only introduces new individuals to avoid stay in the neighbor of local extremum.

Now we describe AQMC algorithms as follows:

- Step 1:
  1. Generate initial segment $\mathbf{x}_1, \cdots, \mathbf{x}_N$ of sequence, set $\mathbf{x}_{ij} = \mathbf{x}_j, \varepsilon_{ij} = \varepsilon_0 (0 < \varepsilon_0 < \frac{1}{2})$ for $i = 1, j = 1, \cdots, N$ as the initial population and calculate p_{ij}.
  2. Let $fmax_i = f(\mathbf{x}_{ik}) = \max\limits_{1 \le j \le N} f(\mathbf{x}_{ij})$ and $\mathbf{x}_{max} = \mathbf{x}_{ik}$.
  3. Calculate m_1 and put $m_0 = m_1$.
- Step 2:

  If (the stop criteria satisfied):

  Program end

  Else:
  (a) $i = i + 1$
  (b) Select one individual according to p_{ij} and perform adaptive local search, $fmax_i$ may be changed after local search(LAQMC).
- Step 3: Calculate p_{ij}, m_i, Ed_i.
- Step 4:
  1. Generate a random number $newp$.
  2. If $(newp < Ed_i)$:
     (a) Generate $c_4 \times N (0 < c_4 \le 1)$ new successive points from the sequence and replace the worse individuals of the i'th population (elitist model) by new points. ε_{ij} of new imported individual is reset to be ε_0.
     (b) Let $tempx$ denote the maximum of the function values of the new imported individuals. If $(fmax_i < tempx)$, then $fmax_i = tempx$.
     (c) Calculate p_{ij}, m_i , set $m_0 = m_i$.
  3. Goto Step 2.

The stop criteria may be set according to various situations. For example, if $fmax_i$ has not been improved after several generations, then we stop running the program. We can also set the total generation number in advance. Moreover, there are many applications that are to find optimal parameters, that is to say, the global extremum is known, and we can control the error between $fmax_i$ and the global extremum.

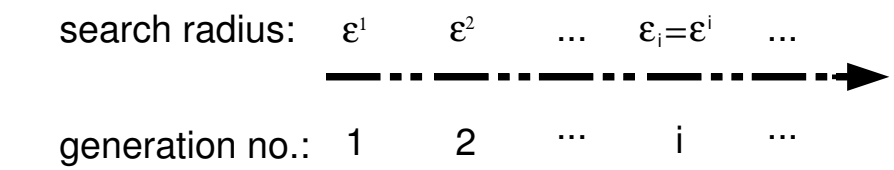

Figure 5.3: The search radius of each generation for local search of Niederreiter' method(LQMC)

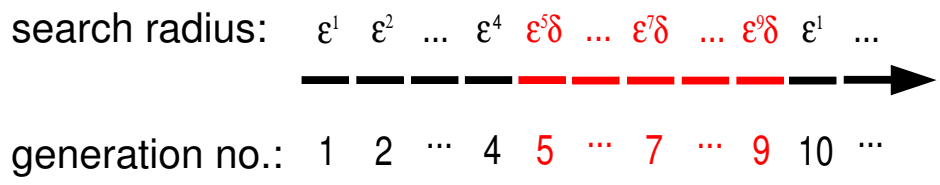

Figure 5.4: The search radius of outer iteration of local search of Niederreiter' method (see Eq. (5.4)), Compare with Fig. 5.3

5.4 Numerical experiments

We have carried out numerical experiments on some classical functions with AQMC. Sobol' sequences [26][25] were used in the experiments. Some examples are given as follows, in which f_1 is taken from [44]. We compare the search result of AQMC with that of LQMC and SNTO, the improvement obtained by AQMC is considerable.

Example 1:

$$f_1 = 100(x_1^2 - x_2)^2 + (1 - x_1)^2, \quad -2.408 \leq x_i \leq 2.408 \tag{5.3}$$

is a function hard to be minimized. The global minimum point is at $(1.0, 1.0)$ and the global minimum is 0. We take $N = 64, \varepsilon_0 = 0.25, \delta = 4.0$ and we have carried out 81920 runs. For the LQMC method, the errors between the global minimum and its approximation are all greater than size $O(10^{-4})$.

For procedure of outer iteration of LQMC [40], if set(see Fig. 5.4)

$$\left.\begin{array}{ll} \varepsilon_i = \varepsilon_{i-1} \times \delta & \text{if}(!i\%5) \\ \varepsilon_i = \varepsilon_0 & \text{if}(!i\%10) \end{array}\right\} \tag{5.4}$$

Then the minimum was found at the 45^{th} generation(see Fig. 5.5 LQMC). Meanwhile, by using the adaptive local search of AQMC(LAQMC), $c_1 = 1.0$,

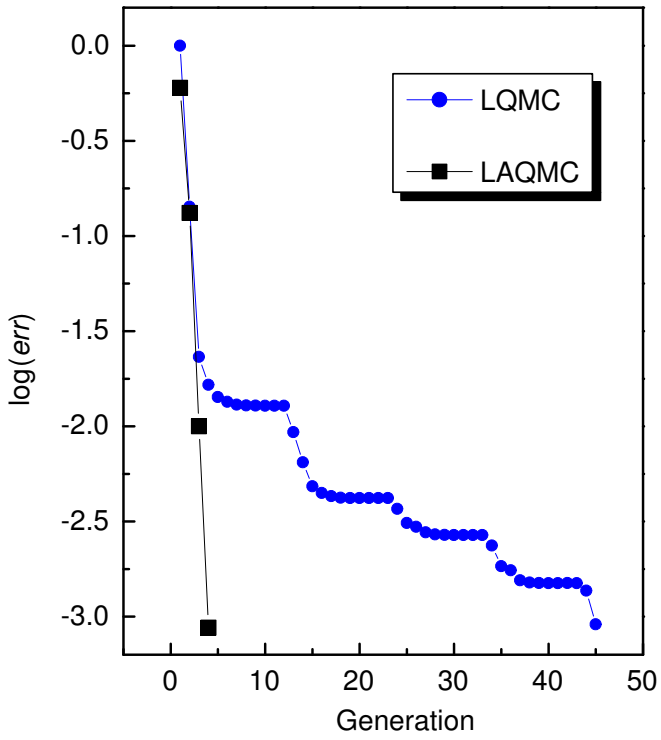

Figure 5.5: Error of LQMC and of LAQMC for function $f_1(s = 2)$

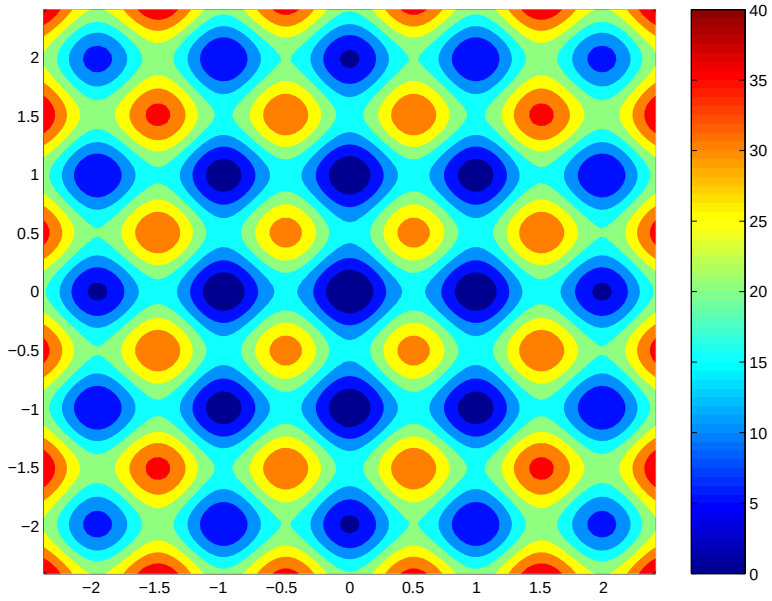

Figure 5.6: Contour of the function of example 2

$c_2 = 1.0$, $c_3 = \varepsilon_0^3$, the minimum was found at the 4^{th} generation(see Fig. 5.5 LAQMC), and 41885 runs out of 81920 runs (51.13%) find the global minimum. Define err as the error between the exactly minimum and the approximation for the minimum, Fig. 5.5 shows that the ability of adaptive local search of AQMC is stronger than LQMC.

Furthermore, adaptive local search of AQMC can avoid the deficiency of LQMC that it could lead into "wrong track". Let's consider the following function.

Example 2:

$$f_2 = sA + \sum_{i=1}^{s}[x_i^2 - A\cos(2\pi x_i)], \quad -4.0 \le x_i \le 5.0, \ A \in R \tag{5.5}$$

The global minimum is 0. Here we set $A = 8$. For the dimension $s = 2$, the global minimum point is at $(0, 0)$ and there exist many local minima in the domain $[0, 1] \times [0, 1]$(see Fig. 5.6). For LQMC method, 36320 runs out of 81920 runs(44.33%) can't find the global minimum. For outer procedure of LQMC method, and ε_i is set as Eq. (5.4). If $N = 64$, then the method found a local minimum at the 11^{th} generation and stayed at the local minimum point(see Fig. 5.7, LQMC N=64). If N is increased to 200, then the global minimum was found at the 25^{th} generation (see Fig. 5.7, LQMC N=200),

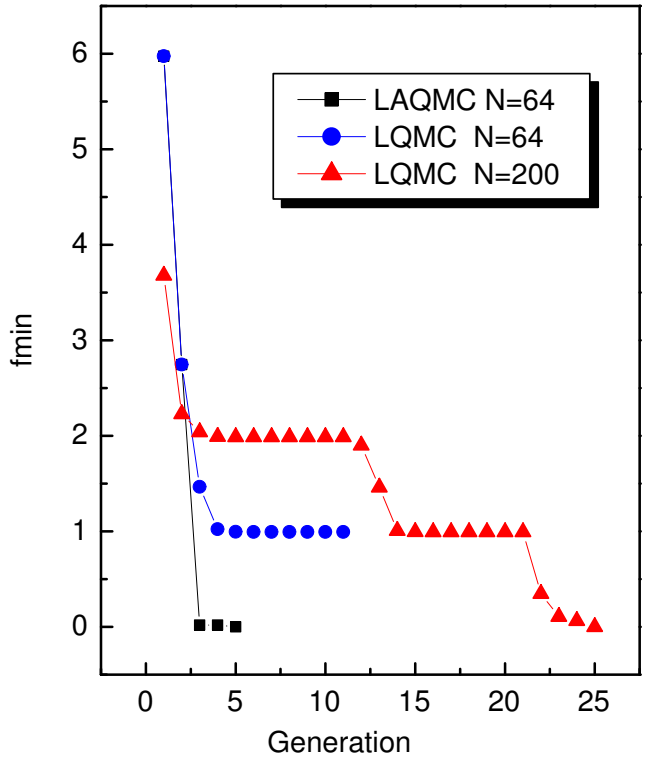

Figure 5.7: Approximation($fmin$) for global minimum of function $f_2(s = 2)$ with LQMC and LAQMC

which means that LQMC calculated 50000 function values before it found the global minimum. Same as the situation of function f_1 , the adaptive local search of AQMC found the global minimum at the 5^{th} generation(see Fig. 5.7, LAQMC N=64). The facts owe to the adaptive search direction and radius.

AQMC is superior to LQMC not only in local search ability, but also in global search ability. For function f_2 defined in Eq. (5.5), LQMC can hardly find the global minimum when $s = 6$. If $N = 1024$ and $\varepsilon_0 = 0.25$, then 8092 runs out of 8192 runs (98.78%) can't find the global minimum. But AQMC can find the global minimum eventually. Table 5.1 shows the results of AQMC methods for f_2 when $s = 6$, where N_p means the total number of calculated function values and $fmin$ means the approximation

Table 5.1: Approximations for the global minimum of $f_2(s = 6)$ with AQMC methods. Compared with the result of LQMC result for dimension 2, the size of N_p is relatively not large.

c_1	c_2	c_3	c_4	N_p	$fmin$
0.040000	1.0	0.0625	0.25	134254	0.0000018688
0.050000	1.0	0.0625	0.25	145119	0.0000052123
0.080000	1.0	0.0625	0.25	190176	0.0000018547

for the minimum of f_2. Remember that LQMC calculated 50000 function values for dimension $s = 2$, here dimension is $s = 6$. So compared with the result of LQMC result for dimension 2, the size of N_p is relatively not large.

Now, we will compare the result of AQMC with that of SNTO. The following two functions are taken from [28].

Example 3:

$$f_3(x, y, z, u) = \exp(xyzu)\sin(x + y + z + u), \quad (x, y, z, u) \in I^4.$$

Example 4:

$$f_4(x, y, z, u) = -(x-\frac{3}{11})^2-(y-\frac{6}{13})^2-(z-\frac{12}{23})^2-(u-\frac{8}{37})^2, \quad (x, y, z, u) \in I^4.$$

We have known that the maximum of f_3 is 1.0261986 and the maximum of f_4 is 0. Tables 3.4 and 3.5 in [28]show that for both f_3 and f_4, the SNTO methods can obtain the error of size $O(10^{-7})$ after calculating more than 2000 function values. However, the same precision is attained only after calculation less than 400 function values for AQMC methods. The improvement is considerable. Tables 5.2 and 5.3 show the results of AQMC methods. All symbols in the tables have been explained in section 5.3.

The computational results show the following:

- AQMC methods are global optimization methods when LQMC and SNTO may lead into "wrong track".
- The local search of AQMC is about 5 times faster than LQMC and SNTO.
- The population size of AQMC is rather less than the sample size of LQMC and SNTO.

Table 5.2: The results of AQMC methods for $f_3(c_1 = 0.5, c_2 = 1.0, c_3 = 0.0625, c_4 = 0.25)$

N_i	$fmax_i$	x	y	z	u
64	1.0170369	0.2187500	0.3437500	0.5312500	0.5937500
32	1.0250066	0.4375000	0.4375000	0.4375000	0.3125000
32	1.0250066	0.4375000	0.4375000	0.4375000	0.3125000
32	1.0256147	0.4340668	0.4310455	0.4357147	0.3428497
32	1.0261741	0.4150982	0.3969021	0.4091587	0.4149303
32	1.0261741	0.4150982	0.3969021	0.4091587	0.4149303
32	1.0261921	0.4139015	0.4027446	0.4100738	0.4123259
32	1.0261969	0.4100674	0.4065787	0.4098912	0.4125084
32	1.0261973	0.4115052	0.4080165	0.4084534	0.4120292
32	1.0261983	0.4106066	0.4085556	0.4100709	0.4104117
352					

Table 5.3: The results of AQMC methods for f_4 $(c_1 = 0.5, c_2 = 1.0, c_3 = 0.015625, c_4 = 0.25)$

N_i	$fmax_i$	x	y	z	u
64	-0.06343331	0.32812500	0.67187500	0.45312500	0.10937500
32	-0.01104883	0.31250000	0.46875000	0.59375000	0.28125000
32	-0.01104883	0.31250000	0.46875000	0.59375000	0.28125000
32	-0.00684690	0.30773926	0.46875000	0.57788086	0.26538086
32	-0.00004844	0.27798462	0.46279907	0.52432251	0.21975708
32	-0.00004844	0.27798462	0.46279907	0.52432251	0.21975708
32	-0.00000507	0.27474183	0.46206683	0.52254421	0.21651429
32	-0.00000211	0.27291776	0.46105346	0.52274688	0.21712231
32	-0.00000013	0.27286076	0.46156648	0.52154983	0.21649529
320					

5.5 Conclusions

As shown in numerical experiments, adaptive quasi-Monte Carlo search method is a global search method. The adaptive technique in local search take advantage of the previous search results and jump from a point to its nearest local extremum point. The local search speeds up the search hugely for function with good quality of continuous, in particular. The examples in this Chapter almost find their global extrema by the adaptive local search. In addition, generating new individuals according to the evolution degree assures the search is a global search.

We suggest that the adaptive local search of AQMC be used for localization of search combined with GA (Genetic Algorithms). The hybrid algorithms will be promising. Chapter 6 gives an example of integrating genetic programming and AQMC method.

Chapter 6

Hybrid of genetic programming and AQMC optimization method

There are many complex systems in real life. In order to analyze, design and predict the system, we often want to model the dynamic systems of ordinary differential equations according to the observed data. In this chapter, we combine genetic programming and adaptive quasi-Monte Carlo optimization method to solve prediction problems. Genetic programming algorithms are used to optimize the right functions of the ordinary differential equations(model structure). Adaptive quasi-Monte Carlo optimization methods are used to optimize the coefficients(model parameters) of the functions.

6.1 Problems

Although there are many complex systems exist in engineering technology, economic management, nature science and social science. problems that demand predictions. The abstract model can be described as that we have the history data, we want to know the data in the future.

$$\begin{pmatrix} x_1(t_1) & \cdots & x_1(t_m) \\ \vdots & \cdots & \vdots \\ x_n(t_1) & \cdots & x_n(t_m) \end{pmatrix} \Rightarrow \begin{pmatrix} x_1(t_{m+1}) & \cdots & x_1(t_{m+num}) \\ \vdots & \cdots & \vdots \\ x_n(t_{m+1}) & \cdots & x_n(t_{m+num}) \end{pmatrix}$$

where $x_i(t_j)$ is the value of i'th variable at time t_j. We want to get the value at time $t_{m+1}, \cdots, t_{m+num}$ from the history data at previous time $t_1, \cdots, t_m$.

Because the variables interact each other, it is a complex system. The ordinary differential equations can be used to describe such dynamic sys-

tems.

$$\begin{cases} \dfrac{\mathrm{d}x_1}{\mathrm{d}t} &= f_1(t, x_1, x_2, \cdots, x_n) \\ \dfrac{\mathrm{d}x_2}{\mathrm{d}t} &= f_2(t, x_1, x_2, \cdots, x_n) \\ \quad\vdots & \\ \dfrac{\mathrm{d}x_n}{\mathrm{d}t} &= f_n(t, x_1, x_2, \cdots, x_n) \end{cases} \tag{6.1}$$

Then we can use Euler method to get the next data from the previous data(regression data).

$$x_i'(t + \Delta t) = x_i(t) + f_i(t, x_1(t), x_2(t), \cdots, x_n(t)) \times \Delta t \tag{6.2}$$

What we have to do is to find good function f_i so that the regression data consist to the history data. The fitness function can be defined as following:

$$fit = \sum_{i=1}^{n} \sum_{j=1}^{m} (x_i'(t_j) - x_i(t_j))^2 \tag{6.3}$$

Once we get the good function f_i which minimizes the above fitness function, we can use Euler method to predict the data in the future.

6.2 Genetic programming

Genetic programming[47][48] is developed from genetic algorithm. It can be used to optimize complex structure such as computer program with dynamic size and structure. Here we want to optimize functions(mathematical expression). The population consist of N individuals. Each individual is a set of n functions. Through the evolution, we get the next generation of population from the previous generation.

The optimization object is equations set, i.e., n functions. Function can be expressed by tree in computer program. The process of evolution of genetic programming is the same as that of genetic algorithm, except that the object of evolution operator is tree, not the chromosomes with fix length as in genetic algorithm. We will explain the expression of tree and the evolution operator on trees later. The evolution progress of genetic programming is as follow:

- Step 1: (initialization)Generate population consisting of N individuals, that is to generate N equations, each equation is n functions(mathematical expression), i.e., n trees;

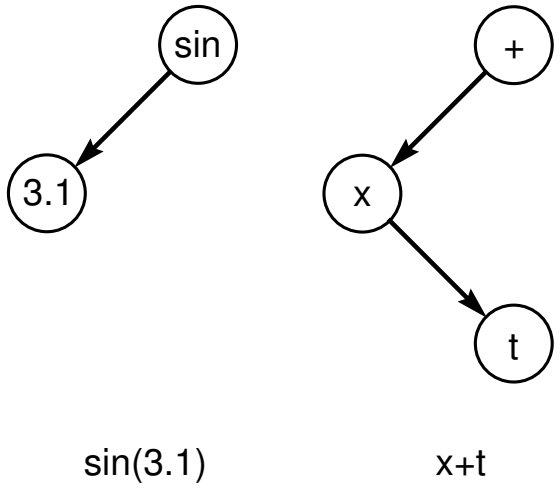

Figure 6.1: Function is expressed by a binary tree. The left child of the math operator node is the first variable of this operator, the right child of the left child of the math operator is the second variable of this operator.

- Step 2: Calculate fitness according to Eq. (6.3);
- Step 3: Get the individuals(trees) of next generation through evolution by duplicate, mutate and crossover operator;
- Step 4: Optimize the coefficients of the functions using AQMC method;
- Step 5: Go to step 2.

We write the program in C++ language [49], in this section we will show the points of the program using part of C++ codes. We first discuss how to express the mathematical expression by tree, then explain how to calculate function values, lastly describe the three evolution operator on trees. The coefficient optimization will be presented in the next section.

Each function is expressed by a binary tree . As shown in Fig. 6.1, if the math operator has only one variable, the tree has only left child, so the left tree in the figure expresses $\sin 3.1$. If the math operator has two variables, the left child of the math operator node is the first variable of this operator, the right child of the left child of the math operator is the second variable of this operator. The right tree in the figure expresses $x + t$, where the second variable t is the right child of the first variable x. We use this kind of binary tree but not to regard the second variable as the child of the math operator in order to generalize the codes. If we regard all the branches as the children of the root node, as for the structure with three branches, for example,

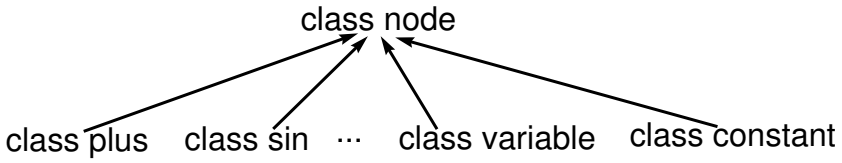

Figure 6.2: The operator class, the constat class and the variable class are derived from the base class node

the computer program, then the root should have three children. As for structure with four branches, the root should have four children, $\cdots$; But we can use the binary tree to express all structures with nondeterministic branches. Set the third branch as the right child of the second branch, the fourth branch as the right child of the third branch $\cdots$.

Tree is expressed by data structure link. Because all the mathematical expressions have been expressed by binary, each node having only two children. Therefore each node only need two pointers point to left child and right child. But our trees have different type nodes including math operator such as "+", "−", "×","/", " sin", "cos", "ln", "exp" and constant(for example, "3.1" in Fig. 6.1) and variable(t, x_1, $\cdots$). These elements can not been expressed by same data structure. We take the advantage of concept "class" of C++ language. Class has three good properties of encapsulation, inherit and polymorphic so that we can realize the link(tree) with heterogeneous nodes. We create a base class "node". The operator class such as "+", "−", "×", "/", "sin", "cos", "ln", "exp", the constat class and the variable class are derived from the base class "node" (Fig. 6.2). All the common properties are defined in the base class. For example, there are two pointer, an int variable "rank" to mark these elements' serial numbers as well. A node with "rank=0" means this node is operator node "+", "rank=8" means this node is constant node, "rank=9" means this node is variable node "t", and so on. In fact, the links don't store the expression such as "+", "t", "3.1" who we can see in Fig. 6.1. The links only store the serial numbers which present different elements. There are three virtual functions in the base class, they are copy_node() who copies node to get a same node, eval() to calculate the function value of the tree whose root is this node, estr() to return the function expression of the tree whose root is this node. Except the same members, each derived class has its respective members. For example, the constant class "constant" has a member to store the value of this constant. Meanwhile, the operate manners for the math operator, constant and variable are not same, the virtual functions in the

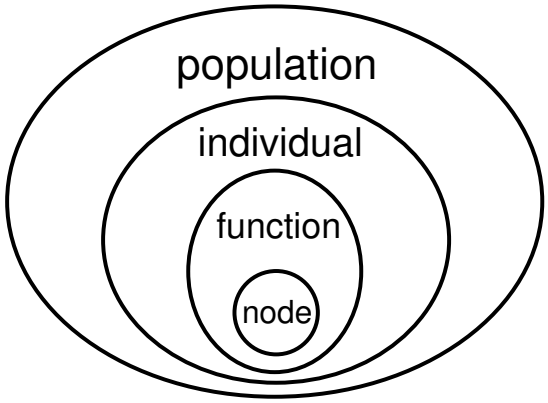

Figure 6.3: Relationship among all classes, class function has class node as its member, and so on $\cdots$

base class will be overrode in the derived classes. The overriding of virtual functions will be explained in the function value calculating.

```
class node
{ public:
    friend class function;
    friend class individual;
    node *right,* left ;
    int rank;//node rank
    int tag;//if operand,tag is the number of variable node(int Rank)
    node(int Rank,int Tag)
   { rank=Rank;
     tag=Tag;
     right=NULL;
     left =NULL;
   }
   virtual node *copy_node(){return NULL;}// copy the node
   virtual double eval(){return 0.;}//get the value of the tree (function)
   virtual char *estr(){return NULL;}// get the expression of the tree
};

class constant:public node
{ public:
   double random_const;
   ...
};
```

The function value calculating is realized by the member function eval() of each node. Every tree has a pointer(node *root) points to its root, through this pointer we can find the left child of the root and the children of this left child, then all the nodes of this tree(tree tour). Once we call the function eval() of the root node, C++ language will automatically call the suit func-

tion according to the class type of the root node. We has mentioned formerly that there are three types of node including math operator, constant and variable, so the virtual function eval() should have different managements for different class type. That is what we call virtual function overriding. For example, as for math operator node "+", the first variable of this node is its left child, the second variable is the right child of its left child, so we should find these two nodes acting as variables of the operator "+" and plus the values of the two sub-trees whose roots are these two nodes.

```
double plus::eval()
      { node *p=left;
        p=p->right;
        return left->eval()+p->eval();
       }
```

It is obvious that the call is a recursion, because the call should calculate function values of two sub-trees.

However, it is easy for constant class to calculate its value, just to get the value stored in its member variable.

```
double constant::eval()
      { return random_const;
      }
```

How to calculate the value of variable is the key of the function value calculating. We have to get the value of variables ("t", "x_1", etc.) from outside and we hope to get the function value of f_1 by directly call function $f_1(x)$ as usual. We use a class "symbol_table" to store the expression and value of the variable in its member "table"(it is an array). There are a member "index" in the class "variable" to index the array "table" of class "symbol_table". For example, "index=0" corresponding to the first element("t") of the array "table", the expression and value of "t" are stored in the first element of array "table".

```
symbol_table sym_tab;
 class symbol_table
 {private:
    struct info
    { char var_name[3];/*to store the expression of the variable*/
      double  var_value;/*to store the value of the variable*/
    };
    info table[n+1];//t,x1,x2
    int table_index;
 public:
    symbol_table();
    void add_value(int index,double r);
    void add_variable(char *s);
    char *get_name(int index);
    double get_value(int index);/*to get the value of the variable by index*/
```

```
    void clear();
};
```

The value of the variable is got by

```
double variable::eval()
{return sym_tab.get_value(index);
}
```

In order to get the value of f_1 by directly calling $f_1(x)$, we have to overload the operator.

```
double function::operator()(double x[])
{double temp;
 for(int i=0;i<=n;i++)
   sym_tab.add_value(i,x[i]);/*x[0]=t, x[1]=x1*/
 temp=root->eval();
 return temp;
}
```

We can see that this routine call the member function add_value() of class "symbol_table" to transform the values of x(x is an array) to array "table". Once the values of variables are updated, call the function eval() of root node of the tree and get the value of the tree(funciton).

Analogy as the function value calculating, we can call root->estr() to get the expression of the tree. The virtual function estr() is also a recursion, it need to be overrode for different class node.

By far, we finish explaining the expression of function and how to calculate the function value. Except that class "node"(node of tree) and class "function"(mathematical function), we need other class such as class "individual"(equations) and class "population"(a set of equations) to build up a whole data structure. These classes have one class as another class' member(see Fig. 6.3), for example, class "node" is the member of class "function", class "function" is the member of class "individual", and so on $\cdots\cdots$.

Now we will describe the operate manner of evolution operators. The three evolution operator including duplicate , mutate and crossover are all handle the trees (functions). Duplicating is to copy the selected tree and get a same tree. Mutating (see Fig. 6.4) first select one node(for example, the node "sin" in Fig. 6.4) randomly, then find the left sub-tree("sin(3.1)") including this node and the sub-tree whose root is the left child of this node, not including the right child(("t")) of this node, and replace this sub-tree by a new generated tree. We should not replace the right child of this selected node because the right child(node "t") of this node is the second variable of its parent("$*$")(please refer to the above description of binary tree). In Fig. 6.4, the sub-tree "sin(3.1)" is replaced by new generated tree "$x_1 + t$", and the origin function "$4.*(\sin(3.1)*t)$" mutates to function "$4.*((x_1+t)*t)$".

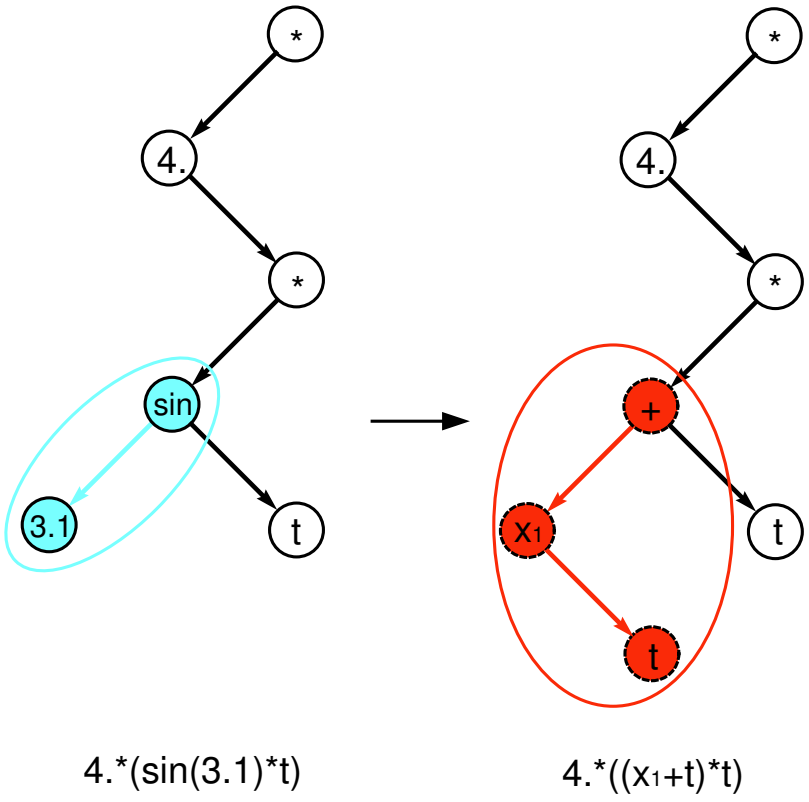

Figure 6.4: Mutate operator of evolution. Select one node of the tree randomly, the left sub-tree of this node "sin(3.1)" replaced by the new generated tree "$x_1 + t$", the right child of this node("t") stay in the same place.

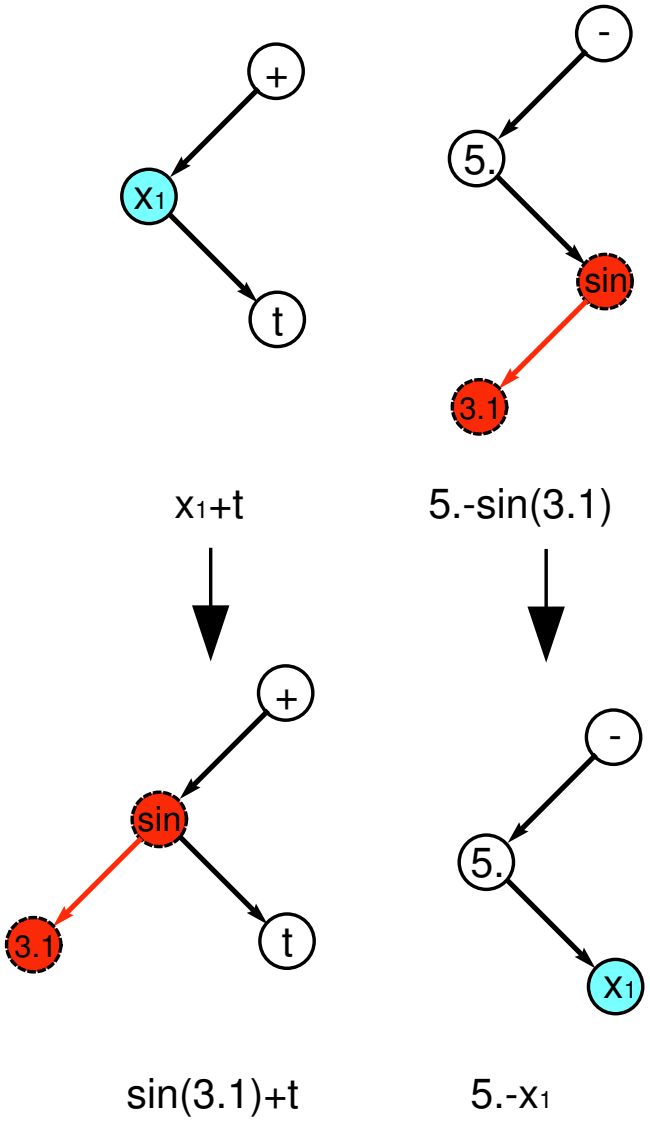

Figure 6.5: Crossover operator of evolution. First respectively select one node "x_1" and "sin" of each of the two trees randomly. Then look for the left sub-trees "x_1" and 'sin(3.1)" of these nodes, exchange these two sub-trees. The right children remain unchanged.

Crossing (Fig. 6.5) first select each node in two trees randomly, then find the left sub-trees and exchange these sub-trees. All operating on tree should remain the right child of the selected node so that the expression of the tree is mathematical valid.

We have to tour tree many times in the programm, stack is need for tree tour. Stack is actually an array with a set of functions so that the data in the stack is first in last out. In order to deal with different type of data with the stack, for example, sometimes we want to deal with pointer, sometimes to deal with numerical value, we use the concept of template. Same as the using of binary tree, template using is to generalize the codes.

```
template <class Tdatatype,int m0> class TStack
{public:
    friend class function;
    TStack(int t_ini,int StackDepth_ini)
   { t=t_ini;
    StackDepth=StackDepth_ini;
   }
    void reset_position(int position)
   { t=position;}
    void push(Tdatatype X);
    void pop();
    void top(Tdatatype *X);
    int sempty();
    Tdatatype ptop();
private:
    Tdatatype s[m0+1];//s[1-m0]
    int t;
    int StackDepth;
};
```

All the program focus on how to deal with complex structure of optimization object(here optimization object is function, we consider how to store and calculate it in computer), this is the task of genetic programming. So we can say the expressing of structure and the operating manner of the evolution operator is the key of genetic programming.

6.3 Coefficients optimization

We imbed coefficient optimization by adaptive quasi-Monte Carlo optimization (AQMC) algorithm described in Chapter 5 in the genetic programming. The function with good structure will be rejected if the coefficients are bad. For example, if the coefficient 3. of variable x_1 in the function (Fig. 6.6) "$\sin(0.5 * t) + 3. * x_1$" be changed to 5., the function fits the system very well. But if we don't do coefficient optimization, the original function with good structure but bad coefficient maybe rejected during the evolution. So we optimize the coefficients once we get a new generation of population.

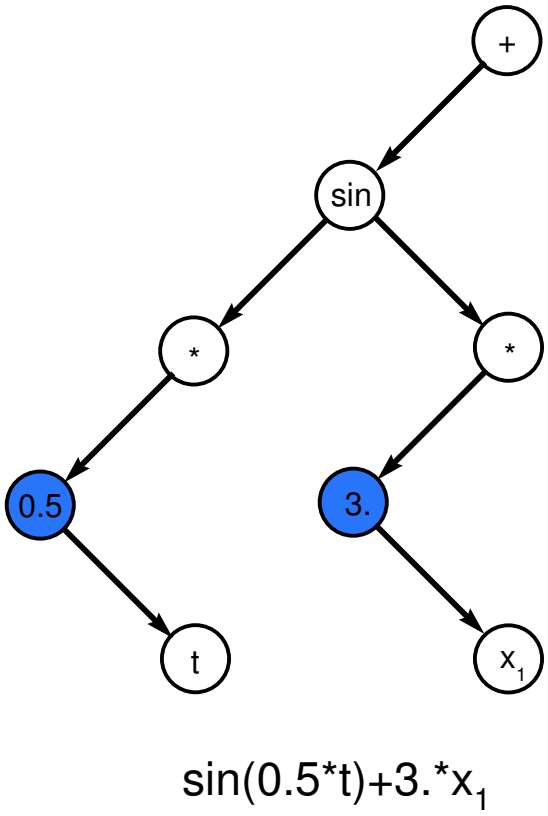

Figure 6.6: Coefficients optimization: tour the tree (function) and trace the address of constant nodes, then use AQMC algorithm to optimize these coefficients

We tour the tree (function) and trace the address of constant nodes, then use AQMC algorithm to optimize these coefficients. The fitness function is the same as that of genetic programming.

6.4 Example

We test our hybrid method by an example of electric power consumption prediction of Hangzhou city (year 2000). We have the electric power consumption from year 1990 to year 1999. We use the method to predict the consumption of year 2000. In fact, we also know the consumption of year 2000. Table 6.1 compared the one predicted data with the original data, the relative error is 4.5% which is less than the practical demanding error(10%).

Table 6.1: Electric power consumption prediction of Hangzhou city (year 2000) 4.5% error

t (year)	x_1 (10^7 wh)	x_1' (10^7 wh)
1990	485278.06	
1991	535014.80	548806.96
1992	604257.63	605577.47
1993	680473.09	684688.80
1994	784010.28	771858.12
1995	851131.16	890411.27
1996	930382.96	967341.10
1997	992755.26	1058242.41
1998	1048329.91	1129830.92
1999	1168747.17	1193650.49
2000	1394604.41	1332031.53

The prediction errors of our runs are almost within 6%, and the time each run is very short. When compared with other prediction methods such as neural network, our method is the most excellent. Other methods can not predict the sudden change of the system. For example, the electric power consumption from year 1999 to year 2000 increased rapidly while the consumption from year 1990 to year 1999 steady. The neural network method can not predict the consumption of year 2000 using the history data. Our method can predict this rapidly change owe to the ordinary differential equations describing complex dynamic systems. And genetic programming is a powerful tool to optimize ordinary differential equations. It is superior to gray system method for the systems with multiple variables.

Chapter 7

The application of AQMC methods to light transport in tissue

Among various methods solving light transport, Monte Carlo simulation has been used widely because it has advantage in dealing with macroscopical tissue and it is easy to be realized and program. Monte Carlo simulation method is time saving when compared with fine methods such as finite difference time domain(FDTD) method and is more suitable to be as forward method when solve the inverse problem of light transport. Monte Carlo simulation of photon propagation offers a flexible yet rigorous approach toward photon transport in turbid tissues. In this chapter, we introduce the Monte Carlo simulation method for light transport in tissue. Then use the Adaptive quasi-Monte Carlo optimization method to solve the inverse problem of light transport in tissue.

7.1 Introduction

The knowledge of light interacting in biological tissue is important when developing new diagnostic methods and medical treatments utilizing lasers. The more these methods are fined, the more detailed studies of the transport of light in tissue are required.

Monte Carlo simulation method is a probability method based on the the [transport equation. When light propagates, two processes — absorption and scattering — occur to various extents due to the medium. A common way of describing the optical properties of the medium is by the refraction index, n, the absorption coefficient, μ_a, the scattering coefficient, μ_s, and the anisotropy coefficient, g. The absorption and scattering coefficients describe the probability per unit path length of a photon being absorbed or scattered,

respectively. The reciprocal of $\mu_a + \mu_s$, $1/(\mu_a + \mu_s)$, is interpreted as the mean free path length between photon interactions with the medium. The quantity $\mu_t = \mu_a + \mu_s$ is called the total attenuation coefficient. The g-factor is defined by the mean value of the cosine of the scattering angle. These optical parameters can be used as input parameters for Monte Carlo simulation and determine the random walk of the photon.

There are fundamental assumptions made throughout this chapter. First, the distribution of light is assumed static with time, and consequently, both optical properties which change and irradiance times shorter than about one nanosecond are excluded. Second, all media are assumed to have homogeneous optical properties. The index of refraction is assumed to be uniform so that light will travel in a straight line until it is scattered or absorbed. A third assumption is that the tissue geometry may be approximated by an infinite plane-parallel slab with finite thickness. This assumption requires that the beam width be smaller than the width of the tissue. The boundaries are assumed smooth and to reflect specularly according to Fresnel's law. Such a shape allows generalization to layered tissues or extension to an infinitely thick tissue. The last assumption is that the polarization of light may be ignored.

7.2 Monte Carlo simulation

The Monte Carlo method relies on tracing photon packet trajectories in a random walk fashion, where the scattering and absorption events are governed by the probabilities given by μ_s and μ_a, as well as the phase function $p(\mathbf{s}, \mathbf{s}')$. The key decisions to be made in a simulation are the mean free path between scattering events, and the scattering angle. In addition, the internal reflection must be handled.

Prahl described the rules of Monte Carlo simulation and give the flowchart (Fig. 7.1) in [50].

A photon will be absorbed or scattering according to μ_a and μ_s, the probability of absorption at any photon interaction site is $\mu_a/(\mu_a + \mu_s)$. Unless μ_a is very low, the probability of photon surviving after a few scattering events is low. This means a very large number of photons have to be traced to yield acceptable accuracy at large distance from the source. Variance reduction techniques are used to improve the accuracy of the simulation for smaller number of photons. Instead of terminating one photon at absorption, photon packets are launched with initial weights (usually set to be 1). A proportion of μ_a/μ_t weight will be absorbed and the rest will be scattered. The packet will be traced for ever till it is terminated using roulette method.

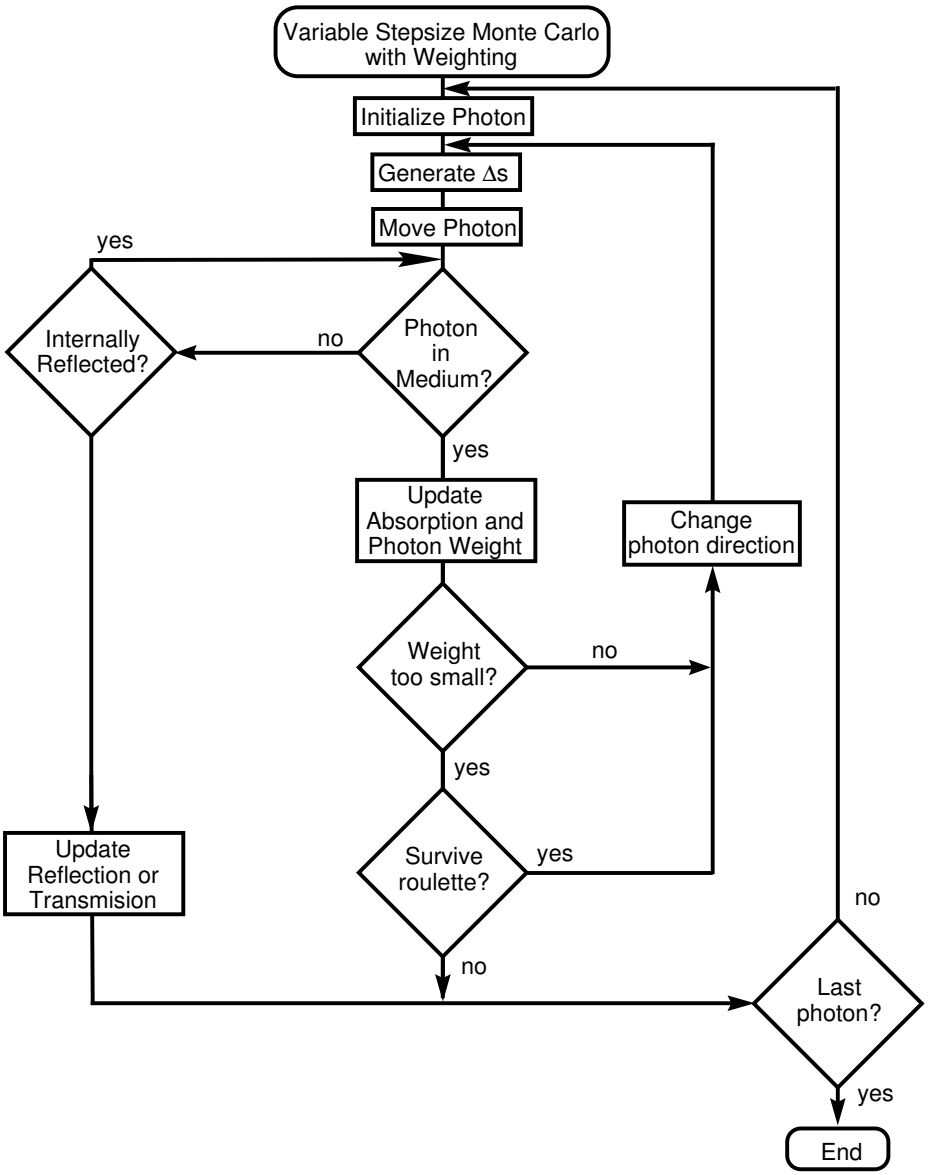

Figure 7.1: Flowchart of Monte Carlo simulation for light transport. Once launched, the photon is moved a distance $\triangle s$ where it may be scattered, absorbed, propagated undisturbed, internally reflected, or transmitted out of the tissue. The photon is repeatedly moved until it either escapes from or is absorbed by the tissue. If the photon escapes from the tissue, the reflection or transmission of the photon is recorded. If the photon is absorbed, the position of the absorption is recorded. This process is repeated until the desired number of photons have been propagated. The recorded reflection, transmission, and absorption profiles will approach true values (for a tissue with the specified optical properties) as the number of photons propagated approaches infinity.

When the weight falls below some threshold value,e.g., 0.001, there is a one in m chance that the photon packet will survive the roulette procedure. In case it survives, its weights is increased m times in that the total amount of launched energy in the simulation is conserved.

The photon packets will travel a distance before the event of absorption and scattering. Here we focus on the generation of the step size $\triangle s$ using inverse continuous cumulative distribution function (CDF) method described in Chapter 2.

According to the definition of μ_t, the probability of interaction per unit path length in the medium between s and $s + \mathrm{d}s$ is $\mu_t \mathrm{d}s$. This can also be expressed in terms of the probability:

$$\mu_t \mathrm{d}s = \frac{-\mathrm{d}P(S \geq s)}{P(S \geq s)} \tag{7.1}$$

Integrating this equation, yields

$$\mu_t s = -\ln P(S \geq s)$$

So

$$P(s) = 1 - \exp(-\mu_t s)$$

using the Eq. (2.7), we get

$$s = \frac{-\ln(1-\xi)}{\mu_t}$$

where ξ is uniformly distributed random number over $[0, 1]$, so the equation is equivalent to

$$s = \frac{-\ln \xi}{\mu_t} \tag{7.2}$$

We use six parameters to describe the position of photon packet, three Cartesian coordinates for the spatial position and three direction cosines for direction of the travel. So for a photon packet located at (x, y, z) travelling a distance $\triangle s$ in the direction (μ_x, μ_y, μ_z), the new coordinates (x', y', z') are given by

$$\begin{aligned} x' &= x + \mu_x \triangle s \\ y' &= y + \mu_y \triangle s \\ z' &= z + \mu_z \triangle s \end{aligned} \tag{7.3}$$

The possibility of internal reflection occurs when the photon is propagated across a boundary into a region with a different index of refraction. We assume that tissue geometry be approximated by a plane-parallel slab

geometry, infinite in the x and y directions with a thickness τ in the z-direction. The probability that the photon will be internally reflected is determined by the Fresnel reflection coefficient $R(\theta_i)$

$$R(\theta_i) = \frac{1}{2}\left[\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)} + \frac{\tan^2(\theta_i - \theta_t)}{\tan^2(\theta_i + \theta_t)}\right] \tag{7.4}$$

where $\theta_i = \cos^{-1}\mu_z$ is the angle of incidence on the boundary and the angle of transmission θ_t is given by Shell's law

$$n_i \sin\theta_i = n_t \sin\theta_t \tag{7.5}$$

where n_i and n_t are the indices of refraction of the medium from which the photon is incident and transmitted, respectively. A random number ξ uniformly distributed between zero and one is used to decide whether the photon is reflected or transmitted. If $\xi < R(\theta_i)$ then the photon is internally reflected, otherwise the photon exits the tissue and the event is recorded as backscattered light (when the photon exits the top) or transmitted light (when it exits the bottom). If the photon is internally reflected, then the internally reflected photon position (x'', y'', z'') is obtained by changing only the z-component of the photon coordinates

$$(x'', y'', z'') = \begin{cases} (x, y, -z) & if \;\; z < 0, \\ (x, y, 2\tau - z) & if \;\; z > \tau. \end{cases} \tag{7.6}$$

The new photon direction (μ'_x, μ'_y, μ'_z) is

$$(\mu'_x, \mu'_y, \mu'_z) = (\mu_x, \mu_y, -\mu_z) \tag{7.7}$$

and both μ_x and μ_y remain unchanged.

If the photon packets are still in the tissue after propagated or internal reflect when across a boundary, it will be absorbed partly. If the weight after absorbed is low than an threshold, we use roulette procedure to determine if the photon will be survived. If the photon packets has large enough weight(need't do roulette) or survive the roulette procedure, they will scatter. The scattering deflection angle θ can be sampled from phase function. The scattering phase function $p(\mathbf{s}, \mathbf{s}')$ describes the angular probability of scattering from direction $\mathbf{s}'$ to $\mathbf{s}$. The phase function is sometimes written as $p(cos\theta)$ to emphasize the angular dependency. The most common phase function in turbid medium is called Henyey-Greenstein phase function [51], with the form:

$$p(\cos\theta) = \frac{(1 - g^2)}{2(1 + g^2 - 2g\cos\theta)^{3/2}} \tag{7.8}$$

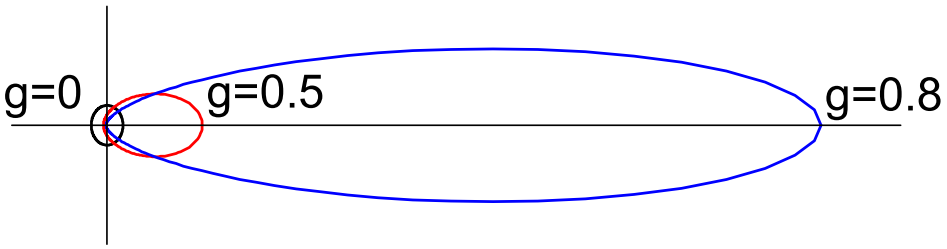

Figure 7.2: The shape of the Henyey-Greenstein phase function for three values of g-factor

where g is called the scattering anisotropy factor. The shape of the Henyey-Greenstein function is shown in Fig. 7.2 for three values of g. Inserting Eq. (7.8) in Eq. (2.7), and solving for $\cos\theta$, yields

$$\cos\theta = \frac{1}{2g}[1 + g^2 - (\frac{1 - g^2}{1 - g + 2g\xi})^2] \tag{7.9}$$

As for the isotropy media, $p(\cos\theta) = \frac{1}{2}$, so

$$\cos\theta = 2\xi - 1 \tag{7.10}$$

The azimuthal scattering angle is uniformly distributed in the interval $0 < \phi < 2\pi$, so we get

$$\phi = 2\pi\xi \tag{7.11}$$

If a photon is scattered at an angle (θ, ϕ) from the direction (μ_x, μ_y, μ_z) in which it is travelling, then the new direction (μ'_x, μ'_y, μ'_z) is specified by

$$\begin{aligned} \mu'_x &= \tfrac{\sin\theta}{\sqrt{1-\mu_z^2}}(\mu_x\mu_z\cos\phi - \mu_y\sin\phi) + \mu_x\cos\theta \\ \mu'_y &= \tfrac{\sin\theta}{\sqrt{1-\mu_z^2}}(\mu_y\mu_z\cos\phi + \mu_x\sin\phi) + \mu_y\cos\theta \\ \mu'_z &= -\sin\theta\cos\phi\sqrt{1-\mu_z^2} + \mu_z\cos\theta \end{aligned} \tag{7.12}$$

If the angle is too close to the normal (say $|\mu_z| > 0.99999$), the following formulas should be used

$$\begin{aligned} \mu'_x &= \sin\theta\cos\phi \\ \mu'_y &= \sin\theta\sin\phi \\ \mu'_z &= \tfrac{\mu_z}{|\mu_z|}\cos\phi \end{aligned} \tag{7.13}$$

to obtain the new photon directions.

The photon package is repeatedly moved until it either escapes from or is absorbed by the tissue(dies in the roulette procedure). If the photon escapes

from the tissue, the reflection or transmission of the photon is recorded. If the photon is absorbed, the position of the absorption is recorded. This process is repeated until the desired number of photons have been propagated. The recorded reflection, transmission, and absorption profiles will approach true values (for a tissue with the specified optical properties) as the number of photons propagated approaches infinity.

7.3 Inverse problem

The photon medical treatments design is to find the optical properties such as the refraction index (n), the absorption (μ_a),the scattering coefficients (μ_s)and the scattering anisotropy factor (g) given that we have measured the propagation light. This is what we say inverse problem of light transport. We use the program Monte Carlo simulation multi-layered media (MCML) [52] as the forward method. The adaptive quasi-Monte Carlo optimization described in Chapter 5 is used to optimize the optical properties.

We mentioned in the introduction that the model of infinite plane-parallel slab with finite thickness can be generalize to layered tissue. MCML program is to deal with multi-layered tissue. Except that the above described procedure, MCML also consider the internal reflection and transmission between the layers. To be simplex, we only set the layer to be 1, the above medial and below media are all glass. We define the fitness function as the following

$$fit = \sum_{n=0}^{num-1} (Tr[n] - Tr_0[n])^2 \tag{7.14}$$

Where Tr_0 is the transmission profile(or reflection profile, but it is easy to measure transmission profile in experiments)measured in the experiments. If there is no experiment data, we can use the simulation data when the optical properties are set as the exact values. Here Tr_0 is the transmission profile in one Monte Carlo simulation when set the optimal parameters as the known optimal parameters. Tr is the transmission for one run when the optical properties ranged in the domain. So we have to find the best parameters which minimize the fitness function fit. For each run of MCML, randomly select optical parameters as the input parameters of MCML and get the transmission profile, then calculate the value of function fit.

We study the demand of local search and generating new individuals when we use adaptive quasi-Monte Carlo(AQMC) optimization method. For this practical physical problem, the values of the fitness function changes sharply and there are a great deal of minima. We need to generate much more individuals during the search process. The AQMC method introduced

in chapter 5 focus on local search, though it generate new individuals according to the evolution degree. The concept of evolution degree represents the saturation of local search ability, the probability of generating new individuals will be large only when it is hard for local search to find better extremum. In addition, the evolution degree will be reset to be 0 once a set of new random points is generate and the population is updated. Even if a much more less approximation for the minimum is find when generate new individuals, the program have to jump to local search, not continue generating new individuals. Therefore, this method is slow to find global extremum for the problem who need to generate more new individuals because this method generate new individuals according to evolution degree.

We need to find a measurement to present the ability of local search and generating new individuals. The measurement should consider the improvement on individuals. If local search finds better extremum point, the probability of local search should be increased; if better extremum point is find when generating new individuals, the probability of generating new individuals should be increased. We use the concept of performanc/cost ratio to present the search ability.

Definition 7.3.1 (performanc/cost ratio) *Denote $max\_initial$ as the maximal fitness of the initial population. If it find new maximum $max\_current$ after calculating N_c funciton values, define performanc/cost ratio as:*

$$eff = (max\_current - max\_initial)/N_c$$

Then we can calculate the performanc/cost ratio of local search , $lseff$, and performanc/cost ratio of generating new individuals, $nieff$, respectively. The probability of generating new individuals $newp$ can be determined by these two ratio. The weights, lsw and niw, are used to decide the proportion of the two search process. So we have

```
if( lseff <1.0E−10 && nieff< 1.0E−10)
   newp=niw;
else
   newp=niw∗nieff/(lsw∗lseff+niw∗nieff);
```

and do local search or generate new individuals according the probability, that is generate random number rnd, if $rnd < newp$, then generate a set of new individuals.

```
do{
   rnd=(double)(rand())/RAND_MAX;
   if(rnd<newp)
    { /∗generate new individual∗/
      ...
    }
   else
     { /∗ local  search∗/
```

Table 7.1: One example of AQMC optimization algorithms solving for inverse problem of light transport in tissue

	n	μ_a	μ_s	g
origin	1.375	1	100	0.9
approximation	1.375	1.13	101.	0.9

```
    ...
  }
}while(fglobalmax[S]<−1.0E−5);
```

The full codes of the optimization are in appendix.

For the tissue with 1cm thickness of the slab and optical parameters $n = 1.375$, $\mu_a = 1$, $\mu_s = 100$, $g = 0.9$, we set the domain of the four parameters as $n \in [1.0, 2.0]$, $\mu_a \in [0.01, 3]$ (the absorption coefficients can not be set as zero to avoid infinite loop of the program), $\mu_s \in [2, 200]$ and $g \in [0.0, 1.2]$ (in fact g is physically ranged in [-1, 1], here we use the positive values and extend the maximal value 1 to 1.2 to avoid too much invalid value in local search). We got the approximation values of these four optical properties(Fig. 7.1) after 263 MCML runs.

The optimization method can also be used for multi-layered tissue, but the results are not as good as that of the one layer tissue. Compared with the numerical experiments in chapter 5, we can see that we should study the property of problem for real physical optimization. We hope to do some research using real experiment data to study the demands of local search and global search for Monte Carlo optimization method and find an efficient measurement.

Chapter 8

AppendixC codes

8.1 C code of Sobol'sequence generator

This program is modified from [26], it can generate maximum 160-dimension Sobol' sequence. The input file sobol_para.txt stores the values of array mdeg[](the degree of primitive polynomial) and ip[](the value of binary express consisting of the coefficints of the primitive polynomial).

```
#include "math.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXBIT 30
#define MAXDIM 160
#define S 2

/*When n is negative, internally  initializes  a set  of
MAXBIT direction numbers for each of MAXDIM different
Sobol' sequences. When n is positive (but <= MAXDIM),
returns as the vector x[1..n] the next values from n
of these sequences.
(n must not be changed between  initializations .)*/

int IMIN(int a,int b)
{
    return a<b ? a:b;
}

void sobseq(int *n, double x[])
{ FILE *fr1;
  int j,k,l;
  unsigned long i,im,ipp;
  static double fac;
  static unsigned long in,ix[MAXDIM+1],*iu[MAXBIT+1];
```

```
static unsigned long mdeg[MAXDIM+1];
static unsigned long ip[MAXDIM+1];
static unsigned long iv[MAXDIM*MAXBIT+1];
long temp;

if(*n < 0)
{ //read mdeg[] and ip[]
 if((fr1=fopen("sobol_para.txt","r"))==NULL)
 {
      printf("\nFile_not_found!");
      exit(0);
 }
 for(i=1;i<=MAXDIM;i++)
      fscanf(fr1,"%d",&mdeg[i]);//read mdeg[]
 for(i=1;i<=MAXDIM;i++)
      fscanf(fr1,"%d",&ip[i]);//read ip[]
 fclose(fr1);
 //set the values of iv[]//
 srand((unsigned)time(NULL));//give the rand seed
 for(i=1;i<=mdeg[MAXDIM];i++)
 {
  for(j=1;j<=MAXDIM;j++)
  { temp=2*(int((1L<<(i-1))*float(rand())/RAND_MAX)+1)-1;
    iv[(i-1)*MAXDIM+j]=temp;
  }
 }
 /* Initialize , don't return a vector.*/
 for (k=1;k<=MAXDIM;k++)
      ix[k]=0;
 in=0;
 if (iv[1] != 1)
      return;
 fac=1.0/(1L << MAXBIT);
 for (j=1,k=0;j<=MAXBIT;j++,k+=MAXDIM)
      iu[j] = &iv[k];
 /*To allow both 1D and 2D addressing.*/
 for (k=1;k<=MAXDIM;k++)
 {
      for (j=1;j<=mdeg[k];j++)
           iu[j][k] <<= (MAXBIT-j);
      /*Stored values only require normalization.*/
      for (j=mdeg[k]+1;j<=MAXBIT;j++)
     {
           /*Use the recurrence to get other values.*/
           ipp=ip[k];
           i=iu[j-mdeg[k]][k];
           i ^= (i >> mdeg[k]);
           for (l=mdeg[k]-1;l>=1;l--)
          {
                if (ipp & 1)
                     i ^= iu[j-l][k];
                ipp >>= 1;
          }
           iu[j][k]=i;
```

```
        }
    }
  }
  else
  {
    /*Calculate the next vector in the sequence.*/
    im=in++;
    for (j=1;j<=MAXBIT;j++)
    {
        /*Find the rightmost zero bit.*/
        if (!(im & 1))
            break;
        im >>= 1;
    }
    if (j > MAXBIT)
        printf("\nMAXBIT too small in sobseq\n");
    im=(j-1)*MAXDIM;
    for (k=1;k<=IMIN(*n,MAXDIM);k++)
    {
      /*XOR the appropriate direction number into each
      component of the vector and convert to a floating
      number.*/
      ix[k] ^= iv[im+k];
      x[k]=ix[k]*fac;
    }
  }

}


void main( )
{FILE *fw1;
static double xsob[S+1];
int i,j;
int ini,run;
ini=-1;
sobseq(&ini,xsob);
if( (fw1 = fopen( "sobol_points", "w" )) == NULL )
 { printf( "The file 'sobol_points' was not opened\n" );
   exit(0);
 }
for(i=1;i<=1024;i++)
{run=S;
sobseq(&run,xsob);
for(j=1;j<=S;j++)
    fprintf(fw1,"%f\t",xsob[j]);
fprintf(fw1,"\n");
}//end of i
fclose(fw1);
}

// begin of input file sobol_para.txt
1
2
```

```
3  3
4  4
 5  5  5  5  5  5
 6  6  6  6  6  6
 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7
 8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9
 9 9 9 9 9 9 9 9 9 9 9
 9 9 9 9 9 9 9 9 9 9 9
 9 9 9 9 9 9 9 9 9 9 9
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
0
1
1  2
1  4
 2  4  7  11  13  14
 1  13  16  19  22  25
   1  4  7  8  14  19  21  28  31  32  37  41  42  50  55
56  59  62
  14  21  22  38  47  49  50  52  56  67  70  84  97  103
115  122
  8  13  16  22  25  44  47  52  55  59  62  67  74  81
 82  87  91  94  103  104  109  122 124  137  138  143
 145  152  157  167  173  176  181  182  185  191  194
 199  218  220  227  229  230  234  236  241  244  253
  4  13  19  22  50  55  64  69  98  107  115  121  127
 134  140  145  152  158  161  171  181  194  199  203
 208  227  242  251  253  265  266  274  283  289  295
 301  316  319  324  346  352  361  367  382  395  398
 400  412  419  422  426  428  433  446  454  457  472
493  505  508
// end of input  file
```

8.2 C code of Halton sequence generator

This program is traslated into C language by me from the Fortan codes written according to [53].

```
//============the program to generate Halton sequence====//
#include "math.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#define MAXBIT 64
#define MAXDIM 40
#define MAXNUM 1000
#define S 2  /*dimension*/

/*input:n, if the first time to run the subroutine, *n=-S;*/
void halseq(int *n, double x[])
{   int j,s;
    static double prime[MAXDIM+1]={0.0,2.0,3.0,5.0,7.0,11.0,13.0,17.0,19.0,23.0,\
                    29.0,31.0,37.0,41.0,43.0,47.0,53.0,59.0,61.0,67.0,71.0,73.0,\
                      79.0,83.0,89.0,97.0,101.0,103.0,107.0,109.0,113.0,127.0,\
                        131.0,137.0,139.0,149.0,151.0,157.0,163.0,167.0,173.0};

    static double xhal[MAXDIM+1],E,Delta,Tiny=1.0;
    static unsigned char Flag[2]={0,0};
    double T,F,G,H;
    if(*n < 0)
    { /*FIRST CHECKS WHETHER THE USER-SUPPLIED DIMENSION "
        DIMEN"
       OF THE QUASIRANDOM VECTORS IS ACCEPTABLE
      ( STRICTLY BETWEEN 0 AND 41):
       IF SO, FLAG(1)=.TRUE.*/
       s=-*n;/*the dimension*/
       Tiny=1L>>MAXBIT;
       if((s>=1)&&(s<=40))
           Flag[0]=1;
       else
      {
           printf("The dimension must between 1 and 40!\n");
           exit(0);
      }
       /* COMPUTE AND CHECK TOLERANCE*/
       E = 0.9* (1.0/ ((double)MAXNUM*prime[s])-10.0*Tiny);
       Delta = 100.0*Tiny*(double)(MAXNUM+1)*log10((double)MAXNUM);
       if(Delta <= (0.09* (E-10.0*Tiny)))
           Flag[1]=1;
       else
      {
           printf("The dimension must between 1 and 40!\n");
           exit(0);
      }
       /*NOW COMPUTE FIRST VECTOR*/
       for(j=1;j<=s;j++)
      {
           prime[j]=1.0/prime[j];
           xhal[j]=prime[j];
           x[j]=xhal[j];
      }
    }

    else
    {
        s=*n;
        for(j=1;j<=s;j++)
```

```
        {
            T = prime[j];
            F = 1.0 - xhal[j];
            G = 1.0;
            H = T;
            while ((F-H)<E)
           {
                G = H;
                H = H*T;
           }
            xhal[j] = G + H - F;
            x[j]=xhal[j];
        } /*end of for(j=1;j<=s;j++)*/
    } /*end of else*/
}

void main( )
{
    FILE *result;
    double xhal[S+1];
    int i,j;
    int ini,run;
    int num_random=120;
    if( ( result  = fopen( "halton.dat", "w" )) == NULL )
    {    printf( "The file 'halton.dat' was not opened\n" );
         exit(0);
    }
    fprintf(result,"Halton sequence .....\ n");
    /* initialize  and generate the  first  random number*/
    ini=-S;
    halseq(&ini,xhal);
    for(j=1;j<=S;j++)
         fprintf(result,"%f\t",xhal[j]);
    fprintf(result,"\n");
    fclose(result);
    for(i=2;i<=num_random;i++)
    {  if( ( result  = fopen( "halton.dat", "a" )) == NULL )
        {    printf( "The file 'halton.dat' was not opened\n" );
             exit(0);
        }
         run=S;
         halseq(&run,xhal);
         for(j=1;j<=S;j++)
            fprintf(result,"%f\t",xhal[j]);
         fprintf(result,"\n");
         fclose(result);
    } //end of i
}
```

8.3 Parallel programming for MC, AMC and AMC

This MPI parallel program can be run on computer cluser. All the codes except the sub-routine from [26] are written by the author, all right resevered.

```
/*
 Copyright 2003 by Guiyuan Lei
 All rights reserved. All codes except the ran2() in this section can be used in the
        case that user cite this thesis.
 Monte Carlo integration in parallel programming(MPI)
 all the processes use a same sequence,
 passing the random seeds and serial of random number(n) through the processes.
 the pseudorandom sequence is generated by rans()
 from book "Numerical recipes in C: The art of scientific computing"
 seeds:
 static long idum2=123456789;
 static long iy=0;
 static long iv[NTAB];
*/
#include "math.h"
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAXBIT 64
#define MAXDIM 40
#define MAXNUM 1000000
#define S 15  //dimension=15
#define Pi 2*arcsin(1.)
/*length of sub-sequence generated by each process*/
#define Max_Len 32768

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
/*Uniformally distributed random sequence generator
  p282 Chapter 7. Random Numbers in "Numerical Recipes in C"
*/
static long idum2=123456789;
static long iy=0;
```

```
static long iv[NTAB];

/*Long period (> 2 * 10^18 ) random number generator of L'Ecuyer with Bays-
        Durham shuffle
  and added safeguards. Returns a uniform random deviate between 0.0 and 1.0 (
        exclusive of
  the endpoint values). Call with idum a negative integer to initialize ; thereafter ,
        do not alter
  idum between successive deviates in a sequence. RNMX should approximate the
        largest oating
  value that is less than 1.
*/
float ran2(long *idum)
{
    int j;
    long k;
    float temp;
    if (*idum <= 0)
    { /* Initialize */
        /*Be sure to prevent idum = 0*/
        if (-(*idum) < 1)
          * idum=1;
        else
          * idum = -(*idum);
        idum2=(*idum);
        for (j=NTAB+7;j>=0;j--)
        { /*Load the shuffle table (after 8 warm-ups)*/
            k=(*idum)/IQ1;
          * idum=IA1*(*idum-k*IQ1)-k*IR1;
            if (*idum < 0)
              * idum += IM1;
            if (j < NTAB)
                iv[j] = *idum;
        }
        iy=iv[0];
    }
    /*Start here when not initializing .*/
    k=(*idum)/IQ1;
    /* Compute idum=(IA1*idum) % IM1 without over ows by Schrage's method.*/
   *idum=IA1*(*idum-k*IQ1)-k*IR1;
    if (*idum < 0)
      * idum += IM1;
    k=idum2/IQ2;
    /*Compute idum2=(IA2*idum) % IM2 likewise.*/
    idum2=IA2*(idum2-k*IQ2)-k*IR2;
    if (idum2 < 0)
        idum2 += IM2;
    j=iy/NDIV;/*Will be in the range 0..NTAB-1.*/
    iy=iv[j]-idum2;/* Here idum is shuffled, idum and idum2 are combined to generate
        output.*/
    iv[j] = *idum;
    if (iy < 1)
        iy += IMM1;
    if ((temp=AM*iy) > RNMX)
```

```
        return RNMX; /*Because users don't expect endpoint values.*/
    else
        return temp;
}

/*the integrand from:
  Math. Comput. Modelling Vol. 23, No, 8/9, pp. 87-96, 1996
  p92 Example2,Figure2,5
*/
double f(double x[])
{/*use the first S dimension of x[], just 1-S elements*/
    double f;
    int i;
    double sum_x=0.;
    for(i=1;i<=S;i++)
        sum_x+=x[i]/i;
    f=exp(sum_x);
    return f;
}
/*Calculating function value and sum the value usint CMC,AMC,FAMC method*/
void MC(long temp_serial, int n, double x[],double interval,double sum[])
{
    int l;
    double f_x;
    double x_k[S+1];
    double c_k[S+1];
    f_x=f(x);/*function value of x*/
    sum[1]+=f_x;/*Crude Monte Carlo estimate*/
    sum[2]+=f_x;
    for(l=1;l<=S;l++)
        c_k[l]=1.0- x[l];/*the antithetic variables of x*/
    sum[2]+=f(c_k);/*Antithetic Monte Carlo estimate*/
    for(l=1;l<=S;l++)
    { /*The domain of function is devided into N=n^S subcube D_k
                c_k[l]: leftest border of subcube k*/
        c_k[l]=temp_serial%n;
        temp_serial=temp_serial/n;
        x_k[l]=(c_k[l]+x[l])*interval;
    }
    sum[3]+=f(x_k);
    for(l=1;l<=S;l++)
    { /*Here c_k[l] is centre of subcube k
            interval is the interval of subcube*/
        c_k[l]=(0.5+c_k[l])*interval;
        x_k[l]=2*c_k[l]- x_k[l];/*the antithetic variables*/
    }
    sum[3]+=f(x_k);/*Fine Antithetic Monte Carlo estimate*/
}
/*end of calculating function value usint CMC,AMC,FAMC method*/
/*Linear fit of the data (X,T), least square error method*/
void Fit_linear(long X[],double T[],int count,double A[])
{
    int i;
    double Xi;
```

```
    double Ti;
    int sum_true=1;/*if sum the data*/
    double sum_Xi=0.0;
    double sum_Xi_square=0.0;
    double sum_Ti=0.;
    double sum_Xi_Ti=0.;
    double a;/*slope*/
    double b;/*intercept*/
    for(i=1;i<=count;i++)
    {
        sum_true=1;/*to sum the data*/
        Ti=T[i];
        if(Ti<1.0E-20)
            sum_true=0;
        /*to calculate slope*/
        if(sum_true)
        {
            Xi=log10(X[i]);
            sum_Xi+=Xi;/*sum x*/
            sum_Xi_square+=Xi*Xi;/*sum x^2*/
            Ti=log10(T[i]);
            sum_Ti+=Ti;/*sum T*/
            sum_Xi_Ti+=Xi*Ti;/*sum x*T*/
        } /*end of if(sum_true)*/
        else
            count--;
    } /*end of i(step)*/

    a=(count*sum_Xi_Ti-sum_Xi*sum_Ti)/(count*sum_Xi_square-sum_Xi*sum_Xi);
    b=(sum_Ti*sum_Xi_square-sum_Xi*sum_Xi_Ti)/(count*sum_Xi_square-sum_Xi*
        sum_Xi);
    A[0]=a;
    A[1]=b;
}
/*Empirical standard deviate(sd) error*/
double sd_error(double s_run[],int runs)
{
    int l;
    double temp;
    double proximate_int_average=0.;
    for(l=0;l<runs;l++)
        proximate_int_average+=s_run[l];
    proximate_int_average/=runs;
    temp=0.0;
    for(l=0;l<runs;l++)
        temp+=(s_run[l]-proximate_int_average)*(s_run[l]-proximate_int_average);
    temp=sqrt(temp/(runs-1));
    return temp;
}
/*Empirical root mean square error(rmse)*/
double rmse(double s_run[],double exact_int,int runs)
{
    int l;
    double temp=0.;
```

```
    for(l=0;l<runs;l++)
        temp+=(s_run[l]−exact_int)*(s_run[l]−exact_int);
    temp=sqrt(temp/runs);
    return temp;
}

main(int argc,char** argv)
{
    FILE *f_value,*f_rmse, *f_sd;
    char str[20]; /* file name*/
    double s_step_run [4][4][76];
    double variance_step[4];
    int step=3;
    long points_step[4];
    long N;/*number of random point points*/
    int runs=75;/*compute the root mean square error over 75 runs*/
    /*the sum of function value for Crude MC, AMC and FAMC methods*/
    double sum[4],G_sum[4];
    long i,j,k,m;

    int l;/*index for dimension*/
    int n;/*number of sub−interval*/

    double interval;
    double exact_int=5.6102534948577798;/*for s=15*/
    /* exact_int=3.0060133559748561; //for s=4*/
    long ini;

    double A[2];/*the coefficient of linear fit */
    /*each process calculate sub_seq_len random points every procs_step random points
        */
    long procs_step;
    long size,remainder;/*size, loop size*/
    int sub_seq_len=Max_Len;/*no more than Max_Len*/
    double random[Max_Len][S+1];

    int nid,nid_before,nid_after,noprocs,last_nid;/*for parallel programming*/
    MPI_Status status;
    long seeds[NTAB+3];/*iv[NTAB],idum,iy,ini*/
    MPI_Request req_send_seeds,req_recv_seeds;

    /* call MPI initialization*/
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&nid);
    MPI_Comm_size(MPI_COMM_WORLD,&noprocs);

    for(j=2;j<=step+1;j++)
    {
        N=1;
        for(i=0;i<S;i++)
            N*=j;
        points_step[j−1]=N;/*set calculating points for each step*/
    }
```

```
 if(nid==noprocs-1)
{
     for(i=1;i<=3;i++)/*deal with three methods:CMC=1,AMC=2,FAMC=3*/
    {  sprintf(str,"f_value-%d.dat",i);
          f_value=fopen(str, "w" );
          for(j=1;j<=step;j++)
               fprintf(f_value,"%ld\t\t",points_step[j]);
          fprintf(f_value,"\n");
          fclose(f_value);
    }
     ini=-S;/*the value of ini will be changed by ran2()*/
     for(m=1;m<=2500;m++)/*ingore the first 2500 random number*/
          ran2(&ini);
     /*envelope the seeds*/
     for(m=0;m<NTAB;m++)
          seeds[m]=iv[m];
     seeds[m]=idum2;
     seeds[m+1]=iy;
     seeds[m+2]=ini;
     nid_after=(nid+1)%noprocs;
     MPI_Isend(seeds,NTAB+3,MPI_LONG,nid_after,10,MPI_COMM_WORLD,&
      req_send_seeds);
}
/*generate next random numbers and perform integration*/
for(i=0;i<runs;i++)
{
     for(j=1;j<=step;j++)
    {
          N=points_step[j];
          n=j+1;
          interval=1./n;
          /*calculate the sum on distributed noprocs computer*/
          for(k=0;k<4;k++)
               sum[k]=0.;
          /*the number of random points each process should calculate*/
          procs_step=sub_seq_len*noprocs;
          remainder=N%sub_seq_len;
          size=N-remainder;
          last_nid=noprocs-1;
          /*============loop for size===========*/
          for(k=nid*sub_seq_len;k<N;k+=procs_step)
        {  /*receive the random seeds*/
               if(nid)   /*nid_before=(nid+noprocs-1)%noprocs;*/
                    nid_before=nid-1;
               else
                    nid_before=last_nid;
               MPI_Irecv(seeds,NTAB+3,MPI_LONG,nid_before,10,
      MPI_COMM_WORLD,&req_recv_seeds);
               MPI_Wait(&req_recv_seeds,&status);
               /*unenvelope the seeds*/
               for(m=0;m<NTAB;m++)
                    iv[m]=seeds[m];
               idum2=seeds[m];
               iy=seeds[m+1];
```

```
        ini=seeds[m+2];

        if(k+sub_seq_len<=size)
    {
          for(m=0;m<sub_seq_len;m++) /*generate sub-sequence*/
           {     for(l=1;l<=S;l++)
                   random[m][l]=ran2(&ini);
           }
    }
        else
    {
          for(m=0;m<remainder;m++) //generate remainder points*/
         {
               for(l=1;l<=S;l++)
                    random[m][l]=ran2(&ini);
         }
    }
        /*send the seeds
          envelope the seeds*/
        for(m=0;m<NTAB;m++)
            seeds[m]=iv[m];
        seeds[m]=idum2;
        seeds[m+1]=iy;
        seeds[m+2]=ini;

        nid_after=(nid+1)%noprocs;
        /*in general the (noprocs-1)'th process send the seeds to process 0*/
        last_nid=noprocs-1;
        /*the process who deal with the N'th random number will send the
seeds to process 0
          then process 0 start the first random number of new successive N
random points
        */
        if((k+sub_seq_len==N)||k+remainder==N)
    {
          nid_after=0;
          last_nid=nid;
          /*boadcast so that process 0 know the change of last_nid*/
          MPI_Bcast(&last_nid,1,MPI_INT,nid,MPI_COMM_WORLD);
    }
        MPI_Isend(seeds,NTAB+3,MPI_LONG,nid_after,10,
MPI_COMM_WORLD,&req_send_seeds);
        /* calculate the function value and sum them*/
        if(k+sub_seq_len<=size)
    {
          for(m=0;m<sub_seq_len;m++)
              MC(k+m,n,random[m],interval,sum);
    }
        else
    {
          for(m=0;m<remainder;m++)
              MC(k+m,n,random[m],interval,sum);
    }
  } /*end of k: loop size of sub_seq_len*/
```

```
        /*send the data to process 0*/
        MPI_Reduce(sum,G_sum,4,MPI_DOUBLE,MPI_SUM,0,
    MPI_COMM_WORLD);
        if(nid==0)
        {
            for(k=1;k<=3;k++)
                s_step_run[k][j][i]=G_sum[k];
        }
    } /*end of j: step*/
     if(nid==0)
    {
        for(j=1;j<=step;j++)
        {
            N=points_step[j];
            s_step_run [1][j][i]=s_step_run [1][j][i]/N;
            for(k=2;k<=3;k++)
                s_step_run[k][j][i]=s_step_run[k][j][i]/N/2.;
        }
        for(k=1;k<=3;k++)/*deal with three method:CMC=1,AMC=2,FAMC=3*/
        {  sprintf(str,"f_value-%d.dat",k);
            f_value=fopen(str, "a" );
            for(j=1;j<=step;j++)
                fprintf(f_value,"%.20f\t",s_step_run[k][j][i]);
            fprintf(f_value,"\n");
            fclose(f_value);
        }
    } /*end of if(nid==0)*/
} /*end of i :runs*/

 if(nid==0)
{
     for(k=1;k<=3;k++)
    {
        sprintf(str,"rmse-%d.dat",k);
        f_rmse= fopen(str, "w" );
        for(i=1;i<=step;i++)
        {
            fprintf(f_rmse,"%ld\t",points_step[i]);
            variance_step[i]=rmse(s_step_run[k][i], exact_int,runs);
            fprintf(f_rmse,"%.10f\t",variance_step[i]);
            fprintf(f_rmse,"\n");
        }
        Fit_linear(points_step,variance_step,step,A);
        fprintf(f_rmse,"\nFit_Linear...\n");
        fprintf(f_rmse,"Y=%.10f*X+%.10f\n",A[0],A[1]);
        fclose(f_rmse);
    } /*end of k*/
     for(k=1;k<=3;k++)
    {
        sprintf(str,"sd_error-%d.dat",k);
        f_sd= fopen(str, "w" );
        for(i=1;i<=step;i++)
        {
```

```
                fprintf(f_sd,"%ld\t",points_step[i]);
                variance_step[i]=sd_error(s_step_run[k][i],runs);
                fprintf(f_sd,"%.10f\t",variance_step[i]);
                fprintf(f_sd,"\n");
            }

            Fit_linear(points_step,variance_step,step,A);
            fprintf(f_sd,"\nFit_Linear...\n");
            fprintf(f_sd,"Y=%.10f*X+%.10f\n",A[0],A[1]);
            fclose(f_sd);
        } /*end of k*/
    } /*end of if(nid==0),analysize*/
    MPI_Finalize();/*end of MPI*/
}
```

8.4 Code of AQMC solving inverse problem of light transport

All the codes except the MCML codes are written by the author, all right resevered.

This program use Monte Carlo Multi-Layer simulation as the forward method. AQMC method is used to solve the inverse problem of the light transport. The probabilities of local search and generating new individuals are decided by the performance/cost ratios.

```
/*****************************
Copyright 2003 by Guiyuan Lei
All rights reserved. All codes except the MCML in this section can be used in the case
        that user cite this thesis.
AQMC for inverse problem of light transport(optimize the optical parameters)
We use the Monte Carlo Multi-Layer(MCML) simulation programmer as the forward
        method.

The Monte Carlo random search method for the global maximum
(if to find the global minimum, use the minus of the fitness function).
Do local search or generate new individuals according to their performance/cost ratio.
We use the Sobol' sequence in optimization(so we need sobol.c file in compiling)

You can download MCML programs from the website
http://omlc.ogi.edu/software/mc/index.html

MCML's inputfile is sample.mci, output file is sample.mco
in this program we use sample.mco as the inputfile and do optimization
all the file list:
      mcmlaqmc.c  //this file
      sobol.h
      sobol.c     //sobol generator
      mcml.h
      mcmlgo.c
```

```
        mcmlio.c
        mcmlnr.c
*/


#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "mcml.h"
#include "sobol.h"

#define N 100
#define S 4

/*Declare before they are used in main()*/
void sobseq(int *n, double x[]);

FILE *GetFile(char *);
short ReadNumRuns(FILE* );
void ReadParm(FILE* , InputStruct * );
void CheckParm(FILE* , InputStruct * );
void InitOutputData(InputStruct, OutStruct *);
void FreeData(InputStruct, OutStruct *);
double Rspecular(LayerStruct * );
void LaunchPhoton(double, LayerStruct *, PhotonStruct *);
void HopDropSpin(InputStruct  *,PhotonStruct *,OutStruct *);
void SumScaleResult(InputStruct, OutStruct *);
void WriteResult(InputStruct, OutStruct, char *);

InputStruct in_parm;
InputStruct in_parm_opti;

OutStruct out_parm;
OutStruct out_parm_opti;

/*The pseudorandom number generator*/
unsigned long Y1=3115,Y2=3115;
unsigned long  m1=1L<<31,m2=1L<<30;
unsigned long a1=65539,a2=410092949;
int b1=0,b2=1;
unsigned long  GambleMAX1=1L<<31-1,GambleMAX2=1L<<30-1;

unsigned long Gamblerand1()
{
    Y1=(Y1*a1+b1)%m1;
    return Y1;
}

unsigned long Gamblerand2()
{
    Y2=(Y2*a2+b2)%m2;
    return Y2;
}
```

```
/*end of the pseudorandom number generating*/

double DMAX(double a,double b)
{
    return a>b ? a:b;
}

//================mcml.c===================//
void DoOneRun(InputStruct *In_Ptr,OutStruct *out_parm)
{
    register long i_photon;
    /* index to photon. register for speed.*/
    /* distribution of photons.*/
    PhotonStruct photon;
    long num_photons = In_Ptr->num_photons, photon_rep=10;
    out_parm->Rsp = Rspecular(In_Ptr->layerspecs);
    i_photon = num_photons;
    do
    {
        LaunchPhoton(out_parm->Rsp, In_Ptr->layerspecs, &photon);
        do
            HopDropSpin(In_Ptr, &photon, out_parm);
        while (!photon.dead);
    }
    while(--i_photon);
}



/*the range of variable of the function*/
double xrange[S][2]={{0.01,3},{2,200},{0.0,1.2},{1.0,2.0}};
/************************************************************
    Report time and write results.
************************************************************/
void ReportResult(InputStruct In_Parm, OutStruct Out_Parm)
{
    char time_report[STRLEN];

    strcpy(time_report, " Simulation time of this run.");
    PunchTime(1, time_report);
    SumScaleResult(In_Parm, &Out_Parm);
    WriteResult(In_Parm, Out_Parm, time_report);
}

/************************************************************
    function definition
************************************************************/
double f(double x[S])
{
    int i;
    double f;
    f=0.;
    in_parm.layerspecs[1].mua=x[0];
    in_parm.layerspecs[1].mus=x[1];
```

```
    in_parm.layerspecs [1]. g=x[2];
    in_parm.layerspecs [1]. n=x[3];
    InitOutputData(in_parm, &out_parm_opti);

    DoOneRun(&in_parm,&out_parm_opti);
    ReportResult(in_parm, out_parm_opti);

    for(i=0;i<50;i++)/*nr==50*/
        f+=(out_parm.Tt_r[i]-out_parm_opti.Tt_r[i])*(out_parm.Tt_r[i]-out_parm_opti.
         Tt_r[i]);
    f=-f;/*to find the minimum of f equal to find the maximum of -f*/
    return f;
}

/************************************************************
    calculate fitness
************************************************************/
void fitness(double fv[N+1],double pf[N+1])
{
    double Cmin;
    double Sumf=0.;
    int i;
    Cmin=fv[1];
    Sumf+=fv[1];
    for( i=2;i<=N;i++)
    {
        Sumf+=fv[i];
        if((Cmin-fv[i])>1.0E-8)
            Cmin=fv[i];
    }
    Sumf=Sumf-Cmin*N;
    for( i=1;i<=N;i++)
        pf[i]=(fv[i]-Cmin)/Sumf;
}

/************************************************************
    select individual to do local search
************************************************************/
int individual_select (double pf[])
{
    int m;
    double gamblep;
    double gamblesum;
    gamblep=(double)(Gamblerand1())/GambleMAX1;
    gamblesum=0.0;
    for(m=1;m<=N;m++)
    {
        gamblesum=gamblesum+pf[m];
        if((gamblesum-gamblep)>1.0E-20)
            break;
    }
    if(m>N)
        m=N;
    return m;
```

```
}

/*************************************************************
   roulette , select individual to be replaced by new point
*************************************************************/
int individual_replace (double pf[])
{
    int m;
    double gamblep;
    double gamblesum;
    gamblep=(double)(Gamblerand2())/GambleMAX2;
    gamblesum=0.0;
    for(m=1;m<=N;m++)
    { /*the bigger fitness ,the less to be replaced*/
        gamblesum=gamblesum+(1-pf[m])/(N-1);
        if((gamblesum-gamblep)>1.0E-20)
            break;
    }
    if(m>N)
        m=N;
    return m;
}

/*************************************************************
    adaptive local search of AQMC method
*************************************************************/
void laqmc(double x[N+1][S],double c[S],double *fv_c, int Ni,double *search_step,
         double c3,int *vanish)
{
    double gcx[N+1][S];
    double flocalmax=*fv_c;
    int j,k;
    int indomain;
    double radius=*search_step;
    double tempx;
    for(j=0;j<S;j++)
        gcx[0][j]=c[j];

    for(j=1;j<=Ni;j++)
    {
        for(k=0;k<S;k++)
            gcx[j][k]=gcx[0][k]+radius*(2*x[j][k]-(xrange[k][0]+xrange[k][1]));
        indomain=1;
        for(k=0;k<S;k++)
        {
            if((xrange[k][0]-gcx[j][k])>1.0E-6||(gcx[j][k]-xrange[k][1])>1.0E-6)
                indomain=0;
        }
        if(indomain)
        {
            tempx=f(gcx[j]);
            if((tempx-flocalmax)>1.0E-8)
            {
                flocalmax=tempx;
```

```
                    for(k=0;k<S;k++)
                        gcx[0][k]=gcx[j][k];/*to store the tempory max value*/
                }
            } /*end of indomain*/
             else
            {
                (* vanish)++;
                printf("\t%dth point not in the domain\t",j);
                printf("(%f,%f,%f,%f)\n",gcx[j][0], gcx[j][1], gcx[j][2], gcx[j][3]) ;
            }
        } /*end of for j*/
         if((flocalmax-*fv_c)>1.0E-8)
        {
             radius=fabs(c[0]-gcx[0][0])/(xrange[0][1]-xrange[0][0]) ;
             c[0]=gcx[0][0];
             for(k=1;k<S;k++)
            {
                 tempx=fabs(c[k]-gcx[0][k])/(xrange[k][1]-xrange[k][0]) ;
                 if((tempx-radius)>1.0E-10)
                     radius=tempx;
                 c[k]=gcx[0][k];//to store the tempory max value
            }
            * search_step=radius;
            * fv_c=flocalmax;
        }
         else
            * search_step=(*search_step)*c3;
}

/*************************************************************
     report the result
*************************************************************/
void report(double fglobalmax[S+1],int count, int vanish)
{
     int j;
     FILE *result;
     if( (result = fopen( "mcmlmin", "a" )) == NULL )
     {
          printf( "The file 'mcmlmin' was not opened\n" );
          exit(0);
     }
     fprintf( result,"   mcmlmin=%.10f\n",-fglobalmax[S]);
     for(j=0;j<S;j++)
          fprintf(result,"x[%d]=%.10f\t",j,fglobalmax[j]);
     fprintf(result,"\n");
     fprintf(result,"count(the calculated function value number)=%d\n",count-vanish
          );
     fprintf(result,"
          ========================================\n");
     fclose(result);
}

/*************************************************************
     look for the global maximum
```

```
**********************************************************/
void fmax()
{
    FILE *result;
    int vanish=0;
    int count=0;/*to count the calculation function value number*/
    double x[N+1][S];/*the initial N Sobol' random number*/
    double fx[N+1][S];/*variable(x) of each individual*/
    double pf[N+1];/*fitness of each individual*/
    double fv[N+1];/*function value of each individual*/
    double ss[N+1];/*search radius of each individual*/
    double fglobalmax[S+1];/*to store the varible and max value of the function*/
    int  fglobalposition ;
    int  i ,j ,k,m;
    int Ni;/*the search number in local search*/
    double epslon=0.25;/*the initial value of search radius*/
    /*parameters of AQMC optimization method*/
    double c1=0.04;
    double c2=1.0;
    double c3=epslon*epslon;
    double c4=0.04;
    int newN=c4*N;
    double radius=1.0;
    double temp;
    double evolution;
    double lseff=1.0,  nieff =1.0;/*local search and new individual performance/cost
        ratio*/
    double ls_max_initial,ls_max_current, ni_max_initial ,ni_max_current;
    double ls_N=0,ni_N=0;/*the count of local search and new individual*/
    double newp=0.2;/*the probability to generate new individuals*/
    double rnd;/*random number*/
    int  ini ,run;

    static double xsob[S+1];
    srand((unsigned)time(NULL));/*give the rand seed*/

    printf( "The Monte Carlo random search method for the global optimum starts
        ...\n");
    if( ( result  = fopen( "mcmlmin", "w" )) == NULL )
    {
        printf( "The file 'mcmlmin' was not opened\n" );
        exit(0);
    }
    fprintf (  result ,"===The Monte Carlo random search method for the global
        optimum===\n\n");
    fprintf ( result ,"N(population size)=%d\n",N);
    fprintf ( result ,"epslon(the initial search radius)=%f\n",epslon);
    fprintf ( result ,"c1=%f\tc2=%f\tc3=%f\tc4=%f\n",c1,c2,c3,c4);
    fprintf ( result ,"\n");
    fprintf ( result ,"\n");
    fprintf ( result ,"==================\n");
    fclose ( result );

    i=1;
```

```
printf("Generation %d(initial population)...\n",i);
/*generate N quasirandom,dimension S*/
ini=-1;
sobseq(&ini,xsob);/*the Sobol' generator  initilization */
run=S;
count=count+N;
for( i=1;i<=N;i++)
{
    sobseq(&run,xsob);
    for(j=0;j<S;j++)
   {
        x[i][j]=xrange[j][0]+(xrange[j][1]-xrange[j][0]) *xsob[j+1];
        fx[i][j]=x[i][j];
   }
    fv[i]=f(fx[i]);
    printf("\t %dth point: %f(%f,%f,%f,%f)\n",i,-fv[i],x[i][0], x[i][1], x[i][2], x[i
     ][3]) ;
    ss[i]=epslon;
}
/*end of generating N quasirandom,dimension S*/

/*globalmax*/
fglobalposition =1;
for( i=2;i<=N;i++)
{
    if((fv[i]-fv[fglobalposition])>1.0E-8)
        fglobalposition =i;
}
for(i=0;i<S;i++)
    fglobalmax[i]=fx[fglobalposition][i];
fglobalmax[S]=fv[fglobalposition];
/*to calculate the fitness*/
fitness(fv,pf);
i=1;
report(fglobalmax,count,vanish);
ni_max_initial=fglobalmax[S];
ls_max_initial=fglobalmax[S];
ls_max_current=fglobalmax[S];
ni_max_current=fglobalmax[S];
i=2;
do
{
    printf("Generation %d...\n",i);
    printf("\t probability to generate new individuals %f\n",newp);
    /*========fitness=======*/
    rnd=(double)(rand())/RAND_MAX;
    if(rnd<newp)/*generate new individual*/
   { printf("\tgenerate %d new points...\n",newN);
        count=count+newN;
        ni_N+=newN;
        /*to find the invidicual to be replaced by new point*/
        for(j=1;j<newN;j++)
       {
            do
```

```
        {
              m=individual_replace(pf);
        }
          while(m==fglobalposition);/*keep the best individual*/
          /*to generate one new point*/
          sobseq(&run,xsob);
          for(k=0;k<S;k++)
               fx[m][k]=xrange[k][0]+(xrange[k][1]-xrange[k][0])*xsob[k+1];
          fv[m]=f(fx[m]);
          printf("\t %dth point: %f(%f,%f,%f,%f)\n",j,-fv[m],fx[m][0],fx[m][1],
  fx[m][2],fx[m][3]);
          ss[m]=epslon;
          if((fv[m]-fglobalmax[S])>1.0E-8)
        {
              fglobalposition =m;
              fglobalmax[S]=fv[m];
              for(k=0;k<S;k++)
                   fglobalmax[k]=fx[fglobalposition ][k];
              report(fglobalmax,count,vanish);
              ni_max_current=fv[m];
              printf("\tFind max when generate new individual!\n");
        }
    } /*end of generating  New points*/
      nieff=(ni_max_current-ni_max_initial)/ni_N;
      printf("\tnew individuals performance/cost ratio %f\n",nieff);
      fitness(fv,pf);
} /*end of if(newp.....)*/
 else /*local search*/
{ /*to select one point for local searching*/
      m=individual_select(pf);
      Ni=(int)(c2*N*DMAX(ss[m],c1));
      ls_N+=Ni;
      count=count+Ni;
      temp=fv[m];
      printf("\tlocal search, %d points...\n",Ni);
      /*local search*/
      laqmc(x,fx[m],&(fv[m]),Ni,&(ss[m]),c3,&vanish);
      if(fv[m]-temp>1.0E-5)/*if find new local max*/
    {  fitness(fv,pf);
    }
      /*global max*/
      if((fv[m]-fglobalmax[S])>1.0E-8)
    {
          for(j=0;j<S;j++)
               fglobalmax[j]=fx[m][j];
          fglobalmax[S]=fv[m];
          fglobalposition =m;
          report(fglobalmax,count,vanish);
          ls_max_current=fv[m];
          printf("\tFind max in local search!\n");
    }
      lseff=(ls_max_current-ls_max_initial)/ls_N;/*performance/cost ratio of
  local search*/
      printf("\tlocal performance/cost ratio %f\n",lseff);
```

```
        } /*end of local search*/
         /*calculate the probability of generating new points*/
         if( lseff <1.0E-10 && nieff< 1.0E-10)
             newp=0.2;
         else
             newp=0.2*nieff/(0.8*lseff+0.2*nieff);
         i++;
    } while(fglobalmax[S]<-1.0E-5);
}

time_t PunchTime(char F, char *Msg)
{
#if GNUCC
    return(0);
#else
static clock_t ut0; /* user time reference. */
    static time_t rt0; /* real time reference. */
    double secs;
    char s[STRLEN];

    if(F==0)
    {
        ut0 = clock();
        rt0 = time(NULL);
        return(0);
    }
    else if(F==1)
    {
        secs = (clock() - ut0)/(double)CLOCKS_PER_SEC;
        if (secs<0)
            secs=0; /* clock() can overflow. */
        sprintf(s, "User time: %8.0lf sec = %8.2lf hr. %s\n",
                secs, secs/3600.0, Msg);
        puts(s);
        strcpy(Msg, s);
        return(difftime(time(NULL), rt0));
    }
    else if(F==2)
        return(difftime(time(NULL), rt0));
    else
        return(0);
#endif
}

/*********************************************************
 *  Get the file name of the input data file from the
 *  argument to the command line.
 ****/
void GetFnameFromArgv(int argc, char * argv[], char * input_filename)
{
    if(argc>=2)
    { /* filename in command line */
        strcpy(input_filename, argv[1]);
    }
```

```
    else
        input_filename[0] = '\0';
}


void main(int argc, char *argv[])
{
    char input_filename[STRLEN];
    FILE *input_file_ptr;
    short num_runs;/* number of independent runs. */
    /*InputStruct in_parm;
    OutStruct out_parm;*/
    GetFnameFromArgv(argc, argv, input_filename);
     input_file_ptr  = GetFile(input_filename);
    CheckParm(input_file_ptr, &in_parm);
    num_runs=ReadNumRuns(input_file_ptr);
    ReadParm(input_file_ptr, &in_parm);
    ReadParm(input_file_ptr, &in_parm_opti);

    InitOutputData(in_parm, &out_parm);
    DoOneRun(&in_parm,&out_parm);
    ReportResult(in_parm, out_parm);

    fmax();
    FreeData(in_parm, &out_parm);
    FreeData(in_parm_opti, &out_parm_opti);
     fclose ( input_file_ptr );
}
```

Bibliography

[1] J. Dongarra and F. Sullivan. Guest editors' introduction: the top 10 algorithms. *Computing in Science and Engineering*, 2(1):22–23, January/February 2000.

[2] B. Cipra. The best of the 20th century: editors name top 10 algorithms. *SIAM News*, 33(4):1, 2000.

[3] N. Metropolis and S. Ulam. The monte carlo method. *J. Am. stat. Ass.*, 44:335–341, 1949.

[4] R. E. Caflisch. Monte carlo and quasi-monte carlo methods. *Acta Numerica*, 7:1–49, 1998.

[5] W. Feller. *An introduction to probability theory and its applications: Vol. I.* Wiley, New York, 1971.

[6] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods.* SIAM, Philadelphia, 1992.

[7] R. Y. Rubinstein. *Simulation and the Monte Carlo method.* Wiley, New York, 1981.

[8] H. Niederreiter. *Studies in pure Mathematics(To the memory of Paul Turán)*, pages 523–529. Birkhäuser Verlag, Basel, 1983.

[9] M. P. Allen and D. J. Tildesley. *Computer simulation of liquids.* Oxford University Press, New York, 1989.

[10] W. J. Morokoff and R. E. Caflisch. A quasi-monte carlo approach of particle simulation of the heat equation. *SIAM J. Numer. Anal.*, 30(6):1558–1573, 1993.

[11] C. Lécot and I. Coulibaly. A quasi-monte carlo scheme using nets for a linear boltzmann equation. *SIAM J. Numer. Anal.*, 35(1):51–70, 1998.

[12] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. chem. Phys.*, 21:1087–1092, 1953.

[13] B. H. Sendov and I. dimov, editors. *International youth workshop on Monte Carlo methods and parallel algorithms*. World Scientific Publishing Co. Ltd., Singapore, 1989.

[14] H. Niederreiter and P. J. Shiue, editors. *Monte Carlo and quasi-Monte Carlo methods in scientific computing*. Springer-Verlag, New York, 1995.

[15] H. Niederreiter, P. Hellekalek, G. Larcher, and P. Zinterhof, editors. *Monte Carlo and Quasi-Monte Carlo methods 1996*. Springer-Verlag, New York, 1998.

[16] J.E. Gentle. *Random number generation and Monte Carlo methods*. Springer-Verlag, New York, 1998.

[17] B. F. J. Manly. *Randomization, bootstrap and Monte Carlo methods in biology*. Chapman and Hall, London, 2nd edition, 1997.

[18] D. H. Lehmer. *Proc. 2nd sympos. on large-scale digital calculating machinery*, pages 141–146. Harvard University Press, Cambridge, 1951.

[19] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluation multi-dimensional integrals. *Numer. Math.*, 2:84–90, 1960.

[20] H. Faure. Discrépance de suites associées à un système de numération (en dimension s). *Acta Arith.*, 41:337–351, 1982.

[21] I.M. Sobol'. The distribution of points in a cube and the approximate evaluation of integrals. *USSR Comp. Math. and Math. Phys.*, 7:86–112, 1967.

[22] H. Niederreiter. Point sets and sequences with small discrepancy. *Monatsh. Math.*, 104:273–337, 1987.

[23] H. Niederreiter. Quasi-monte carlo methods and pseudo-random numbers. *Bull. Amer. Math. Soc.*, 84(6):957–1041, 1978.

[24] I. A. Antonov and V. M. Saleev. An economic method of computing lp_r-sequences. *USSR Comput. Math. Math. Phys.*, 19:252–256, 1979.

[25] P. Bratley and B. L. Fox. Algorithms 659 implementing sobol's quasirandom sequence generator. *ACM Trans. Math. Software*, 14(1):88–100, 1988.

[26] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical recipes in C: The art of scientific computing*. Cambridge University Press, New York, 1992.

[27] G. Y. Lei. B-splines smoothed rejection sampling method and its applications in quasi-monte carlo integration. *Journal of Zhejiang University, Science*, 3:339–343, 2002.

[28] K. T. Fang and Y. Wang. *Number-theoretic methods in statistics*. Chapman & Hall, London, 1994.

[29] X. Q. Wang. Improving the rejection sampling method in quasi-monte carlo methods. *Journal of Computational and Applied Mathematics*, 114:231–246, 2000.

[30] B. Moskowitz and R.E. Caflisch. Smoothness and dimension reduction in quasi-monte carlo methods. *Math. Comput. Modelling*, 23(8-9):37–54, 1996.

[31] L. Schumaker. *Spline functions: basic theory*. Wiley, New York, 1981.

[32] P. Bratley, B. L. Fox, and H. Niederreiter. Implementation and tests of low-discrepancy sequences. *ACM Transactions on Modeling and Computer Simulation*, 2(3):195–213, 1992.

[33] J. M. Hammersley and D. C. Handscomb. *Monte Carlo methods*. Methuen & Co Ltd, Lodon, 1964.

[34] J. M. Hammersley and K. W. Morton. A new monte carlo technique: antithetic variates. *Proc. Camb. Phil. Soc.*, 52:449–475, 1956.

[35] S. Haber. A modified monte carlo quadrature. ii. *Math. Comp.*, 21:388–397, 1967.

[36] A. Karaivanova and I. Dimov. Error analysis of an adaptive monte carlo method for numerical integration. *Mathematics and Computers in Simulation*, 47:201–213, 1998.

[37] I. M. Sobol' and A. V. Tutunnikov. A variance reducing multiplier for monte carlo integration. *Math. Comput. Modelling*, 23(8-9):87–96, 1996.

[38] G. Y. Lei. Adaptive quasi-monte carlo method for multiple-extrema optimization. *Control Theory and Applications*, 19:431–434, 2002.

[39] G.Y. Lei. Adaptive random search in quasi-monte carlo methods for global optimization. *Computers and Mathematics with Applications*, 43:747–754, 2002.

[40] H. Niederreiter and P. Peart. Localization of search in quasi-monte carlo methods for global optimization. *SIAM J. Sci. Stat. Comput.*, 7(2):660–664, 1986.

[41] Y. Wang and K. T. Fang. Number theoretic methods in applied statistics. *Chinese Ann. Math. Ser.*, B,11:41–55, 859–914, 1990.

[42] H. Niederreiter. *Topics in classical number theory*, pages 1163–1208. North-Holland, Amsterdam, 1984.

[43] J. H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[44] K. A. DeJong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[45] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, Reading, Mass., 1989.

[46] Z. Michalewicz. *Genetic algorithms + data Structures = evolution programs.* Springer-Verlag, Berlin, 1994.

[47] J. R. Koza. *Genetic programming.* MIT Press, Cambridge, Mass., 1992.

[48] B. Souček. *Dynamic, genetic, and chaotic programming.* John Wiley & Sons, 1992.

[49] Y. K. Wan. *C++ language and the object-oriented programming.* The press of Tsinghua University, 1998.

[50] S. A. Prahl, M. Keijzer, S. L. Jacques, and A. J. Welch. A monte carlo model of light propagation in tissue. *SPIE Institute Series*, 5:102–111, 1989.

[51] L. G. Henyey and J. L. Greenstein. Diffuse radiation in the galaxy. *Astrophys. J.*, 93:70–83, 1941.

[52] L. H. Wang, S. L. Jacques, and L. Q. Zheng. Mcml - monte carlo modeling of photon transport in multi-layered tissues. *Computer Methods and Programs in Biomedicine*, 47:131–146, 1995.

[53] J. H. Halton and G. B. Smith. Algorith 247: Radical-inverse quasi-random point sequence. *Comm. ACM*, 7:701–702, 1964.

Index