A GUARANTEED, ADAPTIVE, AUTOMATIC ALGORITHM FOR UNIVARIATE FUNCTION MINIMIZATION

BY

XIN TONG

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Applied Mathematics
in the Graduate College of the
Illinois Institute of Technology

Approved ____________________
Advisor

Chicago, Illinois
July 2014

ACKNOWLEDGMENT

This thesis could not have been completed without Professor Fred J. Hickernell who not only served as my advisor but also encouraged and challenged me throughout my academic program. He guided me through the research process, never accepting anything less than my best efforts. Another faculty member, Professor Sou-Cheng Choi and graduate students, Ms. Yuhan Ding, Mr. Yizhi Zhang, and Ms. Lan Jiang, also helped me a lot with my research and thesis. I thank them all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis proposes a guaranteed, adaptive, automatic algorithm for solving univariate function minimization problem on the unit interval. The key to this adaptive algorithm is performing the analysis for cones of input functions that are twice differentiable. This cone is defined in terms of two semi-norms, a stronger one and a weaker one. Three fixed-cost algorithms based on linear splines are used to find the bounds for an input function and its minimum value. The estimated minimum value and possible optimal solution set are given by those bounds. This algorithm is guaranteed to provide either a minimum value within a user-specified tolerance or a possible optimal solution set whose volume is less than another user-specified tolerance.

CHAPTER 1
INTRODUCTION

There are several numerical algorithms for solving the univariate function minimization problem. For example, MATLAB's **fminbnd** attempts to find a minimum of a function of one variable within a fixed interval using the golden section search method [1, 5]. Unfortunately, **fminbnd** might give only a local minimum. Moreover, **fminbnd** is not a guaranteed algorithm. In this article, we construct a guaranteed, adaptive, automatic algorithm for solving univariate function minimization problem based on the numerical analysis on cone of input functions. An algorithm is automatic when its computational effort depends on the tolerances and the input function, but no additional user input is required. An algorithm is adaptive when its computational cost depends on some semi-norm of the input function, not input by the user, but reliably inferred from function data.

1.1 Problem Definition

The problem to be solved is univariate function minimization on the unit interval:

$$S(f) := \min_{0 \le x \le 1} f(x). \tag{1.1}$$

The space of input functions is the Sobolev space $\mathcal{W}^{2,\infty} = \{f \in C[0,1] : ||f''||_\infty < \infty\}$. This requires the input functions to be continuous and have finite second derivatives on $[0,1]$.

1.2 Basic Concepts and Fixed-cost Algorithms

The details of numerical analysis on cone of input functions are presented in [4]. In particular, our algorithm follows the analysis on L_∞ approximation of univariate functions [4, Section 6]. In this article, we use same definition for a cone of input functions as well as the three fixed-cost algorithms. Given a fixed sample size n, a fixed-cost algorithm uses only n function values and incurs computational costs of order n in terms of time and memory. In addition, some useful propositions from L_∞ approximation of univariate functions in [4] are listed in next section.

The first fixed-cost algorithm is based on linear splines on $[0, 1]$. Using n function values for $n \in \{2, 3, \dots\}$, we define the linear spline algorithm as follows:

$$A_n(f)(x) := (n-1)[f(x_i)(x_{i+1} - x) + f(x_{i+1})(x - x_i)], \quad x_i \le x \le x_{i+1}$$
$$x_i = \frac{i-1}{n-1}, \quad i = 1, \dots, n. \tag{1.2}$$

This linear spline function is preferred to approximate the input function because the error can be expressed in terms of its second derivative times a quadratic polynomial.

The stronger semi-norm of the input function space is $||f''||_\infty$; correspondingly the weaker semi-norm is defined as

$$||f' - A_2(f)'||_\infty = ||f' - f(1) + f(0)||_\infty.$$

Then the adaptive algorithm to approximate $S(f)$ is defined for a *cone* of input functions:

$$\mathcal{C}_\tau := \{f \in \mathcal{W}^{2,\infty} : ||f''||_\infty \le \tau ||f' - f(1) + f(0)||_\infty\}, \tag{1.3}$$

where the τ is the cone constant. The reason for defining the cone with two semi-norms is explained in analysis in [4]. The semi-norm $||f''||_\infty$ is used to bound the difference between $A_n(f)(x)$ and $f(x)$. Also, $||f' - f(1) + f(0)||_\infty$ vanishes if f is a linear function.

One can approximate the $\mathcal{L}_\infty$ norm of $f' - f(1) + f(0)$ using the same data in (1.2) by another fixed-cost algorithm

$$\begin{aligned}\widetilde{F}_n(f) :&= ||A_n(f)' - A_2(f)'||_\infty \\ &= \sup_{i=1,\dots,n-1} |(n-1)[f(x_{i+1}) - f(x_i)] - f(1) + f(0)|.\end{aligned} \tag{1.4}$$

Moreover, a lower bound on $||f''||_\infty$ can be derived using a central difference method. For $n > 3$, this lower bound is given in [4, equation(31)] by

$$F_n(f) := (n-1)^2 \sup_{i=1,\dots,n-2} |f(x_i) - 2f(x_{i+1}) + f(x_{i+2})| \le ||f''||_\infty. \tag{1.5}$$

1.3 Useful Propositions

The following approximation theory results from [4] are used in our algorithm to be developed in Chapter 2.

1. The difference between f and its linear spline algorithm $A_n(f)$ can be bounded in terms of an integral involving its second derivative using integration by parts [4, equation(32)]. For $x \in [x_i, x_{i+1}]$, it follows that:

$$\begin{aligned} f(x) - A_n(f)(x) &= f(x) - (n-1)[f(x_i)(x_{i+1} - x) + f(x_{i+1})(x - x_i)] \\ &= (n-1)\int_{x_i}^{x_{i+1}} v_i(t,x) f''(t)\,dt, \end{aligned} \tag{1.6}$$

   where the continuous, piecewise differentiable kernel v_i is defined as

$$v_i(t,x) := \begin{cases} (x_{i+1} - x)(x_i - t), & x_i \le t \le x, \\ (x - x_i)(t - x_{i+1}), & x < t \le x_{i+1}. \end{cases}$$

2. The quantity $||f' - f(1) + f(0)||_\infty - \widetilde{F}_n(f)$ has the following bounds [4, p 39]:

$$0 \le ||f' - f(1) + f(0)||_\infty - \widetilde{F}_n(f) \le \frac{1}{2(n-1)} ||f''||_\infty.$$

3. An upper bound on the stronger norm, $||f''||_\infty$, is

$$||f''||_\infty \le \tau \mathfrak{C}_n \widetilde{F}_n(f), \tag{1.7}$$

   where $\mathfrak{C}_n = \frac{2(n-1)}{2(n-1)-\tau}$, provided $f \in \mathcal{C}_\tau$.

4. A necessary condition for f to lie in the cone $\mathcal{C}_\tau$ is

$$f \in \mathcal{C}_\tau \Longrightarrow F_n(f) \le ||f''||_\infty \le \tau \mathfrak{C}_n \widetilde{F}_n(f) \tag{1.8}$$

$$\Longrightarrow \tau_{\min,n} := \frac{2(n-1)F_n(f)}{2(n-1)\widetilde{F}_n(f) + F_n(f)} \le \tau. \tag{1.9}$$

Chapter 2 constructs our algorithm step by step. It starts from deriving and computing several bounds and then moves to estimate minimum values and a possible optimal solution set. Then the statement of the algorithm with a theorem and a short proof follow. Chapter 3 illustrates the algorithm using bump test functions. In addition, a family of functions with two local minimum points is used to compare the results with **fminbnd**. Last, a summary and a discussion about future work is presented in Chapter 4.

CHAPTER 2

ALGORITHM

2.1 Bounds on f and $\min_{0\le x\le 1} f(x)$

First, we derive a lower bound on the input function $f(x)$. In (1.6), the difference between $f(x)$ and its linear spline algorithm $A_n(f)(x)$ on $[x_i, x_{i+1}]$ is written as an integral of a kernel function $v_i(t,x)$ times the second derivative of input function $f''(x)$. Applying the Hölder's inequality (see, for example, [7, Theorem 12.54]) and (1.7), for $x \in [x_i, x_{i+1}]$, the error bound becomes:

$$\begin{aligned}
|f(x) - A_n(f)(x)| &= (n-1)\left|\int_{x_i}^{x_{i+1}} v_i(t,x) f''(t)dt\right| \\
&\le (n-1)||f''||_\infty \int_{x_i}^{x_{i+1}} |v_i(t,x)|dt \\
&= ||f''||_\infty \frac{(x-x_i)(x_{i+1}-x)}{2} \\
&\le \tau \mathfrak{C}_n \widetilde{F}_n(f) \frac{(x-x_i)(x_{i+1}-x)}{2}.
\end{aligned}$$

Unlike the L_∞ approximation of univariate functions taking $||f(x) - A_n(f)(x)||_\infty$ on $x \in [x_i, x_{i+1}]$, this error bound depends on x. This allows us to define the lower bound $f^i_{nL}(x)$, for $f(x)$ for $x \in [x_i, x_{i+1}]$, i.e.,

$$f^i_{nL}(x) := A_n(f)(x) - \tau \mathfrak{C}_n \widetilde{F}_n(f) \frac{(x-x_i)(x_{i+1}-x)}{2} \le f(x), \quad \forall x \in [x_i, x_{i+1}]. \quad (2.1)$$

Note that $A_n(f)$ is a piecewise linear function and $\tau \mathfrak{C}_n \widetilde{F}_n(f)$ is treated as a constant for a given sample size n. Then the lower bound on $f(x)$ is a piecewise quadratic function on $[x_i, x_{i+1}]$. By completing the square and rescaling the ranges, the lower bound function f_{nL}^i on $[x_i, x_{i+1}]$ yields

$$\begin{aligned}
f_{nL}^i(x) &= (n-1)[f(x_i)(x_{i+1}-x) + f(x_{i+1})(x-x_i)] - \tau \mathfrak{C}_n \widetilde{F}_n(f) \frac{(x-x_i)(x_{i+1}-x)}{2} \\
&= \frac{\tau \mathfrak{C}_n \widetilde{F}_n(f)}{2} \left(x - \frac{x_i + x_{i+1}}{2}\right)^2 + (n-1)[f(x_{i+1}) - f(x_i)] \left(x - \frac{x_i + x_{i+1}}{2}\right) \\
&\qquad\qquad + \frac{1}{2}[f(x_{i+1}) + f(x_i)] - \frac{\tau \mathfrak{C}_n \widetilde{F}_n(f)}{8(n-1)^2} \\
&= \frac{\tau \mathfrak{C}_n \widetilde{F}_n(f)}{8(n-1)^2} \left[2(n-1)\left(x - \frac{x_i + x_{i+1}}{2}\right) + \frac{2(n-1)^2[f(x_{i+1}) - f(x_i)]}{\tau \mathfrak{C}_n \widetilde{F}_n(f)}\right]^2 \\
&\quad - \frac{1}{4}[f(x_{i+1}) - f(x_i)] \frac{2(n-1)^2[f(x_{i+1}) - f(x_i)]}{\tau \mathfrak{C}_n \widetilde{F}_n(f)} \\
&\quad - \frac{1}{4}[f(x_{i+1}) - f(x_i)] \frac{\tau \mathfrak{C}_n \widetilde{F}_n(f)}{2(n-1)^2[f(x_{i+1}) - f(x_i)]} + \frac{1}{2}[f(x_{i+1}) + f(x_i)]
\end{aligned}$$

Defining

$$t := 2(n-1)\left(x - \frac{x_i + x_{i+1}}{2}\right) \in [-1, 1] \quad \text{and} \quad C_n^i := \frac{2(n-1)^2[f(x_{i+1}) - f(x_i)]}{\tau \mathfrak{C}_n \widetilde{F}_n(f)},$$

then we can transform the lower bound $f_{nL}^i(x)$ into

$$\begin{aligned}
g_{nL}^i(t) = &\frac{f(x_{i+1}) - f(x_i)}{4C_n^i}(t + C_n^i)^2 - \frac{f(x_{i+1}) - f(x_i)}{4}\left(C_n^i + \frac{1}{C_n^i}\right) \\
&+ \frac{1}{2}[f(x_{i+1}) + f(x_i)].
\end{aligned}$$

Next, we derive bounds on $\min\limits_{0 \le x \le 1} f(x)$. Now, the input function f is bounded from below by a quadratic function f_{nL}^i on $[x_i, x_{i+1}]$, which opens upward since $\frac{f(x_{i+1}) - f(x_i)}{4C_n^i} = \frac{\tau \mathfrak{C}_n \widetilde{F}_n(f)}{8(n-1)^2} \ge 0$. The corresponding C_n^i implies the location of vertex of f_{nL}^i. If $|C_n^i| < 1$, g_{nL}^i attains its minimum at $t = -C_n^i$, which means that f_{nL}^i attains its minimum at an interior point of $[x_i, x_{i+1}]$. Otherwise, g_{nL}^i attains its minimum at $t = -1$ or $t = 1$, i.e., f_{nL}^i attains its minimum at $x = x_i$ or $x = x_{i+1}$.

It follows that the minimum value of f^i_{nL} on $[x_i, x_{i+1}]$ is

$$l^i_n := \begin{cases} \frac{1}{2}[f(x_{i+1}) + f(x_i)] - \frac{1}{4}[f(x_{i+1}) - f(x_i)]\left(C^i_n + \frac{1}{C^i_n}\right), & -1 \le C^i_n \le 1, \\ f(x_{i+1}), & C^i_n < -1, \\ f(x_i), & C^i_n > 1, \end{cases}$$
$$= \frac{1}{2}[f(x_{i+1}) + f(x_i)] - \frac{1}{4}|f(x_{i+1}) - f(x_i)|\left(\min(|C^i_n|, 1) + \frac{1}{\min(|C^i_n|, 1)}\right). \quad (2.2)$$

The lower bound on $\min\limits_{0 \le x \le 1} f(x)$ is chosen as the smallest value among the minimum values of f^i_{nL} on each interval $[x_i, x_{i+1}]$, i.e.

$$L_n := \min_{1 \le i \le n-1} l^i_n. \quad (2.3)$$

On the other hand, since $\min\limits_{1 \le i \le n} f(x_i)$ is always greater than or equal to $\min\limits_{0 \le x \le 1} f(x)$, the upper bound on $\min\limits_{0 \le x \le 1} f(x)$ is

$$U_n := \min_{0 \le x \le 1} A_n(f)(x) = \min_{1 \le i \le n} f(x_i) \ge \min_{0 \le x \le 1} f(x). \quad (2.4)$$

Finally, we have the bounds for $\min\limits_{0 \le x \le 1} f(x)$:

$$L_n \le \min_{0 \le x \le 1} f(x) \le U_n. \quad (2.5)$$

2.2 Intervals Containing Optimal Solutions

In this section, we are going to find the intervals containing the possible optimal solutions using the bounds derived above. First of all, we define $\mathcal{X}$ as the optimal (minimum) solution set for input function f:

$$\mathcal{X} := \{t \in [0, 1] : f(t) \le f(x), \quad \forall x \in [0, 1]\}.$$

We will start looking at the subintervals $[x_i, x_{i+1}]$, $i = 1, 2, \ldots, n-1$ and then combine these results for interval $[0, 1]$. Given any interval $[x_i, x_{i+1}]$, for $x \in [x_i, x_{i+1}]$, (2.1) and (2.3) imply

$$L_n \le l_n^i \le f_{nL}^i(x) \le f(x).$$

If $l_n^i > U_n$, then

$$\min_{x_i \le x \le x_{i+1}} f(x) \ge l_n^i > U_n \ge \min_{0 \le x \le 1} f(x).$$

So the interval $[x_i, x_{i+1}]$ can be excluded from the optimal solution set. On the other hand, if $l_n^i \le U_n$, there exist some x in $[x_i, x_{i+1}]$ such that

$$l_n^i \le f_{nL}^i(x) \le U_n.$$

Such x may be an optimal solution. We shall find these intervals which satisfy $l_n^i \le U_n$ and next find the possible solutions in these intervals.

Define the index set

$$\mathcal{J}_n := \{i : l_n^i \le U_n\}.$$

Next we collect all the possible points on the interval $[x_i, x_{i+1}]$, $i \in \mathcal{J}_n$ such that

$$\{x : f_{nL}^i(x) \le U_n\}.$$

Then the possible solution set on $[0, 1]$ is given by

$$\mathcal{Z}_n := \cup_{i \in \mathcal{J}_n} \{x : f_{nL}^i(x) \le U_n\}.$$

This possible solution set is guaranteed to contain the optimal solution set, i.e.,

$$\mathcal{Z}_n \supseteq \mathcal{X}. \tag{2.6}$$

2.3 Stopping Conditions

Our adaptive, automatic algorithm is guaranteed to provide either a minimum value agreed to within a user-specified tolerance ε or a possible optimal solution set whose volume is less than another user-specified tolerance δ. For convenience, we call ε the error tolerance and δ the X tolerance. Guarantee of the error tolerance requires that the difference between the estimated minimum value and true minimum value is less than ε. Guarantee of the X tolerance requires that the volume of possible optimal solution set is less than δ.

The estimated minimum value is chosen as $\min\limits_{0\le x\le 1} A_n(f)(x)$ since the linear spline algorithm $A_n(f)$ approximates the input function f, and it is based on the sampled function values. The error for $\min\limits_{0\le x\le 1} A_n(f)(x)$ becomes

$$\operatorname{err}_n(f) := \min_{0\le x\le 1} A_n(f)(x) - \min_{0\le x\le 1} f(x) = U_n - \min_{0\le x\le 1} f(x).$$

Using the bounds for $\min\limits_{0\le x\le 1} f(x)$ in (2.5), the error is bounded by

$$\operatorname{err}_n(f) = U_n - \min_{0\le x\le 1} f(x) \le U_n - L_n. \tag{2.7}$$

If the error bound $U_n - L_n$ is less than error tolerance, the difference between estimated value $\min\limits_{0\le x\le 1} A_n(f)(x)$ and $\min\limits_{0\le x\le 1} f(x)$ must be less than the error tolerance. We will take $U_n - L_n$ as the estimated error.

The possible optimal solution set $\mathcal{Z}_n$ is a collection of intervals such that $\{x : f^j_{nL}(x) \le U_n\}$, $j \in \mathcal{J}_n$. We will simplify (and enlarge) this possible optimal solution set $\mathcal{Z}_n$ by combining the intervals whose indices are consecutive. Define the index set of left endpoints for all the new intervals as

$$\underline{\mathcal{J}}_n := \{i : l^i_n \le U_n \wedge (l^{i-1}_n > U_n \vee i = 1)\},$$

and define the index set of right endpoints for all the new intervals as

$$\overline{\mathcal{J}}_n := \{i : l^i_n \le U_n \wedge (l^{i+1}_n > U_n \vee i = n-1)\}.$$

Let $\underline{i}$ be the jth element of the index set $\underline{\mathcal{J}}_n$ and $\bar{\imath}$ be the jth element of $\overline{\mathcal{J}}_n$. The left endpoint of the jth new interval is

$$\underline{\xi}_{n,j} := \min\{x \ge x_{n,\underline{i}} : f^{\underline{i}}_{nL}(x) \le U_n\},$$

and the corresponding right endpoint of the jth new interval is

$$\overline{\xi}_{n,j} := \max\{x \le x_{n,\bar{\imath}+1} : f^{\bar{\imath}}_{nL}(x) \le U_n\}.$$

Finally the possible optimal solution set turns out to be

$$\mathcal{X}_n := \cup_j [\underline{\xi}_{n,j}, \overline{\xi}_{n,j}], \tag{2.8}$$

which is automatically guaranteed to contain the optimal solution set by (2.6), i.e.,

$$\mathcal{X}_n \supseteq \mathcal{Z}_n \supseteq \mathcal{X}. \tag{2.9}$$

To compute its volume is to compute the length of each interval and sum them up, i.e.,

$$\mathrm{Vol}(\mathcal{X}_n) = \sum_j (\overline{\xi}_{n,j} - \underline{\xi}_{n,j}).$$

Finally, the stopping conditions should be chosen as

$$U_n - L_n \le \varepsilon$$

or

$$\mathrm{Vol}(\mathcal{X}_n) \le \delta.$$

The practical stopping conditions also involve a cost budget $N_{\max}$. This cost budget is the maximum function values used to estimate the minimum value and possible solutions. We will explain how this cost budget works in our algorithm in Chapter 3.

2.4 Algorithm: Adaptive Univariate Function Minimization

We now use the arguments above to construct our guaranteed algorithm for univariate function minimization. After present this algorithm we prove that it works.

Algorithm (funmin_g). Let the sequences of algorithms $\{A_n\}_{n\in\mathcal{I}}$, $\{\widetilde{F}_n\}_{n\in\mathcal{I}}$, $\{F_n\}_{n\in\mathcal{I}}$ be as described above. Let $\tau \ge 2$ be the cone constant. Set $k = 1$. Let $n_1 =$

$\lceil(\tau+1)/2\rceil+1$. For any error tolerance ε, X tolerance δ and input function f, do the following:

Stage 1. Estimate $||f'-f(1)+f(0)||_\infty$ and bound $||f''||_\infty$. Compute $\widetilde{F}_{n_k}(f)$ and $F_{n_k}(f)$.

Stage 2. Check the necessary condition for $f\in\mathcal{C}_\tau$. Compute

$$\tau_{\min,n_k}=\frac{F_{n_k}(f)}{\widetilde{F}_{n_k}(f)+F_{n_k}(f)/(2n_k-2)}.$$

If $\tau\ge\tau_{\min,n_k}$, then go to stage 3. Otherwise, set $\tau=2\tau_{min,n_k}$. If $n_k\ge(\tau+1)/2$, go to stage 3. Otherwise, choose

$$n_{k+1}=1+(n_k-1)\left\lceil\frac{\tau+1}{2n_k-2}\right\rceil.$$

Go to stage 3.

Stage 3. Check for convergence. Compute the bounds L_{n_k} and U_{n_k}:

$$\begin{aligned}U_{n_k}&=\min_{1\le i\le n_k}f(x_i),\\ L_{n_k}&=\min_{1\le i\le n_k-1}l^i_{n_k}\\ &=\min_{1\le i\le n_k-1}\frac{1}{2}[f(x_{i+1})+f(x_i)]\\ &\quad-\frac{1}{4}|f(x_{i+1})-f(x_i)|\left(\min(|C^i_{n_k}|,1)+\frac{1}{\min(|C^i_{n_k}|,1)}\right),\end{aligned}$$

where $C^i_{n_k}=\frac{2(n_k-1)^2[f(x_{i+1})-f(x_i)]}{\tau\mathfrak{C}_{n_k}\widetilde{F}_{n_k}}$. Compute the volume of possible solution set:

$$\begin{aligned}\mathrm{Vol}(\mathcal{X}_{n_k})&=\cup_j[\underline{\xi}_{n_k,j},\overline{\xi}_{n_k,j}]\\ &=\sum_j(\overline{\xi}_{n_k,j}-\underline{\xi}_{n_k,j}),\end{aligned}$$

where

$$\begin{aligned}\underline{\xi}_{n_k,j}&=\min\{x\ge x_{n_k,\underline{\imath}}:f^{\underline{\imath}}_{n_kL}(x)\le U_{n_k}\},\\ \overline{\xi}_{n_k,j}&=\max\{x\le x_{n_k,\overline{\imath}+1}:f^{\overline{\imath}}_{n_kL}(x)\le U_{n_k}\}.\end{aligned}$$

Here the $\underline{i}$ is the jth element of the index set $\underline{\mathcal{J}}_{n_k}$ given by

$$\underline{\mathcal{J}}_{n_k} = \{i : l_{n_k}^i \leq U_{n_k} \wedge (l_{n_k}^{i-1} > U_{n_k} \vee i = 1)\},$$

and the $\bar{i}$ is the jth element of the index set $\overline{\mathcal{J}}_{n_k}$ given by

$$\overline{\mathcal{J}}_{n_k} = \{i : l_{n_k}^i \leq U_{n_k} \wedge (l_{n_k}^{i+1} > U_{n_k} \vee i = n_k - 1)\}.$$

Check whether n_k is large enough to satisfy either the error tolerance or X tolerance, i.e.,

$$\text{a)} \quad U_{n_k} - L_{n_k} \leq \varepsilon$$

$$\text{or}$$

$$\text{b)} \quad \text{Vol}(\mathcal{X}_{n_k}) \leq \delta.$$

If it is true, return an approximation of $\min\limits_{1\leq i\leq n_k} f(x_i)$ and $\mathcal{X}_{n_k}$ and terminate the algorithm. If it is not true, choose:

$$n_{k+1} = 1 + 2(n_k - 1).$$

Go to stage 1.

Theorem. *For any n,*

$$L_n \leq \min_{0\leq x\leq 1} f(x) \leq U_n, \tag{2.10}$$

$$\mathcal{X}_n \supseteq \mathcal{X}. \tag{2.11}$$

When the algorithm terminates with given error tolerance ε and X tolerance δ at sample size n_k, either $\text{err}_{n_k}(f) = U_{n_k} - \min\limits_{0\leq x\leq 1} f(x) \leq \varepsilon$ or $\text{Vol}(\mathcal{X}) \leq \delta$ is satisfied.

Proof. We shall prove these statements by the previous deduction.

1. For any sample size n, the lower and upper bounds on $\min_{0 \le x \le 1} f(x)$ are given by L_n (2.3) and U_n (2.4). Then we have (2.10).

2. For any sample size n, the possible optimal solution set is given by $\mathcal{X}_n$ (2.8). Then we have (2.11).

3. When the algorithm terminates at a simple size n_k, either $U_{n_k} - L_{n_k} \le \varepsilon$ or $\mathrm{Vol}(\mathcal{X}_{n_k}) \le \delta$ is satisfied. The inequality (2.5) implies

$$\mathrm{err}_{n_k}(f) = U_{n_k} - \min_{0 \le x \le 1} f(x) \le U_{n_k} - L_{n_k} \le \varepsilon.$$

   Relation (2.9) implies

$$\mathrm{Vol}(\mathcal{X}) \le \mathrm{Vol}(\mathcal{X}_{n_k}) \le \delta.$$

$\square$

We do not yet have an upper bound on the cost of **funmin_g**. Therefore, we cannot yet prove that the algorithm terminates. This is an area for further research.

CHAPTER 3
NUMERICAL EXAMPLES

In this chapter, MATLAB is used to develop the Algorithm **funmin_g**. This algorithm has similar structure to **funappx_g** for adaptive univariate function recovery [4, Algorithm 5], which is included in GAIL [3]. The toolbox is created following the practices of reliable reproducible computational research in [2]. It also includes a Monte Carlo algorithm given in [6]. The algorithm **funmin_g** is only supported on Matlab 7.x and above. The following experiments in this chapter are run with MATLAB 8.1.0.604 (R2013a) and Microsoft Windows 7.

3.1 A Family of Bump Test Functions

The first example is a family of bump test functions defined by

$$f(x) = \begin{cases} \frac{1}{2a^2}[-4a^2-(x-z)^2-(x-z-a)|x-z-a| & \\ +(x-z+a)|x-z+a|], & z-2a \le x \le z+2a, \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

with $\log_{10}(a) \sim \mathcal{U}[-4,-1]$ and $z \sim \mathcal{U}[2a, 1-2a]$. Here $\mathcal{U}[p,q]$ stands for continuous uniform distribution on $[p,q]$. For instance, let $a = 0.03$ and $z = 0.4$. Then the function f is depicted by Figure 3.1. The minimum value of this function family is -1 at $x = z$. It follows that $||f' - f(1) + f(0)||_\infty = 1/a$ and $||f''||_\infty = 1/a^2$. Then we

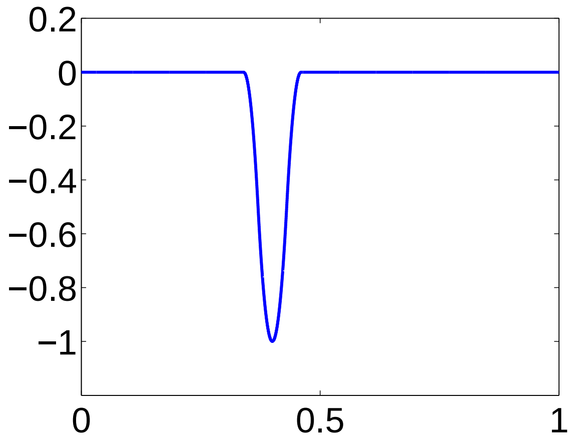

Figure 3.1. An example of the bump test functions.

can compute the probability that $f \in \mathcal{C}_\tau$ using the definition of $\mathcal{C}_\tau$:

$$\begin{aligned}
\mathbb{P}(f \in \mathcal{C}_\tau) &= \mathbb{P}(||f''||_\infty \le \tau ||f' - f(1) + f(0)||_\infty) \\
&= \mathbb{P}(1/a^2 \le \tau/a) \\
&= \mathbb{P}(a \ge 1/\tau) \\
&= \begin{cases} 0, & \tau \le 10, \\ (\log_{10}(\tau) - 1)/3, & 10 < \tau \le 10000, \\ 1, & \tau > 10000, \end{cases} \\
&= \max(0, \min(1, (\log_{10}(\tau) - 1)/3)).
\end{aligned}$$

3.1.1 Experiment with $\varepsilon = 10^{-8}$ and $\delta = 0$. As the first experiment, we chose 10000 random test functions and applied **funmin_g** with initial τ values of $11, 101, 1001$, an error tolerance of $\varepsilon = 10^{-8}$, and an X tolerance of $\delta = 0$. If the X tolerance $\delta = 0$, the algorithm is terminated by satisfying the error tolerance. The algorithm is considered successful for a particular f if the exact and approximate minimum value agree to within ε. The success and failure rates are given in Table 3.1. Our algorithm imposes a cost budget of $N_{\max} = 10^7$. If the proposed n_{k+1} in

Stages 2 or 3 exceeds $N_{\max}$, our algorithm returns a warning and falls back to the previous n_k. The probability that f initially lies in $\mathcal{C}_\tau$ is the smaller number in the second column of Table 3.1, while the larger number is the empirical probability that f eventually lies in $\mathcal{C}_\tau$ after possible increases in τ made by Stage 2 of the algorithm.

For this experiment, our algorithm was successful without warning for almost all f that finally lies inside $\mathcal{C}_\tau$. It diverged for a small percentage of functions lying inside the cone since the sample size was constrained by the cost budget to satisfy the error tolerance.

Table 3.1. Empirical success rates with $\varepsilon = 10^{-8}$ and $\delta = 0$.

τ	$\mathbb{P}(f \in \mathcal{C}_\tau)$	Success No Warning	Success Warning	Failure No Warning	Failure Warning
11	1.28% $\rightarrow$ 21.22%	21.22%	0.00%	78.78%	0.00%
101	34.02% $\rightarrow$ 53.20%	53.18%	0.02%	46.79%	0.01%
1001	67.04% $\rightarrow$ 85.79%	81.20%	2.59%	14.19%	2.02%

3.1.2 Experiment with $\varepsilon = 0$ and $\delta = 10^{-6}$. The second experiment uses the same family of test functions with an error tolerance of $\varepsilon = 0$ and an X tolerance of $\delta = 10^{-6}$. 10000 random test functions, initial τ values of $11, 101, 1001$ and cost budget 10^7 are chosen the same as the first experiment. If the error tolerance $\varepsilon = 0$, the algorithm is terminated by satisfying the X tolerance. The algorithm is considered successful for a particular f if the exact optimal solution is contained in the possible solution set whose volume agrees to within δ. The success and failure rates are given in Table 3.2. For this experiment, our algorithm was successful for all f that finally lie inside $\mathcal{C}_\tau$, for which there was no warning. It was also successful for a small percentage of functions lying outside the cone.

3.1.3 Experiment with $\varepsilon = 10^{-8}$ and $\delta = 10^{-6}$. The family of bump test functions (3.1) is again used in the third experiment. We considered an error tolerance

Table 3.2. Empirical success rates with $\varepsilon = 0$ and $\delta = 10^{-6}$.

		Success	Success	Failure	Failure
τ	$\mathbb{P}(f \in \mathcal{C}_\tau)$	No Warning	Warning	No Warning	Warning
11	1.52% $\to$ 20.92%	20.92%	0.00%	79.08%	0.00%
101	32.80% $\to$ 52.19%	52.19%	0.00%	47.81%	0.00%
1001	66.93% $\to$ 84.92%	80.31%	4.64%	15.05%	0.00%

of $\varepsilon = 10^{-8}$ and an X tolerance of $\delta = 10^{-6}$. Therefore, the algorithm is considered successful for a particular f if the exact and approximate minimum value agree to within ε or the exact optimal solution is contained in the solution set whose volume agrees to within δ. The number of random functions chosen, the initial τ values and the cost budget are the same as former experiments. Table 3.3 shows results that are analogous to Table 3.2. The algorithm yields the correct value to within the error tolerances for all f that finally lie inside $\mathcal{C}_\tau$ and for which the algorithm does not try to exceed the cost budget.

Table 3.3. Empirical success rates with $\varepsilon = 10^{-8}$ and $\delta = 10^{-6}$.

		Success	Success	Failure	Failure
τ	$\mathbb{P}(f \in \mathcal{C}_\tau)$	No Warning	Warning	No Warning	Warning
11	1.54% $\to$ 21.00%	21.00%	0.00%	79.00%	0.00%
101	33.42% $\to$ 52.28%	52.28%	0.00%	47.72%	0.00%
1001	66.15% $\to$ 85.32%	85.33%	0.00%	14.67%	0.00%

3.1.4 Why funmin_g is fooled by some bump test functions.

The sequences of algorithms $\{A_n\}_{n\in\mathcal{I}}$, $\{\widetilde{F}_n\}_{n\in\mathcal{I}}$, and $\{F_n\}_{n\in\mathcal{I}}$ are used in **funmin_g** to find a minimum sample size to satisfy the stopping conditions. The results, including minimum value of the function and possible optimal solutions, are based on the sample size and function values sampled. When the input function is relatively peaky, which means the $||f''||_\infty$ is much greater than $||f' - f(1) + f(0)||_\infty$, the input function will fail to satisfy the cone condition. Although f may not lie in cone $\mathfrak{C}_\tau$, it may be the case that necessary condition (1.9) is satisfied so **funmin_g** does not increase τ as one might happen in Stage 2. Then **funmin_g** proceeds to Stage 3 and terminates; the n function values are not sufficient to determine the correct answer within the given tolerances because f is not in $\mathfrak{C}_\tau$.

3.2 Functions with Two Local Minimum Points

This example shows that our algorithm can find the global optimal solution. It is much more effective than **fminbnd**. Let us consider the family of test functions defined by

$$f(x) = -5\exp(-[10(x-a_1)]^2) - \exp(-[10(x-a_2)]^2), \qquad 0 \le x \le 1 \qquad (3.2)$$

with $a_1 \sim \mathcal{U}[0, 0.5]$ and $a_2 \sim \mathcal{U}[0.5, 1]$. Here, f has two local minimum points. For instance, let $a_1 = 0.3$ and $a_2 = 0.75$. Then the function f is

$$f(x) = -5\exp(-[10(x-0.3)]^2) - \exp(-[10(x-0.75)]^2), \qquad 0 \le x \le 1$$

and depicted by Figure 3.2.

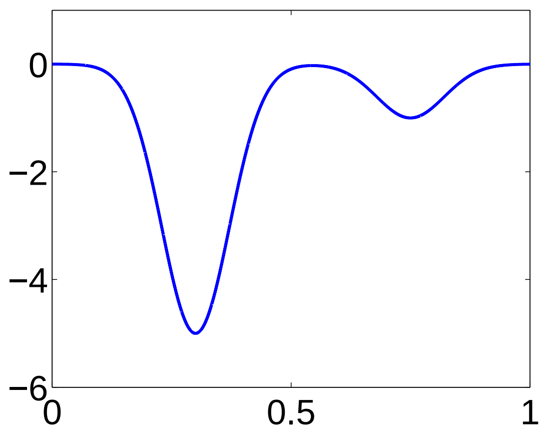

Figure 3.2. A function with two local minimum points.

As an experiment, we chose 10000 random test functions and applied **funmin_g** with an error tolerance of $\varepsilon = 0$ and X tolerance δ of 10^{-2}, 10^{-4}, and 10^{-7}.

The cost budget in this experiment is $N_{\max} = 10^7$. The algorithm **funmin_g** is considered successful for a particular f if the exact global optimal solution is contained in the possible solution set whose volume agrees to within δ. The other algorithm **fminbnd** is considered successful when its optimal solution and the exact solution agree to within δ. The success rates are given in Table 3.4. For this experiment, our algorithm **funmin_g** performed much better than **fminbnd** for solving the global minimization problem (3.2) because **fminbnd** based on golden section search method might give a local minimum, but not a global minimum.

Table 3.4. Empirical success rates of **funmin_g** compared to **fminbnd**.

	funmin_g			**fminbnd**
δ	Success	Success No Warning	Success Warning	Success
10^{-2}	100.00%	100.00%	0.00%	67.28%
10^{-4}	100.00%	100.00%	0.00%	67.28%
10^{-7}	100.00%	0.00%	100.00%	67.28%

CHAPTER 4
CONCLUSIONS

4.1 Summary

Our algorithm is designed to solve the global minimization problem for univariate functions. It focuses on both the minimum value and optimal solutions based on the sampled function values. This algorithm starts by defining a cone of input functions. Then it uses three sequences of fixed-cost algorithms to approximate the input function, lower bound function on the input function and bounds on the minimum value of the input function. Next, the minimum value and possible optimal solution set are given in terms of those bounds. Finally, constructing the guaranteed, adaptive, automatic algorithm follows [4, Algorithm 3].

We shall emphasize some breakthroughs. First, our algorithm provides either minimum values agreed to within error tolerance ε or possible optimal solution agreed to within X tolerance δ. It can help one find the estimated minimum value that differs from the true value by no more than ε, or possible optimal solution set whose volume is no more than δ, or both. Second, our algorithm returns the global solutions to the minimization problem. Even if a function attains its minimum at one boundary of the domain, our algorithm can find the solution closed to that boundary.

Some restrictions must be also mentioned. First of all, the input function f should be twice differentiable on $[0, 1]$. This condition is needed to define the cone. Next, our algorithm can be fooled by a spiky function, i.e., one that yields zero data where probed by the algorithm, but is nonzero elsewhere. Last, this algorithm may need a huge number of data when solving problems with flat functions, for example,

$$f(x) = \exp(-1/(x - 0.5)^2),$$

which is depicted by Figure 4.1.

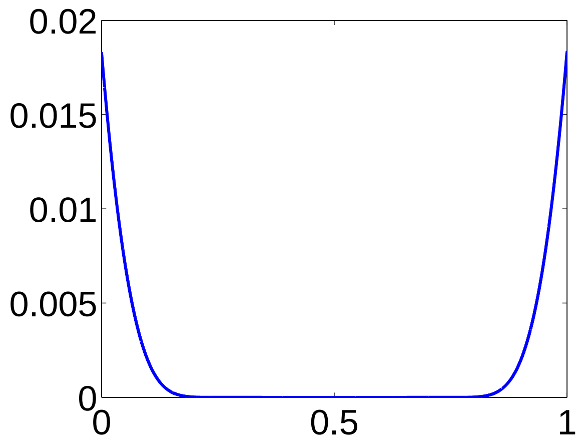

Figure 4.1. Example of a flat function.

4.2 Further Work

4.2.1 Interval Extension. In this thesis, our algorithm focuses on the minimization problem on the unit interval $[0, 1]$. Extending the domain to any interval $[a, b]$ is an important improvement for this algorithm. The following questions should be considered carefully:

1. How to sample the function values within a bounded interval $[a, b]$?

2. Does this algorithm work on an unbounded domain?

We are considering two ideas to improve our algorithm on a bounded interval $[a, b]$. One idea to apply our algorithm is to extend the unit interval directly. This idea try to sample the function values evenly as it does for the unit interval, which is equivalent to rescaling the input function f, i.e.,

$$\min_{x\in[a,b]} f(x) \qquad \Longleftrightarrow \qquad \min_{t\in[0,1]} f(a+(b-a)t).$$

Another idea is to split the interval into unit intervals. Next, apply our algorithm in each interval to find a local minimization solution. Then it is easy to figure out the global minimum among those local solutions. Unfortunately, for the unbounded intervals, such as $[a, +\infty)$, $(-\infty, b]$, and $(-\infty, +\infty)$, our algorithm does not work.

4.2.2 Computational Cost.

Both lower and upper bounds for the cost of [4, Algorithm 5 (Adaptive Univariate Function Recovery)] are given in [4, Theorem 9] as follows:

$$\begin{aligned}
\max\left(\left\lceil\frac{\tau+1}{2}\right\rceil, \left\lceil\sqrt{\frac{\|f''\|_\infty}{8\varepsilon}}\right\rceil\right)+1 \\
&\le \max\left(\left\lceil\frac{\tau+1}{2}\right\rceil, \left\lceil\sqrt{\frac{\tau\|f'-f(1)+f(0)\|_\infty}{8\varepsilon}}\right\rceil\right)+1 \\
&\le \operatorname{cost}(A, f; \varepsilon, N_{\max}) \\
&\le \sqrt{\frac{\tau\|f'-f(1)+f(0)\|_\infty}{2\varepsilon}}+\tau+4 \le \sqrt{\frac{\|f''\|_\infty}{4\varepsilon}}+\tau+4.
\end{aligned}$$

Since our algorithm is related to **funappx_g**, these bounds may be relevant to our algorithm. However, cost of our adaptive algorithm not only depends on the input function f and error tolerance ε, but also the X tolerance δ. The cost of our algorithm may be less than **funappx_g**.

One idea to find the bounds on $\operatorname{cost}(A, f; \varepsilon, \delta)$ is to analyze the stopping conditions separately. Suppose $\delta = 0$. Then the algorithm approximates the minimum value within ϵ. The analysis and result are analogous to **funappx_g**. Next, let $\varepsilon = 0$ and find the cost bounds. Then the upper bounds for $\operatorname{cost}(A, f; \varepsilon, \delta)$ should be the smaller one of the upper bounds between the case with $\delta = 0$ and the case with $\varepsilon = 0$. Similarly, the lower bound of $\operatorname{cost}(A, f; \varepsilon, \delta)$ is chosen as the bigger lower bound between the two cases.

APPENDIX A

MATLAB CODE FOR THE ALGORITHM: FUNMIN_G

```
function [fmin,out_param]=funmin_g(varargin)
%FUNMIN_G Guaranteed global minimum value of univariate function
%  on interval [0,1] and the subset containing optimal solutions
%
%  fmin = FUNMIN_G(f) finds minimum value of function f on the interval
%  [0,1] within a guatanteed absolute error tolerance of 1e-6 and X
%  tolerance of 1e-3. The default initial number of points is 52 and
%  default cost budget is 1e7. Input f is a function handle.
%
%  fmin = FUNMIN_G(f,abstol,TolX,ninit,nmax) finds minimum value of
%  function f on the interval [0,1] with ordered input parameters:
%  guaranteed absolute error tolerance abstol, guaranteed absolute X
%  tolerance TolX, initial number of points ninit and cost budget nmax.
%
%  fmin = FUNMIN_G(f,'abstol',abstol,'TolX',TolX,'ninit',ninit,'nmax',
%  nmax) finds minimum value of function f on the interval [0,1] with
%  a guaranteed absolute error tolerance abstol, guaranteed absolute
%  X tolerance TolX, initial number of points ninit and cost budget
%  nmax. All the three field-value pairs are optional and can be
%  supplied in different order.
%
%  fmin = FUNMIN_G(f,in_param) finds minimum value of function f on the
%  interval [0,1] with a structure input parameters in_param. If a
%  field is not specified, the default value is used.
%
%    in_param.abstol --- guaranteed absolute error tolerance, default
%                        value is 1e-6.
%
%    in_param.TolX --- guaranteed X tolerance, default value is 1e-3.
%
%    in_param.ninit --- initial number of points, default value is 52.
%
%    in_param.nmax --- cost budget, default value is 1e7.
%
%  [fmin, out_param] = FUNMIN_G(f,...) returns minimum value fmin of
%  function f and an output structure out_param, which has the following
%  fields.
%
%    out_param.abstol --- guaranteed absolute error tolerance
%
%    out_param.ninit --- initial number of points
%
%    out_param.nmax --- cost budget
%
%    out_param.TolX --- guaranteed X tolerance
%
%    out_param.tau --- latest value of tau
%
%    out_param.exceedbudget --- 0 if the number of points used to find
%    the minimux value is less than the cost budget; 1, otherwise.
%
%    out_param.npoints --- number of points needed to reach the
%                          guaranteed absolute error tolerance
%                          or the guaranteed X tolerance
```

```
%
%   out_param.error --- estimation of the absolute error bound
%
%   out_param.intervals --- the intervals containing point(s) where the
%   minimum occurs
%
%   out_param.volumeX --- the volume of intervals containing the
%                         point(s) where the minimum occurs
%
%   out_param.tauchange --- it is 1 if tau is too small, and the
%                           algorithm has used a larger tau.
%
%
%  Examples
%
%  Example 1:
%
%  >> f=@(x) (x-0.3).^2+1; [fmin,out_param] = funmin_g(f)
%
%  fmin =
%    1.0000e+00
%  out_param =
%           abstol: 1.0000e-06
%             TolX: 1.0000e-03
%            ninit: 52
%             nmax: 10000000
%              tau: 101
%     exceedbudget: 0
%          npoints: 6529
%            error: 2.9618e-07
%          volumeX: 9.8451e-04
%        tauchange: 0
%        intervals: [2x1 double]
%
%
%  Example 2:
%
%  >> f=@(x) (x-0.3).^2+1; in_param.abstol = 1e-8;
%  >> in_param.ninit = 10; in_param.nmax = 1e6;
%  >> in_param.TolX = 1e-4;
%  >> [fmin,out_param] = funmin_g(f,in_param)
%
%  fmin =
%    1.0000e+00
%  out_param =
%           abstol: 1.0000e-08
%                f: @(x)(x-0.3).^2+1
%            ninit: 10
%             nmax: 1000000
%             TolX: 1.0000e-04
%              tau: 17
%     exceedbudget: 0
%          npoints: 18433
%            error: 5.9665e-09
%          volumeX: 1.3544e-04
%        tauchange: 0
%        intervals: [2x1 double]
%
%
%  Example 3:
%
%   >> f=@(x) (x-0.3).^2+1;
%   >> [fmin,out_param] = ...
   funmin_g(f,'ninit',10,'nmax',1e6,'abstol',1e-4,
%   'TolX',1e-2)
%
%  fmin =
%    1.0000e+00
%  out_param =
%           abstol: 1.0000e-04
%                f: @(x)(x-0.3).^2+1
%            ninit: 10
%             nmax: 1000000
%             TolX: 1.0000e-02
%              tau: 17
%     exceedbudget: 0
%          npoints: 145
%            error: 9.4167e-05
%          volumeX: 1.6961e-02
%        tauchange: 0
%        intervals: [2x1 double]
%
%
%  Sea also FUNAPPX_G
%
%  Reference
%  [1]  Nicholas Clancy, Yuhan Ding, Caleb Hamilton, Fred J. Hickernell,
%       and Yizhi Zhang. The Cost of Deterministic, Adaptive, Automatic
%       Algorithms: Cones, Not Balls. Journal of Complexity, 30:21-45,
%       2014
%
%  [2]  Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang,
%       and Yizhi Zhang, "GAIL: Guaranteed Automatic Integration Library
%       (Version 1.3.0)" [MATLAB Software], 2014. Available from
%       http://code.google.com/p/gail/


% Parse and check the validity of input parameters
[f,out_param] = funmin_g_param(varargin{:});


%% Main algorithm

% initialize number of points
n = out_param.ninit;
% initialize tau
out_param.tau = ceil((n-1)*2-1);
% cost budget flag
```

```
out_param.exceedbudget = 1;
% tau change flag
tauchange = 0;

while n < out_param.nmax;
    % Stage 1: estimate weaker and stronger norm
    x = (0:n−1)/(n−1);
    y = f(x);
    diff_y = diff(y);
    %approximate the weaker norm of input function
    gn = (n−1)*max(abs(diff_y−(y(n)−y(1))/(n−1)));
    %approximate the stronger norm of input function
    fn = (n−1)^2*max(abs(diff(diff_y)));

    % Stage 2: satisfy necessary condition of cone
    if out_param.tau*(gn+fn/(2*n−2))>= fn;
        % Stage 3: check for convergence
        bn = 2*(n−1)*out_param.tau/(2*(n−1)−out_param.tau)*gn;
        cn = 2*(n−1)^2*abs(diff_y)./bn;
        Cn = min(cn,1); % check the conditions for each interval
        ln = (diff_y/2+y(1:n−1))−abs(diff_y).*(Cn+1./Cn)/4;
        % minimum values of each interval
        Ln = min(ln); % lower bound
        min_endpoint = min(y); % upper bound
        error = min_endpoint−Ln;
        % find the intervals containing minimum points
        index = find(cn<1 & ln < min_endpoint);
        m = size(index,2);
        if m > 0
            delta = (n−1)^2*diff_y(index).^2−2*bn*(diff_y(index)./2 ...
                +y(index)−bn/8/(n−1)^2−min_endpoint);
            ints = zeros(2,m);
            ints(1,:)=x(index)+1/2/(n−1)−(n−1)*diff_y(index)./bn ...
                −sqrt(delta)./bn;
            ints(2,:)=x(index)+1/2/(n−1)−(n−1)*diff_y(index)./bn ...
                +sqrt(delta)./bn;
            leftint = find([1 diff(index)~=1]);
            rightint = find([diff(index)~=1 1]);
            q = size(leftint,2);
            interval = zeros(2,q);
            interval(1,:) = ints(1,leftint);
            interval(2,:) = ints(2,rightint);
        else
            interval = zeros(2,0);
        end
        volumeX = sum(interval(2,:)−interval(1,:));
        % satisfy convergence
        if error < out_param.abstol || volumeX < out_param.TolX
            out_param.exceedbudget = 0; break;
        end
        % otherwise increase points number
        l = n;
        n = 2*(n−1)+1;
        % Stage 2: do not satisfy necessary condition
    else
        % increase tau
        out_param.tau = 2*fn/(gn+fn/(2*n−2));
        % change tau change flag
        tauchange = 1;
        % check if number of points large enough
        if n >= ((out_param.tau+1)/2);
            % large enough, go to Stage 3
            bn = 2*(n−1)*out_param.tau/(2*(n−1)−out_param.tau)*gn;
            cn = 2*(n−1)^2*abs(diff_y)./bn;
            Cn = min(cn,1); % check the conditions for each interval
            ln = (diff_y/2+y(1:n−1))  −abs(diff_y).*(Cn+1./Cn)/4;
            % minimum values of each interval
            Ln = min(ln); % lower bound
            min_endpoint = min(y); % upper bound
            error = min_endpoint−Ln;
            % find the intervals containing minimum points
            index = find(cn<1 & ln < min_endpoint);
            m = size(index,2);
            if m > 0
                delta = ...
                    (n−1)^2*diff_y(index).^2−2*bn*(diff_y(index)./2 ...
                    +y(index)−bn/8/(n−1)^2−min_endpoint);
                ints = zeros(2,m);
                ints(1,:)=x(index)+1/2/(n−1)−(n−1)*diff_y(index)./bn ...
                    −sqrt(delta)./bn;
                ints(2,:)=x(index)+1/2/(n−1)−(n−1)*diff_y(index)./bn ...
                    +sqrt(delta)./bn;
                leftint = find([1 diff(index)~=1]);
                rightint = find([diff(index)~=1 1]);
                q = size(leftint,2);
                interval = zeros(2,q);
                interval(1,:) = ints(1,leftint);
                interval(2,:) = ints(2,rightint);
            else
                interval = zeros(2,0);
            end
            volumeX = sum(interval(2,:)−interval(1,:));
            % satisfy convergence
            if error < out_param.abstol || volumeX < out_param.TolX
                out_param.exceedbudget = 0; break;
            end
            % otherwise increase points number
            l = n;
            n = 2*(n−1)+1;

        else
            % not large enough, increase points number, and go to ...
               Stage 1
            l = n;
            n = 1 + (n−1)*ceil(out_param.tau+1/(2*n−2));
        end;
    end;
```

```
end;

% check tau change flag
if tauchange == 1
    warning('MATLAB:funmin_g:peaky','This function is peaky relative ...
        to ninit. You may wish to increase ninit for similar ...
        functions.')
end;

% check cost budget flag
if out_param.exceedbudget == 1
    n = 1;
    warning('MATLAB:funmin_g:exceedbudget','funmin_g attempted to ...
        exceed the cost budget. The answer may be unreliable.')
end

fmin = min_endpoint;
out_param.npoints = n;
out_param.error = error;
out_param.volumeX = volumeX;
out_param.tauchange = tauchange;
out_param.intervals = interval;


function [f, out_param] = funmin_g_param(varargin)
% Parse the input to the funmin_g.m function
default.abstol = 1e-6;
default.TolX = 1e-3;
default.ninit = 52;
default.nmax = 1e7;

if isempty(varargin)
    help funmin_g
    warning('Function f must be specified. Now funmin_g will use ...
        f(x)=(x-0.3)^2+1.')
    f = @(x) (x-0.3).^2+1;
else
    f = varargin{1};
end

validvarargin=numel(varargin)>1;
if validvarargin
    in2=varargin{2};
    validvarargin=(isnumeric(in2) || isstruct(in2) || ischar(in2));
end

if ~validvarargin
% There is only one input f or the second input is not satisfied our ...
   type.
% The default parameters are used.
    out_param.abstol = default.abstol;
    out_param.TolX = default.TolX;
    out_param.ninit = default.ninit;
    out_param.nmax = default.nmax;
```

```
else
    p = inputParser;
    addRequired(p,'f',@isfcn);
    if isnumeric(in2) % more inputs of numerical type. Put them in ...
       order.
        addOptional(p,'abstol',default.abstol,@isnumeric);
        addOptional(p,'TolX',default.TolX,@isnumeric);
        addOptional(p,'ninit',default.ninit,@isnumeric);
        addOptional(p,'nmax',default.nmax,@isnumeric);
    else
        if isstruct(in2) % second input is a structure
            p.StructExpand = true;
            p.KeepUnmatched = true;
        end
        addParamValue(p,'abstol',default.abstol,@isnumeric);
        addParamValue(p,'TolX',default.TolX,@isnumeric);
        addParamValue(p,'ninit',default.ninit,@isnumeric);
        addParamValue(p,'nmax',default.nmax,@isnumeric);
    end
    parse(p,f,varargin{2:end});
    out_param = p.Results;
end

% Check whether the error tolerance is positive
if out_param.abstol < 0
    warning(['Error tolerance should be greater than or equal to 0.'...
        ' funmin_g will use the default error tolerance' ...
        num2str(default.abstol)]);
    out_param.abstol = default.abstol;
end

% Check whether the length tolerance is positive
if out_param.TolX < 0
    warning(['Tolerance on X should be greater than or equal to 0.'...
    ' funmin_g will use the default X tolerance ' ...
       num2str(default.TolX)]);
    out_param.abstol = default.TolX;
end

% Check whether the initial number of points is a positive integer
if (~isposint(out_param.ninit))
    if ispositive(out_param.ninit)
        warning(['Initial number of points should be a integer.' ...
            ' funmin_g will use ' num2str(ceil(out_param.ninit))]);
        out_param.ninit = ceil(out_param.ninit);
    else
        warning(['Initial number of points should be a positive ...
           integer.' ...
            ' funmin_g will use the default initial number of points ...
               ' ...
            num2str(out_param.ninit)]);
        out_param.ninit = default.ninit;
    end
end
```

```
% Check whether the cost budget is a positive integer
if (~isposint(out_param.nmax))
    if ispositive(out_param.nmax)
        warning(['Cost budget should be a integer.'' funmin_g will ...
           use ' ...
            num2str(ceil(out_param.nmax))]);
        out_param.nmax = ceil(out_param.nmax);
    else
        warning(['Cost budget should be a positive integer.'...
        ' funmin_g will use the default budget' ...
           num2str(out_param.nmax)]);
        out_param.nmax = default.nmax;
    end
end
end
end


function b=isfcn(h)
%ISFCN To judge if input is a function handle or not
b = isa(h, 'function_handle');


function b=isposint(a)
% ISPOSINT To judge if input is a positive integer or not
b = (ceil(a)==a) && (a>0);


function b=ispositive(a)
% ISPOSITIVE To judge if a variable is positive or not
b = isnumeric(a) && (a>0);
```

APPENDIX B
MATLAB CODE FOR THE EXPERIMENTS

```
%% Experiment 1: Bump test functions with epsilon=10^(-8) & delta=0

%% Garbage collection and initialization
format compact %remove blank lines from output
format long e %lots of digits
clear all %clear all variables
close all %close all figures
tstart = tic;

%% Program parameters
in_param.abstol = 10^(-8); %error tolerance
in_param.TolX = 0;
in_param.nmax = 10^7; %cost budget

%% Simulation parameters
nrep = 10000;
if (nrep >= 1000)
    warning('off','MATLAB:funmin_g:exceedbudget');
    warning('off','MATLAB:funmin_g:peaky');
end;
a = 10.^(-4+3*rand(nrep,1));
z = 2.*a+(1-4*a).*rand(nrep,1);
x0 = z-2*a;
x1 = z+2*a;
tauvec = [11 101 1001]; %cone condition tau
ntau = length(tauvec);
ratio = 1./a;
gnorm = 1./a;
exactmin = -1;

%% Simulation
ntrapmat = zeros(nrep,ntau);
trueerrormat = ntrapmat;
truesolumat = ntrapmat;
newtaumat = ntrapmat;
tauchangemat = ntrapmat;
exceedmat = ntrapmat;

for i=1:ntau;
    for j=1:nrep;
        f = @(x) 0.5/a(j)^2*(-4*a(j)^2-(x-z(j)).^2-(x-z(j)-a(j)).*...
            abs(x-z(j)-a(j))+(x-z(j)+a(j)).*abs(x-z(j)+a(j))).*...
            (x>=z(j)-2*a(j)).*(x<=z(j)+2*a(j)); %test function
        in_param.ninit = (tauvec(i)+1)/2+1;
        [fmin,out_param] = funmin_g(f,in_param);
        ntrapmat(j,i) = out_param.npoints;
        newtaumat(j,i) = out_param.tau;
        estmin = fmin;
        trueerrormat(j,i) = abs(estmin-exactmin);
        tauchangemat(j,i) = out_param.tauchange;
        exceedmat(j,i) = out_param.exceedbudget;
    end
end
```

```
probinit = mean(repmat(ratio,1,ntau)<=repmat(tauvec,nrep,1),1);
probfinl = mean(repmat(ratio,1,ntau)<=newtaumat,1);
succnowarn = mean((trueerrormat<=in_param.abstol)&(~exceedmat),1);
succwarn = mean((trueerrormat<=in_param.abstol)&(exceedmat),1);
failnowarn = mean((trueerrormat>in_param.abstol)&(~exceedmat),1);
failwarn = mean((trueerrormat>in_param.abstol)&(exceedmat),1);

%% Output the table
display(' ')
display('        Probability    Success   Success   Failure  Failure')
display(' tau       In Cone    No Warning  Warning No Warning Warning')
for i=1:ntau
    display(sprintf(['%5.0f %5.2f%%->%5.2f%% %7.2f%%' ...
        '%10.2f%% %7.2f%% %7.2f%% '],...
        [tauvec(i) 100*[probinit(i) probfinl(i) succnowarn(i) ...
        succwarn(i) failnowarn(i) failwarn(i)]]))
end
toc(tstart)


%% Experiment 2: Bump test functions with epsilon=0 & delta=10^(-6)

%% Garbage collection and initialization
format compact %remove blank lines from output
format long e %lots of digits
clear all %clear all variables
close all %close all figures
tstart = tic;

%% Program parameters
in_param.abstol = 0; %error tolerance
in_param.TolX = 10^(-6);
in_param.nmax = 10^7; %cost budget

%% Simulation parameters
nrep = 10000;
if (nrep >= 1000)
    warning('off','MATLAB:funmin_g:exceedbudget');
    warning('off','MATLAB:funmin_g:peaky');
end;
a = 10.^(-4+3*rand(nrep,1));
z = 2.*a+(1-4*a).*rand(nrep,1);
x0 = z-2*a;
x1 = z+2*a;
tauvec = [11 101 1001]; %cone condition tau
ntau = length(tauvec);
ratio = 1./a;
gnorm = 1./a;
exactmin = -1;
exactsolu = z;

%% Simulation
ntrapmat = zeros(nrep,ntau);
truesolumat = ntrapmat;
```

```
newtaumat = ntrapmat;
tauchangemat = ntrapmat;
exceedmat = ntrapmat;
intnum = ntrapmat;

for i=1:ntau;
    for j=1:nrep;
        f = @(x) 0.5/a(j)^2*(-4*a(j)^2-(x-z(j)).^2-(x-z(j)-a(j)).*...
            abs(x-z(j)-a(j))+(x-z(j)+a(j)).*abs(x-z(j)+a(j))).*...
            (x>=z(j)-2*a(j)).*(x<=z(j)+2*a(j)); %test function
        in_param.ninit = (tauvec(i)+1)/2+1;
        [fmin,out_param] = funmin_g(f,in_param);
        ntrapmat(j,i) = out_param.npoints;
        newtaumat(j,i) = out_param.tau;
        estmin = fmin;
        tauchangemat(j,i) = out_param.tauchange;
        exceedmat(j,i) = out_param.exceedbudget;
        intnum(j,i) = size(out_param.intervals,2);
        for k=1:intnum(j,i)
            if exactsolu(j) <= out_param.intervals(2,k) && ...
                exactsolu(j) >= out_param.intervals(1,k)
                 truesolumat(j,i) = 1;
            end
        end
    end
end

probinit = mean(repmat(ratio,1,ntau)<=repmat(tauvec,nrep,1),1);
probfinl = mean(repmat(ratio,1,ntau)<=newtaumat,1);
succnowarn = mean((truesolumat)&(~exceedmat),1);
succwarn = mean((truesolumat)&(exceedmat),1);
failnowarn = mean((~truesolumat)&(~exceedmat),1);
failwarn = mean((~truesolumat)&(exceedmat),1);

%% Output the table
display(' ')
display('        Probability    Success   Success   Failure  Failure')
display(' tau      In Cone    No Warning  Warning No Warning Warning')
for i=1:ntau
    display(sprintf(['%5.0f %5.2f%%->%5.2f%% %7.2f%%' ...
        '%10.2f%% %7.2f%% %7.2f%%'],...
        [tauvec(i) 100*[probinit(i) probfinl(i) succnowarn(i) ...
        succwarn(i) failnowarn(i) failwarn(i) ]]))
end
toc(tstart)


%% Experiment 3: Bump test functions with epsilon=10^(-8) & ...
   delta=10^(-6)

%% Garbage collection and initialization
format compact %remove blank lines from output
format long e %lots of digits
clear all %clear all variables
```

```
close all %close all figures
tstart = tic;

%% Program parameters
in_param.abstol = 10^(-8); %error tolerance
in_param.TolX = 10^(-6);
in_param.nmax = 10^7; %cost budget
tic

%% Simulation parameters
nrep = 10000;
if (nrep >= 1000)
    warning('off','MATLAB:funmin_g:exceedbudget');
    warning('off','MATLAB:funmin_g:peaky');
end;
a = 10.^(-4+3*rand(nrep,1));
z = 2.*a+(1-4*a).*rand(nrep,1);
x0 = z-2*a;
x1 = z+2*a;
tauvec = [11 101 1001]; %cone condition tau
ntau = length(tauvec);
ratio = 1./a;
gnorm = 1./a;
exactmin = -1;
exactsolu = z;

%% Simulation
ntrapmat = zeros(nrep,ntau);
trueerrormat = ntrapmat;
truesolumat = ntrapmat;
newtaumat = ntrapmat;
tauchangemat = ntrapmat;
exceedmat = ntrapmat;
intnum = ntrapmat;

for i=1:ntau;
    for j=1:nrep;
        f = @(x) 0.5/a(j)^2*(-4*a(j)^2-(x-z(j)).^2-(x-z(j)-a(j)).*...
            abs(x-z(j)-a(j))+(x-z(j)+a(j)).*abs(x-z(j)+a(j))).*...
            (x>=z(j)-2*a(j)).*(x<=z(j)+2*a(j)); %test function
        in_param.ninit = (tauvec(i)+1)/2+1;
        [fmin,out_param] = funmin_g(f,in_param);
        ntrapmat(j,i) = out_param.npoints;
        newtaumat(j,i) = out_param.tau;
        estmin = fmin;
        trueerrormat(j,i) = abs(estmin-exactmin);
        tauchangemat(j,i) = out_param.tauchange;
        exceedmat(j,i) = out_param.exceedbudget;
        intnum(j,i) = size(out_param.intervals,2);
        for k=1:intnum(j,i)
            if exactsolu(j) <= out_param.intervals(2,k) && ...
                exactsolu(j) >= out_param.intervals(1,k)
                 truesolumat(j,i) = 1;
            end
        end
    end
end

probinit = mean(repmat(ratio,1,ntau)<=repmat(tauvec,nrep,1),1);
probfinl = mean(repmat(ratio,1,ntau)<=newtaumat,1);
succnowarn=
    mean((trueerrormat<=in_param.abstol|truesolumat)&(~exceedmat),1);
succwarn=
    mean((trueerrormat<=in_param.abstol|truesolumat)&(exceedmat),1);
failnowarn=
    mean((trueerrormat>in_param.abstol&~truesolumat)&(~exceedmat),1);
failwarn=
    mean((trueerrormat>in_param.abstol&~truesolumat)&(exceedmat),1);

%% Output the table
display(' ')
display('        Probability    Success   Success   Failure  Failure')
display(' tau       In Cone    No Warning  Warning No Warning Warning')
for i=1:ntau
    display(sprintf(['%5.0f %5.2f%%->%5.2f%% %7.2f%%' ...
        '%10.2f%% %7.2f%% %7.2f%% '],...
        [tauvec(i) 100*[probinit(i) probfinl(i) succnowarn(i) ...
        succwarn(i) failnowarn(i) failwarn(i)]]))
end
toc(tstart)


% Experiment 4: Functions with two local minimum points

%% Garbage collection and initialization
format compact %remove blank lines from output
format long e %lots of digits
clear all %clear all variables
close all %close all figures

%% Program parameters
TolXvec = [10^(-2) 10^(-4) 10^(-7)];
in_param.abstol = 0; %error tolerance
in_param.nmax = 10^7; %cost budget
tstart = tic;

%% Simulation parameters
nrep = 10000;
if (nrep >= 1000)
    warning('off','MATLAB:funmin_g:exceedbudget');
    warning('off','MATLAB:funmin_g:peaky');
end;
a1=5; b1=10; c1=0.5-0.5*rand(nrep,1);
a2=1; b2=10; c2=0.5+0.5*rand(nrep,1);
nTolX = length(TolXvec);

intnum = zeros(nrep,nTolX);
succfunmin = zeros(nrep,nTolX);
```

```
succfminbnd = zeros(nrep,nTolX);
xmin = zeros(nrep,nTolX);
exceedmat = zeros(nrep,nTolX);
exactsolu = zeros(nrep,nTolX);

for i=1:nTolX
    in_param.TolX = TolXvec(i);
    for j=1:nrep
        f=@(x) -a1*exp(-(b1*(x-c1(j))).^2)-a2*exp(-(b2*(x-c2(j))).^2);
        exactsolu(j) = ...
            fminbnd(f,0,(c1(j)+c2(j))/2,optimset('TolX',1e-9));
        [fmin,out_param] = funmin_g(f,in_param);
        xmin(j,i)=fminbnd(f,0,1,optimset('TolX',in_param.TolX));
        intnum(j,i) = size(out_param.intervals,2);
        exceedmat(j,i) = out_param.exceedbudget;
        for k=1:intnum(j,i)
            if exactsolu(j) <= out_param.intervals(2,k) && ...
                exactsolu(j) >= out_param.intervals(1,k)
                 succfunmin(j,i) = 1;
            end
        end
        if abs(xmin(j,i)-exactsolu(j)) <= in_param.TolX
            succfminbnd(j,i) = 1;
        end
    end
end

probfunmin=mean(succfunmin,1);
probnowarn=mean(succfunmin&(~exceedmat),1);
probwarn=mean(succfunmin&(exceedmat),1);
probfminbnd=mean(succfminbnd,1);

%% Output the table
display(' ')
display('          Success     Success     Success     Success')
display(' TolX                 No Warning  Warning     fminbnd' )
for i=1:nTolX
    display(sprintf([ '%1.0e    %7.2f%%   %7.2f%%   %7.2f%%   ...
        %7.2f%% '],...
        [TolXvec(i) 100*[probfunmin(i) probnowarn(i) probwarn(i)...
        probfminbnd(i)]]))
end
toc(tstart)
```

BIBLIOGRAPHY

[1] Richard P. Brent. *Algorithms for minimization without derivatives.* Courier Dover Publications, 2013.

[2] Sou-Cheng T. Choi. MINRES-QLP pack and reliable reproducible research via staunch scientific software. *Journal of Open Research Software*, 2014.

[3] Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang, and Yizhi Zhang. GAIL: Guaranteed Automatic Integration Library (Version 1.3), MATLAB Software, 2014.

[4] Nicholas Clancy, Yuhan Ding, Caleb Hamilton, Fred J. Hickernell, and Yizhi Zhang. The cost of deterministic, adaptive, automatic algorithms: Cones, not balls. *Journal of Complexity*, 30:21–45, 2014.

[5] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer methods for mathematical computations*, volume 8. Prentice-Hall Englewood Cliffs, NJ, 1977.

[6] Fred J. Hickernell, Lan Jiang, Yuewei Liu, and Art B. Owen. Guaranteed conservative fixed width confidence intervals via Monte Carlo sampling. In J. Dick, F. Y. Kuo, G. W. Peters, and I. H. Sloan, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2012*, pages 105–128. Springer Berlin Heidelberg, 2014.

[7] John K. Hunter and Bruno Nachtergaele. *Applied Analysis.* World Scientific, 2001.