GUARANTEED ADAPTIVE UNIVARIATE FUNCTION APPROXIMATION

BY

YUHAN DING

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

Approved _____
Advisor

Chicago, Illinois
December 2015

ACKNOWLEDGMENT

First and foremost I would like to express my deepest gratitude to my advisor Dr. Fred Hickernell for his excellent guidance, encouragement, and patience throughout my study at Illinois Institute of Technology. I appreciate all his contributions of time, ideas and supports to make my PhD experience fruitful and interesting. He is so smart, knowledgeable, always patient and ready to help, which provides me with an excellent atmosphere for doing research. His ability to analyze and solve research problems, high scientific standards and personality set a good example for me.

My special words of thanks should also go to Dr. Sou-Cheng Choi. She is not only like a supervisor to me, but also a good friend in life. She gives me very valuable advice regarding my research, especially the suggestion to improve the efficiency of the algorithms. For me, she is a role model as a junior member of academia. Her hard working and strong support mean more to me than I could ever express in words.

I am also very grateful to all key members in the GAIL team: Lan Jiang, Lluís Antoni Jiménez Rugama, Xin Tong, Yizhi Zhang, and Xuan Zhou. Without their support and help, we cannot build up such an interesting and useful toolbox GAIL based on MATLAB.

And I wish to express my sincere thanks to Dr. Gregory Fasshauer. His suggestions and comments offer me a broader vision on my research. He is always willing to help, no matter theoretical questions or MATLAB software problems.

I would also like to thank my committee members, Dr. Shuwang Li and Dr. Kevin Cassel, for their time, insightful questions and helpful comments.

Last but not least, I would like to say thank you to my parents. Their love and support are always my strength and driving force. I owe them everything and wish I could show them just how much I love and appreciate them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Numerical algorithms for univariate function approximation attempt to provide approximate solutions that differ from the original function by no more than a user-specified error tolerance. The computational cost is often determined adaptively by the algorithm based on the function values sampled. While adaptive algorithms are widely used in practice, most lack guarantees, i.e., conditions on input functions that ensure the error tolerance is met.

In this dissertation we establish guaranteed adaptive numerical algorithms for univariate function approximation using piecewise linear splines. We introduce a guaranteed globally adaptive algorithm, `funappxglobal_g`, in Chapter 2, along with sufficient conditions for the success of `funappxglobal_g`. Two-sided bounds on the computational cost are given in Theorem 1. These bounds are of the same order as the computational cost for an algorithm that knows the infinity norm of the second derivative of the input function as a priori. Lower bound on the complexity of the problem is also provided in Theorem 3. To illustrate the advantages of `funappxglobal_g`, corresponding numerical experiments are presented in Section 2.7.

The cost of a globally adaptive algorithm is determined by the most peaky part of the input function. In contrast, locally adaptive algorithms sample more points where the function is peaky and fewer points elsewhere. In Chapter 3, we establish a locally adaptive algorithm, `funappx_g`, with sufficient conditions for its success. An upper bound on the computational cost is also given in Theorem 4. One GUI example is presented to show how `funappx_g` works. Some interesting function approximation problems in computational graphics are also presented.

The key to analyzing these adaptive algorithms is looking at the error for cones of input functions rather than balls of input functions. Non-convex cones provide a setting where adaption may be beneficial.

CHAPTER 1

INTRODUCTION

1.1 Function Interpolation

Function approximation is a classical problem in computational mathematics. The aim is to find a function $\hat{f}$ of simple form that is close to some complicated function, f. The approximation, $\hat{f}$, is computed from values of f at a finite number of sample points, $\{x_i\}_{i=1}^n$. Here we focus on *interpolation*, which means that

$$\hat{f}(x_i) = f(x_i), \qquad i = 1, \ldots, n.$$

Polynomials form a well-known set of candidates for $\hat{f}$. An alternative approach is to divide the approximation interval into a collection of subintervals and construct a different approximating polynomial on each subinterval. This is called piecewise-polynomial approximation.

In this dissertation, our purpose is to efficiently fit a piecewise linear interpolant to f to arbitrary accuracy by carefully choosing the position and number of data points. Our algorithm will be data-driven, choosing the sample points depending on the function data. Its accuracy will be rigorously guaranteed.

1.2 Motivation

Numerical algorithms for function approximation aim to obtain an approximate function $\hat{f}$ that differs from f in $\mathcal{L}^\infty$ norm by no more than a given error tolerance, ε, i.e.,

$$\left\| f - \hat{f} \right\|_\infty \leq \varepsilon.$$

If the algorithms can adjust their effort based on the information about the function obtained through sampling, we call them adaptive algorithms. In fact, adaptive algorithms are very common in numerical software packages. Examples include the

MATLAB Chebfun toolbox [7] for function approximation, integration and solving differential equations; MATLAB's `integral` [17] and the quadrature algorithms in the NAG Library [18] for the integration problem. While these adaptive algorithms work well for many cases, they have no rigorous justification. The methods used to determine the computational cost are either heuristics or asymptotic error estimates that do not hold for finite sample sizes.

On the other hand, most existing guaranteed automatic algorithms are not adaptive. For example, if we use piecewise linear splines to approximate f on the interval $[a, b]$, we can obtain the following upper bound on the approximation error:

$$\left\| f - \hat{f} \right\|_\infty \leq \frac{(b-a)^2}{8(n-1)^2} \left\| f'' \right\|_\infty .$$

See details and a proof in Lemma 1. Of course, we can easily obtain the computational effort to reach the tolerance if we know the value of $\|f''\|_\infty$. However, in practice, it is often difficult to obtain the value of $\|f''\|_\infty$ or an upper bound on $\|f''\|_\infty$. Even if we know that $\|f''\|_\infty$ is no more than some σ, the related numerical algorithm would not be adaptive. Functions for which $\|f''\|_\infty$ is much less than σ would require just as much effort as those for which $\|f''\|_\infty$ is nearly σ. Rather than assuming $\|f''\|_\infty \leq \sigma$, our adaptive algorithms use function data to construct upper bounds on $\|f''\|_\infty$. The key idea is to identify a suitable weaker semi-norm for f

$$f \mapsto \left\| f' - \frac{f(b) - f(a)}{b-a} \right\|_\infty ,$$

and use it to construct a cone of input functions:

$$\mathcal{C}_{n^*} := \left\{ f \in \mathcal{W}^{2,\infty} : \|f''\|_\infty \leq \frac{2n^*}{b-a} \left\| f' - \frac{f(b) - f(a)}{b-a} \right\|_\infty \right\}. \tag{1.1}$$

Here n^* is a parameter, that intuitively represents the minimal size minus 2. Note that the stronger semi-norm is bounded above by a multiple of the weaker semi-norm. More details of this cone are given in Chapter 2.

There are some theoretical results providing conditions under which adaption is useful and when it is not useful. See for example, the comprehensive survey by Novak [14]. It notes that the advantage of adaptive methods can be very large for the linear problems defined on non-convex sets, even exponentially large. Actually, our cone in (1.1) is a non-convex set. More details of the advantages of the cone are presented later in next section.

More recent articles about the power of adaption for approximating functions and for integration have been published by Plaskota and Wasilkowski [15, 16]. They are more interested in functions with singularities. However here we consider a somewhat different situation. Our focus is on cones of input functions with finite second derivatives. Moreover our definition of cone does not depend on the locations of singularities in the higher derivatives. With adaptive stopping rules based on $\mathcal{C}_{n^*}$, we can establish our guaranteed adaptive algorithms.

Our research is a fundamental work of establishing a guaranteed adaptive interpolation algorithm. Piecewise linear interpolation only has a low order of convergence, which may limit the immediate practicality of this work. But it is crucial first step towards rigorous justification for higher order interpolation methods, such as Chebfun. Moreover, our work can and already has been extended to a guaranteed adaptive algorithm for global optimization [19]. Furthermore, one good application of piecewise linear interpolation is function plotting. Our algorithm could be used to adaptively choose the data points to plot a function, say using the `plot` in MATLAB. The error criterion would be chosen so that the plot is accurate to the precision of the eye.

1.3 Why Cones

Most existing numerical analysis is focused on balls of input functions, $\mathcal{B}_\sigma$. A

ball of radius σ is a set of objects whose size is no greater than σ. The algorithms arising from analysis using balls are non-adaptive. The analysis here focuses on cones of input functions, $\mathcal{C}_{n^*}$, which allows us to derive data-driven error bounds and construct adaptive, automatic algorithms. Cones $\mathcal{C}$ are sets that contain all multiples of their elements, i.e.,

$$f \in \mathcal{C} \Rightarrow cf \in \mathcal{C} \qquad \text{for all } c \in \mathbb{R}.$$

We have two reasons for favoring cones.

Function approximation problem and the fixed-cost algorithms, $\{A_n\}_{n\in\mathcal{I}}$, commonly encountered in practice are positively homogeneous; the error functional,

$$\mathrm{err}_n(f) = \|f - A_n(f)\|_\infty ,$$

is also positively homogeneous. This naturally suggests data-driven error bounds, $\widehat{\mathrm{err}}_n(f)$, that are positively homogeneous. If $\mathrm{err}_n(f) \le \widehat{\mathrm{err}}_n(f)$, then $\mathrm{err}_n(cf) \le \widehat{\mathrm{err}}_n(cf)$ for all c, so the set of input functions for which a positively homogeneous error bound is valid is a cone.

A second reason to favor cones is that we want to spend less effort solving problems for input functions that are "easy", i.e., we want an adaptive algorithm. It is not hard to find that our adaptive algorithms possess a stronger optimality than the non-adaptive one. In particular, the costs of the non-adaptive algorithms do not depend on the norms of the input functions, but the costs of the adaptive algorithms do so in a favorable way.

There are rigorous results from information based complexity theory giving general conditions under which adaptive algorithms have no significant advantage over non-adaptive algorithms (e.g., see [20, Chapter 4, Theorem 5.2.1] and [14]). For adaption to be useful, we must violate one of these conditions. In particular, we

violate the condition that the set of input functions is convex. We will present more details of why our cone is non-convex in Section 2.6.

1.4 Outline of the Thesis

In this section, we give the outline of the dissertation. Chapter 2 starts with the problem setting. Error tolerance ε and a black box of function f to give the function values are the input of our algorithm. We don't need any further information, such as the location of peaks, to establish our algorithm. After that we describe the adaptive algorithm in detail and provide a proof of success for cones of input functions. Along with our upper bound on the computational cost of our algorithm, we also present a lower bound on the complexity of the problem. In addition, numerical examples are also given to show how the algorithm works in comparison to the Chebfun toolbox.

The algorithm we present in Chapter 2 is only globally adaptive, which means that if there is a peaky part on a small subinterval, our algorithm will increase the number of sample points on the whole interval simultaneously to reach the error tolerance. To be more efficient, we want to construct algorithms with local adaption, where the sampling density varies according to the function data. Instead of working on the whole interval, we introduce a partition of the whole interval in Chapter 3. We still analyze our algorithm for a cone of input functions. The difference is that the input functions need to satisfy inequalities on all subintervals of $[a, b]$, not only for the whole interval $[a, b]$. Based on the cone condition, we derive an error bound that holds on any subinterval. After that, we build our locally adaptive algorithm. We also show an upper bound on the computational cost of the locally adaptive algorithm and provide several interesting numerical examples in Chapter 3. In addition, we also present a comparison between globally and locally adaptive guaranteed algorithms by several kinds of test functions.

We conclude this dissertation in Chapter 4 with a summary of contributions and results. Potential future research topics are also outlined here.

In Appendix, we attach the MATLAB code for the globally and locally adaptive algorithms. Users can find the most recent version in the GAIL library at the following link: http://gailgithub.github.io/GAIL_Dev.

CHAPTER 2

GUARANTEED GLOBALLY ADAPTIVE ALGORITHM FOR UNIVARIATE FUNCTION APPROXIMATION

2.1 Background

In [5], we presented the idea of cones to establish a framework for deterministic guaranteed, adaptive algorithms. To illustrate the algorithms, univariate function approximation on the interval $[0, 1]$ was presented as one special example in [5]. In addition to deterministic algorithms, the cone idea can also be applied to algorithms with probabilistic error criteria, such as the simple Monte Carlo method in [8] and the Quasi-Monte Carlo Method in [9, 12]. This chapter is a minor extension of our work in [5], to establish a guaranteed (globally) adaptive algorithm for univariate function approximation on the interval $[a, b]$.

As we use piecewise linear splines to approximate, we must select a proper semi-norm and function space for our error analysis. Next, we construct a cone of input functions based on the selected semi-norm and a weaker semi-norm. After that, we place evenly spaced knots in the domain to approximate the original function via piecewise linear interpolation. Moreover, we use the same data to approximate the weaker semi-norm and bound the selected semi-norm. Based on these results, we can do error estimation on the cone condition. If the error estimate satisfies the error tolerance, we return the approximate function and stop. Otherwise, we increase the sample size until the error estimation reaches the error tolerance.

Regardless of the input function, our algorithm starts with the same number of points. But the algorithm ends up using a different final number of points depending on the input function. For example, for linear functions, our algorithm stops after the first iteration and returns an approximation that is exact. But for nonlinear functions, our algorithm generally needs more iterations to reach the error tolerance.

Besides constructing the globally adaptive algorithm, we also present a theorem to guarantee our algorithm works for all the functions inside a cone. Moreover, we also give a two-sided bound on the computational cost of our algorithm.

We not only develop a theorem, but we also implement our algorithms in the MATLAB Guaranteed Automatic Integration Library (GAIL). The globally adaptive algorithm for univariate function approximation was released in the first version of GAIL [4] in 2013. An updated version of `funappxglobal_g` was released in GAIL version 2.0 [2] in 2014.

2.2 Problem Setting

In this section, we define the problem to be solved and introduce our notation. We want to approximate function f on interval $[a, b]$. Suppose we have evenly spaced data $(x_i, f(x_i))$, $i = 1, \ldots, n$, where

$$x_i = \frac{i-1}{n-1}(b-a) + a.$$

Denote a sequence of fixed-cost algorithms as $\{A_n\}_{n \in \mathcal{I}}$, indexed by their computational cost, n, with $\mathcal{I} = \{3, 4, \ldots\}$. Here we use piecewise linear interpolation:

$$A_n(f)(x) := \frac{n-1}{b-a}\left[f(x_i)(x_{i+1} - x) + f(x_{i+1})(x - x_i)\right], \qquad \forall x \in [x_i, x_{i+1}]. \tag{2.1}$$

We can easily obtain the error bound on the approximation by A_n.

Lemma 1. *For any function $f : [a, b] \to \mathbb{R}$ which has a second derivative, A_n defined in* (2.1)*, it follows that*

$$\|f - A_n(f)\|_\infty \leq \frac{(b-a)^2}{8(n-1)^2}\|f''\|_\infty,$$

where

$$\|f''\|_\infty := \|f''\|_{\infty,[a,b]} = \sup_{a \leq x \leq b} |f''(x)|.$$

Proof. We can obtain the error bound from most elementary textbooks on numerical analysis (see for example, Theorem 3.3 in [1]). To make it complete, we present the proof.

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

$$\begin{aligned} f(x) - A_n(f)(x) &= f(x) - \frac{n-1}{b-a}\left[f(x_i)(x_{i+1}-x) + f(x_{i+1})(x-x_i)\right] \\ &= \frac{n-1}{b-a}\int_{x_i}^{x_{i+1}} v_i(t,x) f''(t)\,\mathrm{d}t, \end{aligned} \tag{2.2a}$$

where the continuous, piecewise differentiable Peano kernel v is defined in

$$\begin{aligned} v_i(t,x) &:= -(x_{i+1} - \max(t,x))(\min(t,x) - x_i) \\ &= \begin{cases} (x_{i+1}-x)(x_i - t), & x_i \le t \le x, \\ (x - x_i)(t - x_{i+1}), & x < t \le x_{i+1}. \end{cases} \end{aligned} \tag{2.2b}$$

To derive the error bound for $A_n(f)$ we make use of (2.2a):

$$\begin{aligned} \|f - A_n(f)\|_\infty &= \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\ldots,n-1}} |f(x) - A_n(f)(x)| \\ &\le \frac{n-1}{b-a} \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\ldots,n-1}} \int_{x_i}^{x_{i+1}} |v_i(t,x) f''(t)|\,\mathrm{d}t \\ &\le \frac{n-1}{b-a}\,\|f''\|_\infty \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\ldots,n-1}} \int_{x_i}^{x_{i+1}} |v_i(t,x)|\,\mathrm{d}t. \end{aligned}$$

We know that

$$\begin{aligned} &\int_{x_i}^{x_{i+1}} |v_i(t,x)|\,\mathrm{d}t \\ =&\int_{x_i}^{x} |(x_{i+1}-x)(x_i - t)|\,\mathrm{d}t + \int_{x}^{x_{i+1}} |(x-x_i)(t-x_{i+1})|\,\mathrm{d}t \\ =&(x_{i+1}-x)\int_{x_i}^{x} (t - x_i)\,\mathrm{d}t + (x - x_i)\int_{x}^{x_{i+1}} (x_{i+1} - t)\,\mathrm{d}t \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2}(x_{i+1}-x)(t-x_i)^2\Big|_{x_i}^{x} - \frac{1}{2}(x-x_i)(x_{i+1}-t)^2\Big|_{x}^{x_{i+1}} \\
&= \frac{1}{2}(x-x_i)(x_{i+1}-x)(x_{i+1}-x_i) \\
&= \frac{b-a}{2(n-1)}(x-x_i)(x_{i+1}-x).
\end{aligned}
$$

Thus we have

$$
\begin{aligned}
\|f - A_n(f)\|_\infty &\le \frac{n-1}{b-a}\,\|f''\|_\infty \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\dots,n-1}} \frac{b-a}{2(n-1)}(x-x_i)(x_{i+1}-x) \\
&= \|f''\|_\infty \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\dots,n-1}} \frac{(x-x_i)(x_{i+1}-x)}{2} \\
&= \frac{(b-a)^2}{8(n-1)^2}\,\|f''\|_\infty .
\end{aligned}
$$

□

By Lemma 1, we can bound the error in terms of $\|f''\|_\infty$. That's why we need $\|f''\|_\infty$ to be finite. To obtain an algorithm with guaranteed accuracy, we need the input function to have at least a second derivative. Thus, we introduce the following Sobolev space:

$$
\mathcal{W}^{2,\infty} := \mathcal{W}^{2,\infty}[a,b] = \{f \in C[a,b] : \|f''\|_\infty < \infty\}. \tag{2.3}
$$

Given a "nice" subset of input functions, $\mathcal{N} \subseteq \mathcal{W}^{2,\infty}$, our goal is to establish an algorithm A, $A : (\mathcal{W}^{2,\infty}, (0,\infty)) \to \mathcal{L}^\infty[a,b]$, such that

$$
\|f - A(f,\varepsilon)\|_\infty \le \varepsilon, \qquad \forall f \in \mathcal{N},
$$

where A is an iterative algorithm and at each iteration using the fixed-cost algorithm A_n defined in (2.1). That is

$$
A(f,\varepsilon) = A_n(f)
$$

for some n that depends on ε and the values of f used in $A_n(f)$. The computational cost of A for a given f and ε is the n required and is denoted

$$
\mathrm{cost}(A,f,\varepsilon) \in \mathbb{N}_0.
$$

To obtain an adaptive algorithm, rather than assuming an upper bound on $\|f''\|_\infty$, the key idea is to identify a suitable semi-norm on $\mathcal{W}^{2,\infty}$, that is weaker than $\|f''\|_\infty$. We introduce a weaker semi-norm of f

$$\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty := \left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_{\infty,[a,b]} = \sup_{a\le x\le b}\left|f'(x) - \frac{f(b)-f(a)}{b-a}\right|.$$

Based on this weaker semi-norm, we can introduce a "nice" set $\mathcal{N} = \mathcal{C}_{n^*}$, as defined early in (1.1). Note $n^* = \eta(b-a)$, where η is some non-decreasing function

$$\eta : (0,\infty) \to \mathbb{N}. \tag{2.4}$$

Note that $n^* + 2$ represents the minimum sample size that our algorithm uses. This cone condition is used to derive that data driven upper bound on $\|f''\|_\infty$ in Lemma 3. Our goal now is to construct an algorithm A such that

$$\|f - A(f,\varepsilon)\|_\infty \le \varepsilon, \qquad \forall f \in \mathcal{C}_{n^*}.$$

2.3 Error Estimation

We want to approximate the weaker semi-norm $\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty$ and bound the stronger semi-norm $\|f''\|_\infty$ by the same data $(x_i, f(x_i))$ used to construct the piecewise linear splines. Our approximation for the weaker semi-norm is as follows:

$$\widetilde{F}_n(f) := \|A_n(f)' - A_2(f)'\|_\infty = \sup_{i=1,\dots,n-1}\left|\frac{n-1}{b-a}[f(x_{i+1}) - f(x_i)] - \frac{f(b)-f(a)}{b-a}\right|. \tag{2.5}$$

We can obtain a two-sided error bound for this approximation.

Lemma 2. *For all $f \in \mathcal{W}^{2,\infty}$ and $\widetilde{F}_n$ given in* (2.5)*, we have*

$$0 \le \left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty - \widetilde{F}_n(f) \le \frac{b-a}{2(n-1)}\|f''\|_\infty. \tag{2.6}$$

Proof. Given $\widetilde{F}_n$ and A_n as mentioned before, we want to show that $\widetilde{F}_n(f)$ never overestimates $\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty$. This is true because

$$\begin{aligned}
\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty &= \left\|f' - A_2(f)'\right\|_\infty = \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\dots,n-1}} |f'(x) - A_2(f)'(x)| \\
&\ge \sup_{i=1,\dots,n-1} \frac{n-1}{b-a} \int_{x_i}^{x_{i+1}} \left| f'(x) - \frac{f(b)-f(a)}{b-a} \right| \mathrm{d}x \\
&\ge \sup_{i=1,\dots,n-1} \frac{n-1}{b-a} \left| \int_{x_i}^{x_{i+1}} \left[f'(x) - \frac{f(b)-f(a)}{b-a} \right] \mathrm{d}x \right| \\
&= \sup_{i=1,\dots,n-1} \frac{n-1}{b-a} \left| f(x_{i+1}) - f(x_i) - \frac{f(b)-f(a)}{n-1} \right| = \widetilde{F}_n(f).
\end{aligned}$$

Thus, we get the lower bound.

Next, we derive the upper bound. By (2.2), for $x_i \le x \le x_{i+1}$ we can obtain

$$f'(x) - A_n(f)'(x) = \frac{n-1}{b-a} \int_{x_i}^{x_{i+1}} \frac{\partial v_i}{\partial x}(t,x) f''(t)\,\mathrm{d}t. \tag{2.7}$$

To find an upper bound, we applying the triangle inequality:

$$\begin{aligned}
\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty - \widetilde{F}_n(f) &= \|f' - A_2(f)'\|_\infty - \|A_n(f)' - A_2(f)'\|_\infty \\
&\le \left\|f' - A_n(f)'\right\|_\infty,
\end{aligned}$$

since $(f - A_n(f))(x)$ vanishes at $x = a, b$. Using (2.7) it then follows that

$$\begin{aligned}
&\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty - \widetilde{F}_n(f) \\
&\qquad \le \left\|f' - A_n(f)'\right\|_\infty \\
&\qquad = \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\dots,n-1}} \left| f'(x) - \frac{n-1}{b-a}[f(x_{i+1}) - f(x_i)] \right| \\
&\qquad = \frac{n-1}{b-a} \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\dots,n-1}} \left| \int_{x_i}^{x_{i+1}} \frac{\partial v_i}{\partial x}(t,x) f''(t)\,\mathrm{d}t \right| \\
&\qquad \le \frac{n-1}{b-a} \|f''\|_\infty \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\dots,n-1}} \int_{x_i}^{x_{i+1}} \left| \frac{\partial v_i}{\partial x}(t,x) \right| \mathrm{d}t \\
&\qquad = \frac{n-1}{b-a} \|f''\|_\infty \sup_{\substack{x_i \le x \le x_{i+1} \\ i=1,\dots,n-1}} \left\{ \frac{(b-a)^2}{2(n-1)^2} - (x - x_i)(x_{i+1} - x) \right\}
\end{aligned}$$

$$= \frac{b-a}{2(n-1)} \|f''\|_\infty .$$

Therefore, we obtain the two-sided error bounds. □

Lemma 2 provides an upper bound on $\widetilde{F}_n(f)$ in terms of the weaker semi-norm and also the stronger semi-norm in the following corollary.

Corollary 1. *For all $f \in \mathcal{W}^{2,\infty}$ and $\widetilde{F}_n$ given in (2.5), we have*

$$\widetilde{F}_n(f) \le \left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty \le \frac{(b-a)}{2} \|f''\|_\infty .$$

Proof. By Lemma 2, we obtain

$$\left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty - \widetilde{F}_n(f) \le \frac{b-a}{2(n-1)} \|f''\|_\infty .$$

Since $\widetilde{F}_2(f) = \|A_2(f)' - A_2(f)'\|_\infty = 0$ by definition, then the above inequality implies that

$$\left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty \le \frac{b-a}{2} \|f''\|_\infty ,$$

which is equivalent to

$$\frac{2}{b-a} \left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty \le \|f''\|_\infty .$$

Also by Lemma 2, we have

$$\widetilde{F}_n(f) \le \left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty .$$

Therefore, it is easy for us to obtain

$$\widetilde{F}_n(f) \le \left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty \le \frac{(b-a)\|f''\|_\infty}{2}.$$

□

Also, we can use $\widetilde{F}_n(f)$ to construct an upper bound for $\|f''\|_\infty$. Combining the definition of the cone and Lemma 2, we obtain the following lemma.

Lemma 3. *Let $\mathcal{C}_{n^*}$ and $\widetilde{F}_n$ be defined in* (1.1) *and* (2.5) *respectively. For all $f \in \mathcal{C}_{n^*}$, if $n > 1 + n^*$, we obtain*

$$\left\| f' - \frac{f(b) - f(a)}{b - a} \right\|_\infty \leq \frac{1}{1 - n^*/(n-1)} \widetilde{F}_n(f),$$
$$\|f''\|_\infty \leq \frac{2n^*}{(b-a)(1 - n^*/(n-1))} \widetilde{F}_n(f).$$

Proof. Combining cone condition (1.1) and Lemma 2, we get

$$\begin{aligned} \left\| f' - \frac{f(b) - f(a)}{b - a} \right\|_\infty - \widetilde{F}_n(f) &\leq \frac{b-a}{2(n-1)} \|f''\|_\infty \\ &\leq \frac{b-a}{2(n-1)} \frac{2n^*}{b-a} \left\| f' - \frac{f(b) - f(a)}{b - a} \right\|_\infty \\ &= \frac{n^*}{n-1} \left\| f' - \frac{f(b) - f(a)}{b - a} \right\|_\infty. \end{aligned}$$

Rearranging the above inequality, we have

$$\left(1 - \frac{n^*}{n-1}\right) \left\| f' - \frac{f(b) - f(a)}{b - a} \right\|_\infty \leq \widetilde{F}_n(f). \tag{2.8}$$

If

$$1 - \frac{n^*}{n-1} > 0 \Leftrightarrow n > 1 + n^*,$$

we can divide both sides of (2.8) by $1 - n^*/(n-1)$ and get

$$\left\| f' - \frac{f(b) - f(a)}{b - a} \right\|_\infty \leq \frac{1}{1 - n^*/(n-1)} \widetilde{F}_n(f).$$

Applying again the definition of the cone (1.1), we have

$$\|f''\|_\infty \leq \frac{2n^*}{b-a} \left\| f' - \frac{f(b) - f(a)}{b - a} \right\|_\infty \leq \frac{2n^*}{(b-a)(1 - n^*/(n-1))} \widetilde{F}_n(f).$$

□

Combining Lemma 1 and Lemma 3, we can obtain an adaptive upper bound on the approximation error.

Lemma 4. *Let A_n and $\widetilde{F}_n$ be defined in* (2.1) *and* (2.5) *respectively. For any $f \in \mathcal{C}_{n^*}$, if $n > 1 + n^*$, we obtain*

$$\|f - A_n(f)\|_\infty \leq \frac{n^*(b-a)}{4(n-1)(n-1-n^*)} \widetilde{F}_n(f).$$

By Lemma 4, we know if we can find a large n to make the following inequality valid, i.e.,

$$\frac{n^*(b-a)}{4(n-1)(n-1-n^*)} \widetilde{F}_n(f) \leq \varepsilon,$$

we reach the error tolerance. Hence, the above inequality is the stopping criterion in our algorithm.

Moreover, a lower bound on $\|f''\|_\infty$ can be derived similarly by using a centered difference. Specifically, for $n \geq 3$,

$$F_n(f) := \frac{(n-1)^2}{(b-a)^2} \sup_{i=1,\ldots,n-2} |f(x_i) - 2f(x_{i+1}) + f(x_{i+2})| \,. \tag{2.9}$$

We know $F_n(f)$ never overestimates $\|f''\|_\infty$, i.e., $F_n(f) \leq \|f''\|_\infty$. It follows using Hölder's inequality that

$$\begin{aligned}
F_n(f) &= \frac{(n-1)^2}{(b-a)^2} \sup_{i=1,\ldots,n-2} \left| \int_{x_i}^{x_{i+2}} \left[\frac{b-a}{n-1} - |x - x_{i+1}| \right] f''(x) \, \mathrm{d}x \right| \\
&\leq \frac{(n-1)^2}{(b-a)^2} \sup_{i=1,\ldots,n-2} \|f''\|_\infty \int_{x_i}^{x_{i+2}} \left| \frac{b-a}{n-1} - |x - x_{i+1}| \right| \mathrm{d}x \\
&= \frac{(n-1)^2}{(b-a)^2} \|f''\|_\infty \sup_{i=1,\ldots,n-2} \left[\int_{x_i}^{x_{i+1}} \left| \frac{b-a}{n-1} - (x_{i+1} - x) \right| \mathrm{d}x \right. \\
&\quad \left. + \int_{x_{i+1}}^{x_{i+2}} \left| \frac{b-a}{n-1} - (x - x_{i+1}) \right| \mathrm{d}x \right] \\
&= \frac{(n-1)^2}{(b-a)^2} \|f''\|_\infty \sup_{i=1,\ldots,n-2} \left[\left(\frac{b-a}{n-1} x + \frac{(x_{i+1} - x)^2}{2} \right) \Bigg|_{x_i}^{x_{i+1}} \right. \\
&\quad \left. + \left(\frac{b-a}{n-1} x - \frac{(x - x_{i+1})^2}{2} \right) \Bigg|_{x_{i+1}}^{x_{i+2}} \right] \\
&= \|f''\|_\infty \,.
\end{aligned}$$

With $\widetilde{F}_n(f)$ and $F_n(f)$, we can construct the minimal value of n^* such that $f \in \mathcal{C}_{n^*}$.

Corollary 2. *Let* $\widetilde{F}_n(f)$ *and* $F_n(f)$ *be defined in* (2.5) *and* (2.9) *respectively.* $\forall f \in \mathcal{C}_{n^*}$*, we have*

$$n^*_{\min,n} := \frac{F_n(f)}{2\widetilde{F}_n(f)/(b-a) + F_n(f)/(n-1)} \le n^*.$$

Proof. By Lemma 3 and $F_n(f) \le \|f''\|_\infty$, we have

$$\begin{aligned}
& F_n(f) \le \frac{2n^*}{(b-a)(1-n^*/(n-1))}\widetilde{F}_n(f) \\
\iff & F_n(f) - \frac{n^*}{n-1}F_n(f) \le \frac{2n^*}{b-a}\widetilde{F}_n(f) \\
\iff & F_n(f) \le \left(\frac{2\widetilde{F}_n(f)}{b-a} + \frac{F_n(f)}{n-1}\right) n^* \\
\iff & n^* \ge \frac{F_n(f)}{2\widetilde{F}_n(f)/(b-a) + F_n(f)/(n-1)}.
\end{aligned}$$

□

Corollary 2 offers us a necessary condition to check if we have a large enough cone for the input function.

2.4 Globally Adaptive Algorithm `funappxglobal_g`

Algorithm `funappxglobal_g`. (Adaptive, automatic, multi stage, univariate function recovery) Let the cone constant n^* and the sequence of algorithms $\{A_n\}_{n\in\mathcal{I}}$, $\{\widetilde{F}_n\}_{n\in\mathcal{I}}$, and $\{F_n\}_{n\in\mathcal{I}}$ be as described above. Set $i = 1$, $n_1 = n^* + 2$. For any error tolerance ε and input function f, do the following:

Stage 1. Estimate $\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty$ **and bound** $\|f''\|_\infty$**.** Compute $\widetilde{F}_{n_i}(f)$ in (2.5) and $F_{n_i}(f)$ in (2.9).

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

$$n^*_{\min,n_i} = \frac{F_{n_i}(f)}{2\widetilde{F}_{n_i}(f)/(b-a) + F_{n_i}(f)/(n_i-1)}.$$

If $n^* \geq n^*_{\min,n_i}$, then go to stage 3. Otherwise, set $n^* = 2n^*_{\min,n_i}$. If $n_i \geq n^* + 2$, then go to stage 3. Otherwise, choose

$$n_{i+1} = 1 + (n_i - 1)\left\lceil \frac{2n^* + 1}{2n_i - 2} \right\rceil.$$

Go to Stage 1.

Stage 3. Check for convergence. Check whether n_i is large enough to satisfy the error tolerance, i.e.

$$\widetilde{F}_{n_i}(f) \leq \frac{4\varepsilon(n_i - 1)(n_i - 1 - n^*)}{n^*(b-a)}. \tag{2.10}$$

If this is true, then return $A_{n_i}(f)$ and terminate the algorithm. If this is not true, choose

$$n_{i+1} = 1 + (n_i - 1)\max\left\{2, \left\lceil \frac{1}{(n_i - 1)}\sqrt{\frac{n^*(b-a)\widetilde{F}_{n_i}(f)}{4\varepsilon}} \right\rceil \right\}. \tag{2.11}$$

Go to Stage 1.

Remark 1. We choose the cone constant n^* based on the non-decreasing function η defined in (2.4). It is easy to understand we should have a large cone when the interval is large. Therefore we introduce two positive integers n_{hi} and n_{lo} in algorithm to control the cone constant n^*, where $n_{\text{hi}} \geq n_{\text{lo}}$. Here we choose

$$\eta(b-a) = \left\lceil n_{\text{hi}} \left(\frac{n_{\text{lo}}}{n_{\text{hi}}} \right)^{\frac{1}{1+b-a}} \right\rceil.$$

As the initial number of points $n_1 = n^* + 2$, we know when the interval is relatively large, the initial number points is closer to n_{hi}. If the interval is short, the initial number of points is closer to n_{lo}. But no matter how to choose different n_{hi} and n_{lo}, the initial number of points n_1 is always greater than or equal to 3, as we need at least three points to compute $F_{n_1}(f)$.

Remark 2. This necessary condition

$$n^*_{\min,n_i} = \frac{F_{n_i}(f)}{2\widetilde{F}_{n_i}(f)/(b-a) + F_{n_i}(f)/(n_i-1)} \le n^*,$$

is not sufficient for f to lie in $\mathcal{C}_{n^*}$. Even the condition is satisfied, it is still possible $f \notin \mathcal{C}_{n^*}$. But if $n^*_{\min,n_i} > n^*$, we know $f \notin \mathcal{C}_{n^*}$ for sure. That's why in Algorithm `funappxglobal_g`, we increase n^* to $2n^*_{\min,n}$ whenever $n^*_{\min,n}$ rises above n^*.

Remark 3. In Algorithm `funappxglobal_g`, we do not only increase the number of points, such as $n_{i+1} > n_i$, but also embed the data for A_{n_i} in the data for $A_{n_{i+1}}$. That is, in each iteration of Algorithm `funappxglobal_g`, we use the information from the previous iteration. As our data information is evenly located, we can solve the embedding problem by setting a multiplier based on the information from the last iteration. Also to make our algorithm conservative, we set the multiplier to be

$$\max\left\{2, \left\lceil \frac{1}{(n_i-1)}\sqrt{\frac{n^*(b-a)\widetilde{F}_{n_i}(f)}{4\varepsilon}} \right\rceil \right\}.$$

As we keep using the information from the previous iterations, there is no need to compute all the data values in each iteration. That is, when we implement `funappxglobal_g` in GAIL [3], we only compute new added points in each step. After that, we insert these new data points and function values to their corresponding positions and do the error estimation.

Remark 4. For practical reasons one may impose a computational cost budget, $N_{\max}$. If this is done, Algorithm `funappxglobal_g` computes the correct answer within budget for $f \in \mathcal{C}_{n^*}$ provided either of the cost upper bounds in Theorem 1 does not exceed $N_{\max}$. If `funappxglobal_g` can not reach the error tolerance within the budget $N_{\max}$, `funappxglobal_g` will give the user a warning message of exceeding the budget.

2.5 Computational Cost

2.5.1 Upper bound on Computational Cost.

Theorem 1. *Let A be the adaptive piecewise linear spline defined by Algorithm* `funappxglobal_g`*, and let n_1, n^*, and ε be the inputs and parameters described there. Let $\mathcal{C}_{n^*}$ be the cone of functions defined in* (1.1)*. Then it follows that Algorithm* `funappxglobal_g` *is successful for all functions f in $\mathcal{C}_{n^*}$, i.e., $\|f - A(f,\varepsilon)\|_\infty \le \varepsilon$. Moreover, the cost of this algorithm is bounded below and above as follows:*

$$\begin{aligned}
&\max\left(\left\lceil\frac{2n^*+1}{2}\right\rceil, \left\lceil\sqrt{\frac{(b-a)^2\left\|f''\right\|_\infty}{8\varepsilon}}\right\rceil\right)+1 \\
&\qquad\le \max\left(\left\lceil\frac{2n^*+1}{2}\right\rceil, \left\lceil\sqrt{\frac{n^*(b-a)\left\|f'-\frac{f(b)-f(a)}{b-a}\right\|_\infty}{4\varepsilon}}\right\rceil\right)+1 \\
&\qquad\le \mathrm{cost}(A,f;\varepsilon) \le \left\lceil\sqrt{\frac{n^*(b-a)\left\|f'-\frac{f(b)-f(a)}{b-a}\right\|_\infty}{\varepsilon}}+2n^*\right\rceil+1 \\
&\qquad\le \left\lceil\sqrt{\frac{n^*(b-a)^2\left\|f''\right\|_\infty}{2\varepsilon}}+2n^*\right\rceil+1. \quad (2.12)
\end{aligned}$$

The algorithm is computationally stable, meaning that the minimum and maximum costs for all functions, f, with fixed $\left\|f'-\frac{f(b)-f(a)}{b-a}\right\|_\infty$ or $\|f''\|_\infty$, are an ε-independent constant of each other.

Proof. By Lemma 3, we can obtain

$$\frac{(b-a)(1-n^*/(n-1))}{2n^*}\left\|f''\right\|_\infty \le \left(1-\frac{n^*}{n-1}\right)\left\|f'-\frac{f(b)-f(a)}{b-a}\right\|_\infty \le \widetilde{F}_n(f). \quad (2.13)$$

As we want to find

$$\frac{n^*(b-a)}{4(n-1)(n-1-n^*)}\widetilde{F}_n(f) \le \varepsilon,$$

by (2.13), we have

$$\frac{(b-a)^2}{8(n-1)^2}\left\|f''\right\|_\infty \le \frac{n^*(b-a)}{4(n-1)^2}\left\|f'-\frac{f(b)-f(a)}{b-a}\right\|_\infty \le \frac{n^*(b-a)}{4(n-1)(n-1-n^*)}\widetilde{F}_n(f).$$

Thus we can easily obtain

$$\left\lceil \sqrt{\frac{(b-a)^2 \left\|f''\right\|_\infty}{8\varepsilon}} \right\rceil + 1$$

$$\leq \left\lceil \sqrt{\frac{n^*(b-a)\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty}{4\varepsilon}} \right\rceil + 1$$

$$\leq \mathrm{cost}(A, f; \varepsilon) := \min\left\{ n : \frac{n^*(b-a)}{4(n-1)(n-1-n^*)} \widetilde{F}_n(f) \leq \varepsilon \right\}.$$

We know n should satisfy the condition $n > n^* + 1$, which means even if we approximate some easy functions with cone condition n^*, we still need $\left\lceil \frac{2n^*+1}{2} \right\rceil + 1$ points. Hence, we know

$$\max\left(\left\lceil \frac{2n^*+1}{2} \right\rceil, \left\lceil \sqrt{\frac{(b-a)^2 \left\|f''\right\|_\infty}{8\varepsilon}} \right\rceil \right) + 1$$

$$\leq \max\left(\left\lceil \frac{2n^*+1}{2} \right\rceil, \left\lceil \sqrt{\frac{n^*(b-a)\left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty}{4\varepsilon}} \right\rceil \right) + 1 \leq \mathrm{cost}(A, f; \varepsilon).$$

Let $n_1, n_2, \ldots$ be the sequence of n_i generated by Algorithm `funappxglobal_g`. We shall prove that the following statements must be true:

i) If the convergence criterion (2.10) is satisfied for i, then the algorithm stops, $A_{n_i}(f)$ is returned as the answer, and it meets the error tolerance.

ii) If the convergence criterion (2.10) is not satisfied for i, then n_{i+1} does not exceed the cost upper bounds in (2.12).

Statement i) holds because of the bounds in Lemma 1 and in Lemma 4.

If (2.10) is not satisfied for i, then it follows from the inequality in Corollary 1 that

$$n_i \le \left\lceil \sqrt{\frac{n^*(b-a)\widetilde{F}_{n_i}}{4\varepsilon}} + n^* + 1 \right\rceil \le \left\lceil \sqrt{\frac{n^*(b-a)\left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty}{4\varepsilon}} + n^* + 1 \right\rceil$$
$$\le \left\lceil \sqrt{\frac{n^*(b-a)^2 \left\| f'' \right\|_\infty}{8\varepsilon}} + n^* + 1 \right\rceil.$$

By (2.11) in Algorithm `funappxglobal_g`, we know if

$$\left\lceil \frac{1}{(n_i-1)} \sqrt{\frac{n^*(b-a)\widetilde{F}_{n_i}(f)}{4\varepsilon}} \right\rceil < 2,$$

then we obtain

$$n_{i+1} = 2n_i - 1 \le 2 \left\lceil \sqrt{\frac{n^*(b-a)\left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty}{4\varepsilon}} + n^* + 1 \right\rceil - 1$$
$$\le \left\lceil \sqrt{\frac{n^*(b-a)\left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty}{\varepsilon}} + 2n^* \right\rceil + 1,$$
$$n_{i+1} = 2n_i - 1 \le 2 \left\lceil \sqrt{\frac{n^*(b-a)^2 \left\| f'' \right\|_\infty}{8\varepsilon}} + n^* + 1 \right\rceil - 1$$
$$\le \left\lceil \sqrt{\frac{n^*(b-a)^2 \left\| f'' \right\|_\infty}{2\varepsilon}} + 2n^* \right\rceil + 1,$$

satisfied the upper bound on the cost (2.12).

Otherwise

$$n_{i+1} = \left\lceil \sqrt{\frac{n^*(b-a)\widetilde{F}_{n_i}(f)}{4\varepsilon}} \right\rceil + 1.$$

By Corollary 1, we obtain

$$n_{i+1} \le \left\lceil \sqrt{\frac{n^*(b-a)\left\| f' - \frac{f(b)-f(a)}{b-a} \right\|_\infty}{4\varepsilon}} \right\rceil + 1 \le \left\lceil \sqrt{\frac{n^*(b-a)^2 \left\| f'' \right\|_\infty}{8\varepsilon}} \right\rceil + 1.$$

The upper bound on the cost is obviously satisfied (2.12). □

Remark 5. The maximum and minimum costs of Algorithm `funappxglobal_g` in (2.12) depend on

$$\|f''\|_\infty \text{ and } \left\|f' - \frac{f(b)-f(a)}{b-a}\right\|_\infty$$

of the input function, f. However, these two semi-norms of f are not input to the algorithm, but rather are bounded by the algorithm. The sample size of Algorithm `funappxglobal_g` is adjusted adaptively based on these bounds.

Remark 6. Although adaptive Algorithm `funappxglobal_g` is defined only for proper subsets $\mathcal{C}_{n^*}$ of $\mathcal{W}^{2,\infty}$, they may actually be applied to all $f \in \mathcal{W}^{2,\infty}$ since the fixed-cost algorithms on which they are based are defined for all $f \in \mathcal{W}^{2,\infty}$. If the user unknowingly provides an input f that does not belong to $\mathcal{C}_{n^*}$ for Algorithm `funappxglobal_g`, the answer returned may be wrong because the corresponding Theorem 1 does not apply.

2.5.2 Lower bound on Complexity. Before discussing about the lower bound on complexity, we introduce some definitions. Firstly, we introduce the definition of a $\|f''\|_\infty$ ball of input functions, $\mathcal{B}_\sigma$.

$$\mathcal{B}_\sigma = \{f \in \mathcal{W}^{2,\infty} : \|f''\|_\infty \le \sigma\}, \tag{2.14}$$

where σ is a positive number. Let $\mathcal{N}$ be a subset of $\mathcal{W}^{2,\infty}$. We denote a set of algorithms $\mathcal{A}(\mathcal{N}, \mathcal{L}^\infty)$, where $\mathcal{A} : \mathcal{N} \to \mathcal{L}^\infty$ denotes all the automatic approximation algorithms. Next, we define the *maximum* costs of the algorithm $A \in \mathcal{A}(\mathcal{N}, \mathcal{L}^\infty)$ relative to $\mathcal{B}_s$ as follows:

$$\text{maxcost}(A, \mathcal{N}, \varepsilon, \mathcal{B}_s) = \sup\{\text{cost}(A, f, \varepsilon) : f \in \mathcal{N} \cap \mathcal{B}_s\}.$$

The complexity of univariate function approximation is defined as the maximum cost of the cheapest algorithm that always satisfies the error tolerance:

$$\operatorname{comp}(\varepsilon, \mathcal{A}(\mathcal{N}, \mathcal{L}^\infty), \mathcal{B}_s)$$
$$= \inf \{\operatorname{maxcost}(A, \mathcal{N}, \varepsilon, \mathcal{B}_s) : A \in \mathcal{A}(\mathcal{N}, \mathcal{L}^\infty),$$
$$\|f - A(f, \varepsilon)\|_\infty \leq \varepsilon \ \ \forall f \in \mathcal{N},\ \varepsilon \geq 0\} \in \mathbb{N}_0.$$

Lower complexity bounds are typically proved by constructing fooling functions. Here we first derive a lower bound for the complexity of function approximation defined on a ball of input functions, $\mathcal{B}_\sigma$. This technique is generally known; see for example [21, p. 11–12]. Then it is shown how to extend this idea for the cone $\mathcal{C}_{n^*}$.

For any $n \in \mathcal{J} := \mathbb{N}_0$, suppose that one has the sample points $(\xi_i)_{i=1}^n$ for arbitrary ξ_i, where $a = \xi_0 \leq \xi_1 < \cdots < \xi_n \leq \xi_{n+1} = b$. There must be some $j = 0, \ldots, n$ such that

$$\xi_{j+1} - \xi_j \geq \frac{b-a}{n+1}.$$

The function f_1 is defined as a bump having piecewise constant second derivative on $[\xi_j, \xi_{j+1}]$ and zero elsewhere. For $\xi_j \leq x \leq \xi_{j+1}$,

$$f_1(x) := \frac{1}{32}\left[4(\xi_{j+1} - \xi_j)^2 + (4x - 2\xi_j - 2\xi_{j+1})^2\right.$$
$$\left.+(4x - \xi_j - 3\xi_{j+1})\,|4x - \xi_j - 3\xi_{j+1}| - (4x - 3\xi_j - \xi_{j+1})\,|4x - 3\xi_j - \xi_{j+1}|\right], \quad (2.15)$$

$$f_1'(x) = \frac{1}{4}\left[4x - 2\xi_j - 2\xi_{j+1} + |4x - \xi_j - 3\xi_{j+1}| - |4x - 3\xi_j - \xi_{j+1}|\right],$$
$$f_1''(x) = \operatorname{sgn}(4x - \xi_j - 3\xi_{j+1}) - \operatorname{sgn}(4x - 3\xi_j - \xi_{j+1}) + 1.$$

For this bump function we know $\|f_1''\|_\infty = 1$, and by Corollary 1 we have

$$\frac{2}{b-a}\left\|f_1' - \frac{f_1(b) - f_1(a)}{b-a}\right\|_\infty \leq \|f_1''\|_\infty = 1. \quad (2.16)$$

Also we have

$$\|f_1\|_\infty = f_1((\xi_j + \xi_{j+1})/2) = \frac{(\xi_{j+1} - \xi_j)^2}{16} \geq \frac{(b-a)^2}{16(n+1)^2} =: g(n).$$

Theorem 2. *For $\sigma > 0$ let $\mathcal{B}_\sigma$ be defined in* (2.14)*. The complexity of function recovery on this ball is bounded below as*

$$\operatorname{comp}(\varepsilon, \mathcal{A}(\mathcal{B}_\sigma, \mathcal{L}^\infty), \mathcal{B}_s) \geq \left\lceil \sqrt{\frac{(b-a)^2 \min(s,\sigma)}{16\varepsilon}} \right\rceil .$$

Proof. Suppose that A is any successful automatic algorithm for the ball $\mathcal{B}_\sigma$, i.e., $A \in \mathcal{A}(\mathcal{B}_\sigma, \mathcal{L}^\infty)$, and $\|f - A(f)\|_\infty \leq \varepsilon$ for all $f \in \mathcal{B}_\sigma$. For any fixed $s \leq \sigma$ and $\varepsilon > 0$, let $(\xi_i)_{i=1}^n$ be the design used by A for the zero function, i.e., $f(\xi_i) = 0$ for any $i = 1, \ldots, n$. Thus, we obtain $\operatorname{cost}(A, f, \varepsilon) = n$. Let N_j and N_{j+1} be two successive elements of $\mathcal{J}$ with $N_j + 1 \leq \operatorname{cost}(A, f, \varepsilon) \leq N_{j+1}$.

Let f_1 be constructed according to (2.15) for this $(\xi_i)_{i=1}^n$. Since the data for the functions $\pm s f_1$ are all zero, it follows that $A(sf_1) = A(-sf_1)$. Also note that $\pm s f_1 \in \mathcal{B}_s$. Since A must be successful for $\pm s f_1$, it follows that

$$\begin{aligned}
\varepsilon &\geq \max(\|sf_1 - A(sf_1)\|_\infty, \|-sf_1 - A(-sf_1)\|_\infty) \\
&\geq \frac{1}{2}\left[\|sf_1 - A(sf_1)\|_\infty + \|sf_1 + A(sf_1)\|_\infty\right] \\
&\geq \frac{1}{2}\left\|[sf_1 - A(sf_1)] + [sf_1 + A(sf_1)]\right\|_\infty \\
&= \|sf_1\|_\infty = s\,\|f_1\|_\infty \\
&\geq sg(N_{j+1}).
\end{aligned}$$

Since $\pm s f_1 \in \mathcal{B}_s$, it follows that

$$N_j + 1 \leq \operatorname{cost}(A, \pm sf_1, \varepsilon) \leq \operatorname{maxcost}(A, \mathcal{B}_\sigma, \varepsilon, \mathcal{B}_s).$$

The inequality $g(N_{j+1}) \leq \varepsilon/s$ implies that N_j can be no smaller than the largest $n \in \mathcal{J}$ with $g(n) > \varepsilon/s$. Thus, $g^{-1}(\varepsilon/s) \leq N_j + 1 \leq \operatorname{maxcost}(A, \mathcal{B}_\sigma, \varepsilon, \mathcal{B}_s)$, where g^{-1} is defined by

$$g^{-1}(\varepsilon) = \max\{n \in \mathcal{J} : g(n) > \varepsilon\} + 1.$$

Here the maximum of the empty set is assumed to be -1. Therefore we have

$$\operatorname{comp}(\varepsilon, \mathcal{A}(\mathcal{B}_\sigma, \mathcal{L}^\infty), \mathcal{B}_s) \geq \left\lceil \sqrt{\frac{(b-a)^2 \min(s,\sigma)}{16\varepsilon}} \right\rceil .$$

□

To derive the lower complexity bound for a cone, we choose a parabola

$$f_0(x) = \frac{(x-a)(b-x)}{b-a}. \tag{2.17}$$

Then

$$\left\| f_0' - \frac{f_0(b)-f_0(a)}{b-a} \right\|_\infty = \sup_{a\leq x\leq b} \frac{|b+a-2x|}{b-a} = 1, \qquad \|f_0''\|_\infty = \frac{2}{b-a}. \tag{2.18}$$

Theorem 3. *For $n^* > 1$, the complexity of the function recovery problem over the cone of functions $\mathcal{C}_{n^*}$ defined in* (1.1) *is bounded below as*

$$\operatorname{comp}(\varepsilon, \mathcal{A}(\mathcal{C}_{n^*}, \mathcal{L}^\infty), \mathcal{B}_s) \geq \left\lceil \sqrt{\frac{(b-a)^2(n^*-1)s}{32n^*\varepsilon}} \right\rceil .$$

The adaptive linear spline Algorithm `funappxglobal_g` *has optimal order for recovering functions in $\mathcal{C}_{n^*}$.*

Proof. Let $A \in \mathcal{A}(\mathcal{C}_{n^*}, \mathcal{L}^\infty)$ be an arbitrary successful, possibly adaptive, algorithm. Given an error tolerance, ε, and a positive s, let f_0 be defined in (2.17), and choose

$$c_0 = \frac{s(n^*+1)(b-a)}{4n^*} > 0. \tag{2.19a}$$

Provide the algorithm A with the input function $c_0 f_0$, and let $(\xi_i)_{i=1}^n$ be the data vector to obtain the estimate $A(c_0 f_0)$. Let $\operatorname{cost}(A, c_0 f_0, \varepsilon)$ denote the cost of this algorithm for the function $c_0 f_0$, and let $N_j, N_{j+1} \in \mathcal{J}$ be chosen as before such that $N_j + 1 \leq \operatorname{cost}(A, c_0 f_0, \varepsilon) \leq N_{j+1}$. Define two fooling functions, $f_\pm = c_0 f_0 \pm c_1 f_1$, in terms of f_1 defined in (2.15) with

$$c_1 = \frac{s(n^*-1)}{2n^*} > 0. \tag{2.19b}$$

Then for both fooling functions, by (2.16) and (2.18) we have

$$\begin{aligned}\left\|f_{\pm}''\right\|_{\infty} &\le c_0 \left\|f_0''\right\|_{\infty} + c_1 \left\|f_1''\right\|_{\infty} \\ &= \frac{s(n^*+1)(b-a)}{4n^*} \cdot \frac{2}{b-a} + \frac{s(n^*-1)}{2n^*} \cdot 1 \\ &= s\left[\frac{n^*+1}{2n^*} + \frac{n^*-1}{2n^*}\right] = s \qquad \text{by (2.19).}\end{aligned}$$

Moreover, these fooling functions must lie inside the cone $\mathcal{C}_{n^*}$ because

$$\begin{aligned}&\left\|f_{\pm}''\right\|_{\infty} - \frac{2n^*}{b-a}\left\|f_{\pm}' - \frac{f_{\pm}(b)-f_{\pm}(a)}{b-a}\right\|_{\infty} \\ \le& s - \frac{2n^*}{b-a}\left(c_0\left\|f_0' - \frac{f_0(b)-f_0(a)}{b-a}\right\|_{\infty} - c_1\left\|f_1' - \frac{f_1(b)-f_1(a)}{b-a}\right\|_{\infty}\right) \\ &\qquad\qquad \text{by the triangle inequality} \\ \le& s - \frac{2n^*}{b-a}c_0 + n^*c_1 \qquad \text{by (2.16), (2.18)} \\ =& s - \frac{s(n^*+1)}{2} + \frac{s(n^*-1)}{2} = 0 \qquad \text{by (2.19).}\end{aligned}$$

We note that the data used by algorithm A for both fooling functions is the same, so $A(f_{\pm}) = A(c_0 f_0)$. Consequently,

$$\begin{aligned}\varepsilon &\ge \max\left(\left\|f_+ - A(f_+)\right\|_{\infty}, \left\|f_- - A(f_-)\right\|_{\infty}\right) \\ &\ge \frac{1}{2}\left[\left\|c_0f_0 + c_1f_1 - A(c_0f_0)\right\|_{\infty} + \left\|c_0f_0 - c_1f_1 - A(c_0f_0)\right\|_{\infty}\right] \\ &\ge \frac{1}{2}\left[\left\|c_0f_0 + c_1f_1 - A(c_0f_0)\right\|_{\infty} + \left\|c_1f_1 - c_0f_0 + A(c_0f_0)\right\|_{\infty}\right] \\ &\ge \frac{1}{2}\left\|\left[c_0f_0 + c_1f_1 - A(c_0f_0)\right] + \left[c_1f_1 - c_0f_0 + A(c_0f_0)\right]\right\|_{\infty} \\ &= \left\|c_1f_1\right\|_{\infty} = c_1\left\|f_1\right\|_{\infty} \\ &\ge c_1 g(N_{j+1}).\end{aligned}$$

Since A is successful for these two fooling functions and $f_{\pm} \in \mathcal{C}_{n^*}$, it follows that

$$\operatorname{cost}(A, f_{\pm}, \varepsilon) \le \operatorname{maxcost}(A, \mathcal{C}_{n^*}, \varepsilon, \mathcal{B}_s),$$

and the cost of this arbitrary algorithm A is bounded below by

$$g^{-1}\left(\frac{\varepsilon}{c_1}\right) = g^{-1}\left(\frac{2n^*\varepsilon}{s(n^*-1)}\right) = \left\lceil\sqrt{\frac{(b-a)^2(n^*-1)s}{32n^*\varepsilon}}\right\rceil.$$

This then implies the lower bound on the complexity of the problem. □

2.6 Our Cone is Non-Convex

In Chapter 1, we mentioned adaptive algorithms can have large advantages for the input functions on non-convex set. Here, we want to show our cone $\mathcal{C}_{n^*}$ is non-convex.

Let f_{in} and f_{out} be functions in $\mathcal{W}^{2,\infty}$ with nonzero weaker semi-norm, where f_{in} lies in the interior of this cone, and f_{out} lies outside the cone. This means that

$$\frac{\|f''_{\text{in}}\|_\infty}{\left\|f'_{\text{in}} - \frac{f_{\text{in}}(b)-f_{\text{in}}(a)}{b-a}\right\|_\infty} = 2n^*_{\text{in}} < 2n^* < 2n^*_{\text{out}} = \frac{\|f''_{\text{out}}\|_\infty}{\left\|f'_{\text{out}} - \frac{f_{\text{out}}(b)-f_{\text{out}}(a)}{b-a}\right\|_\infty}.$$

Next define two functions

$$f_\pm = 2(n^* - n^*_{\text{in}}) \left\|f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a}\right\|_\infty f_{\text{out}} \pm 2(n^* + n^*_{\text{out}}) \left\|f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a}\right\|_\infty f_{\text{in}}.$$

Since

$$\begin{aligned}
\left\|f''_\pm\right\|_\infty &\le 2(n^* - n^*_{\text{in}}) \left\|f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a}\right\|_\infty \|f''_{\text{out}}\|_\infty \\
&\quad + 2(n^* + n^*_{\text{out}}) \left\|f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a}\right\|_\infty \|f''_{\text{in}}\|_\infty \\
&= 4n^*_{\text{out}}(n^* - n^*_{\text{in}}) \left\|f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a}\right\|_\infty \left\|f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a}\right\|_\infty \\
&\quad + 4n^*_{\text{in}}(n^* + n^*_{\text{out}}) \left\|f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a}\right\|_\infty \left\|f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a}\right\|_\infty \\
&= 4n^*(n^*_{\text{out}} + n^*_{\text{in}}) \left\|f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a}\right\|_\infty \left\|f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a}\right\|_\infty,
\end{aligned}$$

and

$$\begin{aligned}
&\left\|f'_\pm - \frac{f_\pm(b) - f_\pm(a)}{b-a}\right\|_\infty \\
\ge &- 2(n^* - n^*_{\text{in}}) \left\|f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a}\right\|_\infty \left\|f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a}\right\|_\infty
\end{aligned}$$

$$+2(n^* + n^*_{\text{out}}) \left\| f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a} \right\|_\infty \left\| f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a} \right\|_\infty$$
$$=2(n^*_{\text{out}} + n^*_{\text{in}}) \left\| f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a} \right\|_\infty \left\| f'_{\text{out}} - \frac{f_{\text{out}}(b) - f_{\text{out}}(a)}{b-a} \right\|_\infty,$$

it follows that

$$\left\| f''_{\pm} \right\|_\infty \le 2n^* \left\| f'_{\pm} - \frac{f_{\pm}(b) - f_{\pm}(a)}{b-a} \right\|_\infty,$$

and so $f_{\pm} \in \mathcal{C}_{n^*}$. On the other hand $(f_- + f_+)/2$, which is a convex combination of f_+ and f_-, is

$$2(n^* - n^*_{\text{in}}) \left\| f'_{\text{in}} - \frac{f_{\text{in}}(b) - f_{\text{in}}(a)}{b-a} \right\|_\infty f_{\text{out}}.$$

Since $n^* > n^*_{\text{in}}$, this is a nonzero multiple of f_{out}, and it lies outside $\mathcal{C}_{n^*}$. Thus, this cone is not convex.

2.7 Numerical Examples

In this section, we present two numerical examples for `funappxglobal_g`. The first one is to show the success rate of `funappxglobal_g` if the input function is inside the cone or not. Moreover, we also compute the success rate of `chebfun`. The other one is to show the comparison between adaptive algorithms with and without guaranteed accuracy.

Example 1. This example is similar to the numerical example for univariate function approximation in [5]. To illustrate Algorithm `funappxglobal_g`, we consider the family of bump test functions defined on interval $[0, 1]$ by

$$f(x) = \begin{cases} b[4a^2 + (x-z)^2 + (x-z-a)|x-z-a| \\ \qquad -(x-z+a)|x-z+a|], & z-2a \le x \le z+2a, \\ 0, & \text{otherwise.} \end{cases} \quad (2.20)$$

with $\log_{10}(a) \sim \mathcal{U}[-4,-1]$, $z \sim \mathcal{U}[2a, 1-2a]$, and $b = 1/(2a^2)$. Since

$$\|f' - f(0) + f(1)\|_\infty = 1/a$$

and $\|f''\|_\infty = 1/a^2$, the probability that $f \in \mathcal{C}_{n^*}$ is

$$\min\left(1, \max(0, \left(\log_{10}(2n^*) - 1\right)/3)\right).$$

Table 2.1. The probability of the test function lying in the cone for the original values and eventual values of n^* and the empirical success rate of Algorithm `funappxglobal_g` and `chebfun`

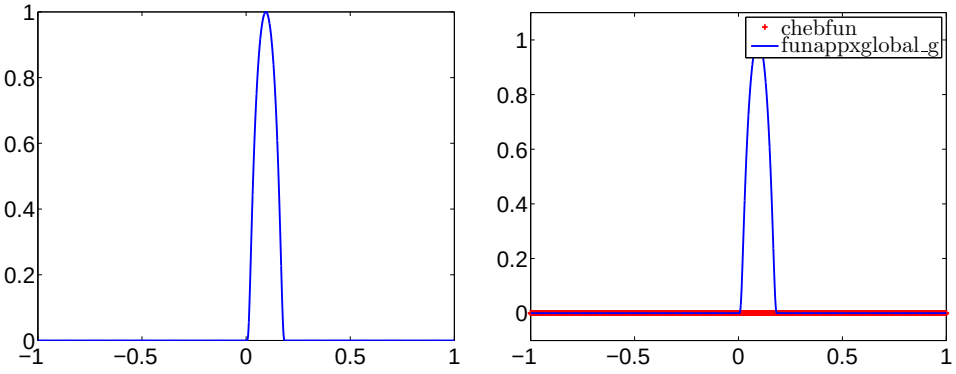
		Success	Success	Failure	Failure
n^*	Prob($f \in \mathcal{C}_{n^*}$)	No Warning	Warning	No Warning	Warning
5	$0\% \to 26\%$	26%	$< 1\%$	74%	$< 1\%$
50	$33\% \to 57\%$	56%	1%	43%	1%
500	$67\% \to 88\%$	75%	5%	12%	8%
`chebfun`		17%		83%	

As an experiment, we chose 10000 random test functions and applied Algorithm `funappxglobal_g` with an error tolerance of $\varepsilon = 10^{-8}$ and initial n^* values of 5, 50, 500. We also use `chebfun` to approximate the same test functions. The algorithm is considered successful for a particular f if the approximation $\hat{f}$ differs from f within ε. The success and failure rates are given in Table 2.1, which can be reproduced by `conepaper_test_funappx_g` in [3] within 1% difference.

Our algorithm imposes a cost budget of $N_{\max} = 10^7$. If the proposed n_{i+1} in Stages 2 or 3 exceeds $N_{\max}$, then our algorithm returns a warning and falls back to the largest possible n_{i+1} not exceeding $N_{\max}$ for which $n_{i+1} - 1$ is a multiple of $n_i - 1$. The probability that f initially lies in $\mathcal{C}_{n^*}$ is the smaller number in the second column of

Table 2.1, while the larger number is the empirical probability that f eventually lies in $\mathcal{C}_{n^*}$ after possible increases in n^* made by Stage 2 of Algorithm `funappxglobal_g`. For this experiment Algorithm `funappxglobal_g` was successful for all f that finally lie inside $\mathcal{C}_{n^*}$ and for which no attempt was made to exceed the cost budget.

But for `chebfun`, we find that the success rate is 17%, which is much smaller than the success rate of our algorithm. Even though `chebfun` is a higher order method, it cannot offer users a reliable answer.



a) b)

Figure 2.1. a) Graph of test function h; b) Graphs of approximate functions by `chebfun` and `funappxglobal_g`.

Table 2.2. Comparison between `chebfun` and `funappxglobal_g` for approximating function h

	`chebfun`	`funappxglobal_g`
number of points	1	$12,001$
approximation error	1	8.0929×10^{-6}

Example 2. In this example, we want to show Chebfun toolbox can be fooled by a simple function, even when all derivatives of the function exist. Test function h is a

piecewise function on $[-1, 1]$ with infinity smoothness.

$$h(x) = \begin{cases} \exp\left(1 - \frac{0.095^2}{x(0.19-x)}\right) & 0 \le x \le 0.19, \\ 0 & \text{else.} \end{cases}$$

If we use `chebfun` to approximate h, we get that the approximate function is zero. The approximation error and number of points needed of `chebfun` are listed in the second column of Table 2.2. When we use `funappxglobal_g` to approximate h with error tolerance $\varepsilon = 10^{-5}$, $n_{\text{lo}} = 10$, and $n_{\text{hi}} = 100$, the results are shown in the third column of Table 2.2. Obviously `funappxglobal_g` uses more points to approximate h than `chebfun` does. However, we can also find `funappxglobal_g` reaches the error tolerance 10^{-5}, while `chebfun` is fooled by h with the error 1.

Actually, any approximation algorithm can be fooled. Given a function f_0, a successful algorithm for f_0 will obtain an approximate function $\hat{f}_0$ at sample points $\{x_i\}_{i=0}^n$, where $\hat{f}_0(x_i) = f_0(x_i), i = 0, \ldots, n$, such that

$$\left\| f_0 - \hat{f}_0 \right\|_\infty \le \varepsilon.$$

We can construct a fooling function

$$f_1(x) = f_0(x) + b(x)p(x),$$

where $p(x) = (x-x_0)(x-x_1)\cdots(x-x_n)$ and $b(x)$ is any nonzero function of x. When we apply the same algorithm to f_1, it is easy to know $\hat{f}_1 = \hat{f}_0$ as the algorithm uses the same way to sample points. Thus, we can choose an appropriate $b(x)$ to make

$$\left\| f_1 - \hat{f}_1 \right\|_\infty > \varepsilon.$$

That's why Chebfun toolbox fails and does not give warnings of possible failure. For `funappxglobal_g`, it may also fail at this kind of situation. But we understand why the algorithm fails, it is because the fooling function f_1 does not lie in our cone $\mathcal{C}_{n^*}$. If we make the cone large enough to include f_1, our algorithm succeeds.

CHAPTER 3

GUARANTEED LOCALLY ADAPTIVE ALGORITHM FOR UNIVARIATE FUNCTION APPROXIMATION

3.1 Motivation

In the previous chapter, we have established a guaranteed (globally) adaptive algorithm. In this chapter, we want to develop a *locally adaptive* guaranteed algorithm for univariate function approximation. Our locally adaptive approach here is also data driven: we construct piecewise linear splines with different support sizes. In the subdomains where f is more peaked, manifested by large values of estimated $f''(x_i)$ where x_i's are our sampling points, we use more data points with smaller support. The globally adaptive version, in contrast, places evenly spaced knots in the entire domain; to achieve good accuracy in spiky parts of f, it often also puts in an unnecessarily large number of sample points in less challenging areas.

To introduce a locally adaptive algorithm, we firstly construct a partition on interval $[a, b]$. On each subinterval, we still use piecewise linear spline to approximate input functions but with fixed number of points. For error analysis, we use the same semi-norm and function space to work well as for the same as the global adaptive algorithm. We still want to analyze our algorithm for a cone of input functions. Instead of satisfying an inequality on the whole interval, the input functions need to satisfy inequalities on all subintervals of $[a, b]$. Based on the cone condition, we derive an error bound that holds on any subinterval. If the error estimates on all subintervals reach the error tolerance, our algorithm stops. If not, we iterate by splitting these subintervals that have not satisfied the error tolerance and double the points.

Similar to the globally adaptive algorithm, we have included implementation of our locally adaptive algorithm in the MATLAB Guaranteed Automatic Integration Library (GAIL) since 2014. A locally adaptive procedure `funappx_g` is in GAIL

versions 2.0 and after [2, 3].

3.2 Problem Setting on Partition

To establish the locally adaptive algorithm, instead of dealing with the whole interval $[a, b]$, we introduce a partition that takes the form of subintervals of $[a, b]$:

$$\mathcal{P} = \{[t_0, t_1], [t_1, t_2], \ldots, [t_{L-1}, t_L]\}, \qquad a = t_0 < t_1 < \cdots < t_L = b, \tag{3.1}$$

where $L = |\mathcal{P}|$ is the cardinality of $\mathcal{P}$, i.e., the number of subintervals. The t_i do not need to be evenly spaced.

We place n_0 evenly spaced nodes on each subinterval $[t_{i-1}, t_i]$, $i = 1, \ldots, L$:

$$x_{ij} = t_{i-1} + (t_i - t_{i-1})\frac{j-1}{n_0 - 1}, \qquad j = 1, \ldots, n_0 - 1. \tag{3.2}$$

Similar to the global adaptive algorithm, we use piecewise linear interpolation on each subinterval:

$$\begin{gathered}\tilde{A}(f; [t_{i-1}, t_i]) := \frac{n_0 - 1}{t_i - t_{i-1}} \left[f(x_{ij})(x_{i,j+1} - x) + f(x_{i,j+1})(x - x_{ij}) \right] \\ \forall x \in [x_{ij}, x_{i,j+1}] \qquad j = 1, \ldots, n_0 - 1.\end{gathered} \tag{3.3}$$

Define the algorithm on $\mathcal{P}$ to be $\hat{A}(f; \mathcal{P})$, where

$$\hat{A}(f; \mathcal{P}) = \begin{cases} \tilde{A}(f; [t_0, t_1]), & t_0 \le x \le t_1, \\ \vdots & \\ \tilde{A}(f; [t_{i-1}, t_i]), & t_{i-1} \le x \le t_i, \\ \vdots & \\ \tilde{A}(f; [t_{L-1}, t_L]), & t_{L-1} \le x \le t_L. \end{cases} \tag{3.4}$$

With the partition $\mathcal{P}$, and algorithm $\tilde{A}$, we can easily obtain the following corollary for the approximation error bound on each subinterval by Lemma 1.

Corollary 3. *For any function $f : [a, b] \to \mathbb{R}$ which has a second derivative, given a partition $\mathcal{P}$ defined in* (3.1) *and $\tilde{A}$ defined in* (3.3)*, it follows that*

$$\left\| f - \tilde{A}(f; [t_{i-1}, t_i]) \right\|_{[t_{i-1},t_i]} \leq \frac{(t_i - t_{i-1})^2}{8(n_0 - 1)^2} \left\| f'' \right\|_{[t_{i-1},t_i]},$$

where $\|f\|_{[t_{i-1},t_i]} := \|f\|_{\infty,[t_{i-1},t_i]} = \sup_{t_{i-1} \leq x \leq t_i} |f(x)|$.

To obtain an algorithm with guaranteed accuracy, here we still need the input function to have at least a second derivative. We still work on the Sobolev space $\mathcal{W}^{2,\infty}$ defined in (2.3). Our goal is, given a "nice" subset of input functions, $\mathcal{N} \subseteq \mathcal{W}^{2,\infty}$, we want to find an algorithm $A(f, \varepsilon)$ such that

$$\|f - A(f, \varepsilon)\|_\infty \leq \varepsilon, \qquad \forall f \in \mathcal{N},$$

where $A(f, \varepsilon) = \hat{A}(f; \mathcal{P})$ and $\mathcal{P}$ is determined by f and ε.

To make our algorithm adaptive, we use the same idea as in the global adaption algorithm. Rather than assuming an upper bound on $\|f''\|_{[t_{i-1},t_i]}$ for each subinterval, we use function data to construct rigorous upper bounds on $\|f''\|_{[t_{i-1},t_i]}$. Thus we introduce a similar weaker semi-norm

$$\begin{aligned}\left\| f' - \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}} \right\|_{[t_{i-1},t_i]} &:= \left\| f' - \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}} \right\|_{\infty,[t_{i-1},t_i]} \\ &= \sup_{t_{i-1} \leq x \leq t_i} \left| f'(x) - \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}} \right|.\end{aligned}$$

With this weaker norm, we can introduce a "nice" cone

$$\begin{aligned}\mathcal{C}_{\boldsymbol{n}^*} := \big\{ & f \in \mathcal{W}^{2,\infty} : \\ & \|f''\|_{[t_{i-1},t_i]} \leq \frac{2n_i^*}{t_i - t_{i-1}} \left\| f' - \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}} \right\|_{[t_{i-1},t_i]}, \forall i = 1, \ldots, |\mathcal{P}|, \forall \mathcal{P} \Bigg\}, \end{aligned} \tag{3.5}$$

where

$$\boldsymbol{n}^* = (n_i^*)_{i=1,\ldots,|\mathcal{P}|}, \qquad n_i^* = \eta(t_i - t_{i-1}), \tag{3.6}$$

η is defined in (2.4). Now our goal is to find an algorithm $A(f,\varepsilon)$ such that

$$\left\|f - A(f,\varepsilon)\right\|_\infty \le \varepsilon, \qquad \forall f \in \mathcal{C}_{\boldsymbol{n}^*}.$$

3.3 Error Estimation on Subinterval $[t_{i-1}, t_i]$

By the definition of algorithm $A(f,\varepsilon)$ and definition of $\hat{A}(f;\mathcal{P})$ in (3.4), we know

$$\left\|f - A(f,\varepsilon)\right\|_\infty = \left\|f - \hat{A}(f;\mathcal{P})\right\|_\infty = \max_{i=1,\ldots,L} \left\|f - \tilde{A}(f;[t_{i-1},t_i])\right\|_{[t_{i-1},t_i]},$$

where $L = |\mathcal{P}|$. If

$$\left\|f - \tilde{A}(f;[t_{i-1},t_i])\right\|_{[t_{i-1},t_i]} \le \varepsilon, \qquad \forall i = 1,\ldots,L,$$

is satisfied, our algorithm reaches our goal. Therefore, in this section, we investigate the error analysis on subinterval $[t_{i-1}, t_i]$, $i = 1,\ldots,L$.

We denote

$$\widetilde{F}(f;[t_{i-1},t_i]) := \sup_{j=1,\ldots,n_0-1} \left| \frac{n_0-1}{t_i - t_{i-1}}[f(x_{i,j+1}) - f(x_{ij})] - \frac{f(t_i)-f(t_{i-1})}{t - t_{i-1}} \right| \tag{3.7}$$

to approximate the weaker semi-norm,

$$\left\| f' - \frac{f(t_i)-f(t_{i-1})}{t_i - t_{i-1}} \right\|_{[t_{i-1},t_i]},$$

by the data information $(x_{ij}, f(x_{ij})), j = 1,\ldots,n_0-1$. Let $a = t_{i-1}, b = t_i, n = n_0$ in Lemma 2 and Corollary 1. For all $f \in \mathcal{C}_{\boldsymbol{n}^*}$, if $n_0 > 1 + n_i^*$, we can obtain the following inequalities:

$$\|f''\|_{[t_{i-1},t_i]} \le \frac{2n_i^*}{(t_i - t_{i-1})(1 - n_i^*/(n_0-1))} \widetilde{F}(f;[t_{i-1},t_i]), \tag{3.8}$$

$$\widetilde{F}(f;[t_{i-1},t_i]) \le \left\| f' - \frac{f(t_i)-f(t_{i-1})}{t_i - t_{i-1}} \right\|_{[t_{i-1},t_i]} \le \frac{(t_i - t_{i-1})}{2} \|f''\|_{[t_{i-1},t_i]}. \tag{3.9}$$

By (3.8) and Corollary 3, we can obtain the upper bound on the approximation error in the following lemma:

Lemma 5. *Given a partition $\mathcal{P}$ defined in* (3.1)*, let $\tilde{A}$ and $\widetilde{F}$ be defined in* (3.4) *and* (3.7)*. $\forall f \in \mathcal{C}_{\boldsymbol{n}^*}$, if $n_0 > n_i^* + 1$, we have*

$$\left\| f - \tilde{A}(f;[t_{i-1},t_i]) \right\|_{[t_{i-1},t_i]} \le \frac{n_i^*(t_i - t_{i-1})}{4(n_0-1)(n_0-1-n_i^*)} \widetilde{F}(f;[t_{i-1},t_i]). \tag{3.10}$$

From Lemma 5, it is easy to know the stopping criterion of the locally adaptive algorithm is

$$\frac{n_i^*(t_i - t_{i-1})}{4(n_0-1)(n_0-1-n_i^*)} \widetilde{F}(f;[t_{i-1},t_i]) \le \varepsilon, \qquad \forall i = 1,\ldots,L.$$

Similar to the globally adaptive algorithm, we can use the same data to approximate the stronger semi-norm $\|f''\|_{[t_{i-1},t_i]}$ by centered difference. For $n_0 \ge 3$,

$$F(f;[t_{i-1},t_i]) := \frac{(n_0-1)^2}{(t_i - t_{i-1})^2} \sup_{j=1,\ldots,n_0-2} \left| f(x_{ij}) - 2f(x_{i,j+1}) + f(x_{ij+2}) \right|. \tag{3.11}$$

It is easy to find $F(f;[t_{i-1},t_i])$ never overestimates $\|f''\|_{[t_{i-1},t_i]}$, i.e.,

$$F(f;[t_{i-1},t_i]) \le \|f''\|_{[t_{i-1},t_i]}.$$

Similarly, let $a = t_{i-1}, b = t_i, n = n_0$ in Corollary 2, we have the necessary condition for locally adaptive algorithm:

$$n_{i,\min}^* := \frac{F(f;[t_{i-1},t_i])}{2\widetilde{F}(f;[t_{i-1},t_i])/(t_i - t_{i-1}) + F(f;[t_{i-1},t_i])/(n_0-1)} \le n_i^*, \ \forall i = 1,\ldots,L.$$

3.4 Locally Adaptive Algorithm `funappx_g`

Algorithm `funappx_g`. (Locally adaptive univariate function recovery) Set

$$\begin{aligned} t_0 &= \text{ left end point of the interval } = a, \\ L &= \text{ the number of subintervals } = 1, \\ t_L &= \text{ right end point of the interval } = b, \\ \mathcal{P} &= \text{ the partition of } [a,b] = \{[t_0,t_1]\}, \end{aligned}$$

$K =$ the set of indices to the subintervals

which the function is not yet approximated well enough $= \{1\}$,

$\boldsymbol{n^*} =$ the vector of cone constants $= (n_1^*) = (\eta(t_1 - t_0))$,

where η is defined in Remark 1. Define

$$\widetilde{F}(\cdot;\mathcal{P}) = \{\widetilde{F}(\cdot;[t_{i-1},t_i])\}_{[t_{i-1},t_i]\in\mathcal{P}}, \qquad F(\cdot;\mathcal{P}) = \{F(\cdot;[t_{i-1},t_i])\}_{[t_{i-1},t_i]\in\mathcal{P}}.$$

Let the sequence of algorithms $\hat{A}(\cdot;\mathcal{P})$ be described in (3.4). Let

$$n_0 = \text{ the number of points per subinterval } = 2n_1^* + 1. \tag{3.12}$$

For any error tolerance ε and input function f, do the following:

Stage 1. Estimate $\left\| f' - \frac{f(t_k)-f(t_{k-1})}{t_k - t_{k-1}} \right\|_{[t_{k-1},t_k]}$ and bound $\|f''\|_{[t_{k-1},t_k]}$. For all $k \in K$, compute $\widetilde{F}(f;[t_{k-1},t_k])$ in (3.7) and $F(f;[t_{k-1},t_k])$ in (3.11).

Stage 2. Check the necessary condition for $f \in \mathcal{C}_{\boldsymbol{n}^*}$. For all $k \in K$, compute

$$n_{\min,k}^* = \frac{F(f;[t_{k-1},t_k])}{2\widetilde{F}(f;[t_{k-1},t_k])/(t_k - t_{k-1}) + F(f;[t_{k-1},t_k])/(n_0 - 1)}.$$

If $n_k^* \geq n_{\min,k}^*$, then go to stage 3. Otherwise, set

$$n_k^* = \frac{2n_0 - 2 - n_{\min,k}^*}{n_0 - 1} n_{\min,k}^*,$$

go to Stage 3.

Stage 3. Check for convergence. For all $k \in K$, estimate errest_k.

$$\text{errest}_k = \frac{n_k^*(t_k - t_{k-1})\widetilde{F}(f;[t_{k-1},t_k])}{4(n_0 - 1)(n_0 - 1 - n_k^*)}. \tag{3.13}$$

Let $K = \{k | \text{errest}_k > \varepsilon\}$. If $K \neq \emptyset$, go to Stage 4. Otherwise, return $A(f,\varepsilon) = \hat{A}(f;\mathcal{P})$ and terminate.

Stage 4. Double the number of points and split intervals in K. Suppose

$$\mathcal{P} = \{[t_0, t_1], \ldots, [t_{k-1}, t_k], \ldots, [t_{L-1}, t_L]\}$$

and $K = \{k_i\}_{i=1}^m$, where $k_0 = 0 < k_1 < k_2 < \ldots < k_m \leq L = k_{m+1}$. Double the number of points on interval $[t_{k_i-1}, t_{k_i}]$ and split it into two, i.e.

$$[t_{k_i-1}, t_{k_i}] \longrightarrow \left[t_{k_i-1}, \frac{t_{k_i-1} + t_{k_i}}{2}\right] \left[\frac{t_{k_i-1} + t_{k_i}}{2}, t_{k_i}\right] \qquad \forall i = 1, \ldots, m,$$

where the k_i here are the present values. To update the endpoints of subintervals and partition, we have the following formulas. For clarification, we use primed notation for the updated ones.

$$\begin{aligned}
&L' = L + m,\\
&m' = 2m,\\
&k'_{2i} = k_i + i, \qquad i = 0, \ldots, m,\\
&k'_{2i+1} = k_{i+1} + i, \qquad i = 0, \ldots, m,\\
&t'_{k+i} = t_k, \qquad k_i \leq k < k_{i+1},\ i = 0, \ldots, m,\\
&t'_{k+i-1} = \frac{t_{k_i-1} + t_{k_i}}{2}, \qquad i = 1, \ldots, m,\\
&t'_{L'} = t_L,\\
&K' = \{k'_1, \ldots, k'_{2m}\},\\
&\boldsymbol{n^{*\prime}} = (n^{*\prime}{}_k)_{k=1}^{L'} = (\eta(t'_k - t'_{k-1}))_{k=1}^{L'},\\
&\mathcal{P}' = \{[t'_0, t'_1], \ldots, [t'_{k-1}, t'_k], \ldots, [t'_{L'-1}, t'_{L'}]\}.
\end{aligned}$$

Then go back to Stage 1.

Remark 7. Different from `funappxglobal_g`, `funappx_g` needs to generate the cone constants at each iteration. But we use the same non-decreasing function η to determine the cone constant $\boldsymbol{n^*} = (n_k^*)_{k=1}^{|\mathcal{P}|}$. While the length of subinterval is larger, input functions on that subinterval need to satisfy the inequality with a larger cone constant n_k^*.

Remark 8. Using the same idea as in `funappxglobal_g`, for practical reasons one may impose a computational cost budget, $N_{\max}$. In addition to the cost budget, one may also input a maximum number of iterations, maxiter. `funappx_g` computes the correct answer within budget for $f \in \mathcal{C}_{\boldsymbol{n}^*}$ provided either of the cost upper bounds in Theorem 4 does not exceed $N_{\max}$ and within the number of iterations less or equal to maxiter.

Remark 9. In `funappxglobal_g`, we only compute new added points in each step by using the information from the previous iterations. Thus, when we implement `funappx_g` in GAIL [3], we also want to compute function values only once. In addition, we want to simultaneously split all the intervals, which do not satisfy error tolerance. The implementation is more complicated in `funappx_g` than in `funappxglobal_g`. Therefore we present more details to show how to implement in MATLAB and how to update K and $\mathcal{P}$.

3.4.1 Implementation in MATLAB of `funappx_g`. In what follows we use MATLAB syntax to explain the implementation of `funappx_g`. Suppose we approximate a function on the interval $[0, 1]$ with 10 initial points, i.e. $a = 0$, $b = 1$, $n_0 = 10$. Suppose after three iterations, the partition becomes

$$\mathcal{P} = \left\{ \left[0, \frac{1}{8}\right], \left[\frac{1}{8}, \frac{1}{4}\right], \left[\frac{1}{4}, \frac{1}{2}\right], \left[\frac{1}{2}, \frac{5}{8}\right], \left[\frac{5}{8}, \frac{3}{4}\right], \left[\frac{3}{4}, 1\right] \right\}.$$

Let `x` denote all of the sample points of the $\mathcal{P}$ and a vector `index` represent the indices of endpoints of all the subintervals. We have

$$\texttt{index} = \begin{pmatrix} 1 & 10 & 19 & 28 & 37 & 46 & 55 \end{pmatrix} \qquad \texttt{x(index)} = \begin{pmatrix} 0 & \frac{1}{8} & \frac{1}{4} & \frac{1}{2} & \frac{5}{8} & \frac{3}{4} & 1 \end{pmatrix},$$

where `x(i)` denote the i-th element of `x`.

In MATLAB, we use a vector `errindex` to keep track of whether the error estimation of each subinterval greater than error tolerance or not. Based on the

partition, as $\left[\frac{1}{4}, \frac{1}{2}\right]$ and $\left[\frac{3}{4}, 1\right]$ are not split at the third iteration, we can assume

$$\texttt{errindex} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \end{pmatrix},$$

where 1 denotes that the error estimation of the subinterval is greater than error tolerance, 0 otherwise. Thus we know that

$$K = \texttt{badindex} = \begin{pmatrix} 1 & 2 & 4 & 5 \end{pmatrix}, \qquad \texttt{goodindex} = \begin{pmatrix} 3 & 6 \end{pmatrix},$$

where `badindex` denotes the indices of the subintervals which need to be split and `goodindex` denotes the indices of the subintervals satisfied error tolerance. Before we introduce how to add new `x` value back to original `x`, we firstly want to illustrate how to update K and `goodindex`.

To update `badindex` and `goodindex`, the key idea is to figure out how many subintervals before this subinterval need to be split. We update `goodindex` at first. As we compute the cumulative sum of `errindex`, we obtain

$$\texttt{cumerrindex} = \begin{pmatrix} 1 & 2 & 2 & 3 & 4 & 4 \end{pmatrix}.$$

Next we let

$$\begin{aligned} \texttt{newgoodindex} &= \texttt{goodindex} + \texttt{cumerrindex}(\texttt{goodindex}) \\ &= \begin{pmatrix} 3 & 6 \end{pmatrix} + \begin{pmatrix} 2 & 4 \end{pmatrix} = \begin{pmatrix} 5 & 10 \end{pmatrix}. \end{aligned}$$

We obtain `newgoodindex` to denote the updated indices of the subintervals satisfying error tolerance. To update K, similar to updating `goodindex`, we still need to use `cumerrindex`. The difference is we not only update the indices in `badindex`, but also generate new indices of new split subintervals and insert them back in `badindex`. To realize it, we add `cumerrindex(badindex)` $-\ 1$ and `cumerrindex(badindex)` to `badindex` respectively and sort them to be `newbadindex`.

$$K \longrightarrow \begin{pmatrix} \texttt{badindex} \\ \\ \texttt{badindex} \end{pmatrix} + \begin{pmatrix} \texttt{cumerrindex}(\texttt{badindex}) - 1 \\ \\ \texttt{cumerrindex}(\texttt{badindex}) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 4 & 5 \\ \\ 1 & 2 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 2 & 3 \\ \\ 1 & 2 & 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 6 & 8 \\ \\ 2 & 4 & 7 & 9 \end{pmatrix}$$

$$\Longrightarrow K = \texttt{newbadindex} = \begin{pmatrix} 1 & 2 & 3 & 4 & 6 & 7 & 8 & 9 \end{pmatrix}.$$

Next we deal with generating new `x`. Firstly, we want to have the step size of each subinterval to generate new points, denoted as `h`. Based on the algorithm, `h` is related to a, b, the number of iterations, and initial number of points n_0. Thus, we have $\texttt{h} = \frac{b-a}{2^{3+1}(n_0-1)} = \frac{1}{144}$. Next, we reshape `x` to a matrix, which is easy for us to use with `goodindex` and `badindex`. If we exclude the last point of `x`, `x` is a vector with size 1×54. Hence, we can reshape it to a matrix of 9×6, named `reshapex`, where each column is the points at the corresponding subintervals without right endpoints.

$$\texttt{x(1:end-1)} = \begin{pmatrix} 0 \\ \\ \frac{1}{72} \\ \\ \vdots \\ \\ \frac{17}{18} \\ \\ \frac{35}{36} \end{pmatrix} \longrightarrow \texttt{reshapex} = \begin{pmatrix} 0 & \frac{1}{8} & \frac{1}{4} & \frac{1}{2} & \frac{5}{8} & \frac{3}{4} \\ \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \\ \frac{1}{9} & \frac{17}{72} & \frac{17}{36} & \frac{13}{18} & \frac{53}{72} & \frac{35}{36} \end{pmatrix}.$$

Thus, new generated points are

$$\begin{aligned} \texttt{newx} &= \texttt{reshapex(:,badindex)} + \texttt{h} \\ &= \begin{pmatrix} 0 & \frac{1}{8} & \frac{1}{2} & \frac{5}{8} \\ \\ \vdots & \vdots & \vdots & \vdots \\ \\ \frac{1}{9} & \frac{17}{72} & \frac{11}{18} & \frac{53}{72} \end{pmatrix} + \frac{1}{144} = \begin{pmatrix} \frac{1}{144} & \frac{19}{144} & \frac{73}{144} & \frac{91}{144} \\ \\ \vdots & \vdots & \vdots & \vdots \\ \\ \frac{17}{144} & \frac{35}{144} & \frac{89}{144} & \frac{107}{144} \end{pmatrix}, \end{aligned}$$

where $\texttt{reshapex}(:, j)$ represents j-th column of matrix `reshapex`. Let `y` denote the function values of sample points . After we obtain `newx`, we can use $\texttt{newy} = f(\texttt{newx})$

to obtain `newy`. Thus, we don't need to recompute the other `y` again. How to insert `newy` back into `y` is exactly the same as how to insert `newx` back into `x`. Next, we explain the details how to deal with this problem.

As we know the columns of `newx` are not the points in the corresponding subinterval. We must insert them back into `reshapex(:,badindex)`. To achieve it, we initialize a zero matrix, called `badx`, with $2(n_0-1) = 18$ rows and `length(badindex)` $= 4$ columns at first. Then we assign `reshapex(:, badindex)` and `newx` into the matrix with a special order. And after we reshape the matrix, we can obtain a matrix, of which columns are the points corresponding to `newbadindex`.

$$\texttt{badx} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \end{pmatrix}_{18\times 4}$$

$$\xrightarrow[\texttt{badx(2:2:end,:)=newx}]{\texttt{badx(1:2:end-1,:)=reshapex(:,badindex)}} \begin{pmatrix} 0 & \frac{1}{8} & \frac{1}{2} & \frac{5}{8} \\ \frac{1}{144} & \frac{19}{144} & \frac{73}{144} & \frac{91}{144} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{9} & \frac{17}{72} & \frac{11}{18} & \frac{53}{72} \\ \frac{17}{144} & \frac{35}{144} & \frac{89}{144} & \frac{107}{144} \end{pmatrix}_{18\times 4}$$

$$\xrightarrow{\text{reshape it to a matrix with size } 9\times 8} \begin{pmatrix} 0 & \frac{1}{16} & \frac{1}{8} & \frac{3}{16} & \frac{1}{2} & \frac{9}{16} & \frac{5}{8} & \frac{11}{16} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{1}{18} & \frac{17}{144} & \frac{13}{72} & \frac{35}{144} & \frac{5}{9} & \frac{89}{144} & \frac{49}{72} & \frac{107}{144} \end{pmatrix}_{9\times 8} = \texttt{badx}.$$

Use the same idea, we initialize a zero matrix `newreshapex` with $n_0 - 1 = 9$ rows and `2*length(badindex)` + `length(goodindex)` $= 10$ columns.

$$\texttt{newreshapex} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}_{9\times 10}$$

$$\xrightarrow[\texttt{newreshapex(:,newbadindex)]=badx}]{\texttt{newreshapex(:,newgoodindex)=reshapex(:,goodindex)}}$$

$$\begin{pmatrix} 0 & \frac{1}{16} & \frac{1}{8} & \frac{3}{16} & \frac{1}{4} & \frac{1}{2} & \frac{9}{16} & \frac{5}{8} & \frac{11}{16} & \frac{3}{4} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{8}{144} & \frac{17}{144} & \frac{13}{72} & \frac{35}{144} & \frac{17}{36} & \frac{5}{9} & \frac{89}{144} & \frac{49}{72} & \frac{107}{144} & \frac{35}{36} \end{pmatrix}_{9\times 10}$$

$$\xrightarrow{\text{reshape to a } 90\times 1\text{vector}} \begin{pmatrix} 0 \\ \vdots \\ \frac{35}{36} \end{pmatrix}_{90\times 1}$$

After we add back the end point of `x`, we obtain new `x`. Use the same idea, we can update `y` easily.

Updating `index` is easier than updating `x`. As we used fixed number of points in each subinterval, the key point to obtain update `index` is to know how many subintervals we have. Thus, the indices of endpoints of each subinterval is just $k(n_0 - 1) + 1$, where $k = 1, \ldots, |\mathcal{P}|$ is the index of the subinterval.

3.5 Computational Cost

In this section, we investigate the computational cost of our locally adaptive algorithm. We firstly introduce one new notation $I_{x,l}$. Let

$$I_{x,l} = \text{unique interval containing } x \text{ whose width is } 2^{-l}(b-a),$$

where $l \in \mathbb{N}_0$. If x is the end point of a subinterval, we have

$$I_{x,l} = \begin{cases} [t_k, t_{k+1}) & x = t_k,\ k = 1, \ldots, |\mathcal{P}| - 1 \\ [t_{L-1}, t_L] & x = t_{|\mathcal{P}|}. \end{cases}$$

Note that

$$l \le l' \implies I_{x,l} \supseteq I_{x,l'}.$$

Also note that

$$y \in I_{x,l} \iff I_{x,l} = I_{y,l}.$$

And if $I_{x,l} = [t_k,\ t_{k+1})$, then we denote corresponding n_k^* by n_l^* for $I_{x,l}$, as

$$n_l^* = \left\lceil n_{\text{hi}} \left(\frac{n_{\text{lo}}}{n_{\text{hi}}} \right)^{\frac{1}{1+2^{-l}(b-a)}} \right\rceil.$$

Thus, for subinterval $I_{x,l}$, we define the error bound from (3.13)

$$\text{errest}(x, l) = \frac{n_l^* 2^{-l} (b-a) \widetilde{F}(f; I_{x,l})}{4(n_0 - 1)(n_0 - 1 - n_l^*)}.$$

Denote

$$\ell(x) = \min\{l \in \mathbb{N}_0 : \text{errest}(x, l) \le \varepsilon\}. \tag{3.14}$$

By (3.9), we can obtain an upper bound on $\text{errest}(x, l)$ in terms of $\|f''\|_{I_{x,l}}$:

$$\text{errest}(x, l) \le C_l \|f''\|_{I_{x,l}}, \qquad \forall x \in [a, b],\ l \in \mathbb{N}_0,$$

where

$$C_l = \frac{n_l^* 2^{-2l} (b-a)^2}{8(n_0 - 1)(n_0 - 1 - n_l^*)}$$

depends on n_0, n_l^*, and l, but not on f. Thus we can just define

$$L(x) = \min \left\{ l \in \mathbb{N}_0 : C_l \|f''\|_{I_{x,l}} \le \varepsilon \right\}. \tag{3.15}$$

It is easy to obtain $\ell(x) \le L(x)$. Next we want to investigate $2^{L(x)}$.

$$2^{L(x)} = 2^{\min\left\{ l \in \mathbb{N}_0 : C_l \|f''\|_{I_{x,l}} \le \varepsilon \right\}} = \min \left\{ 2^l : l \in \mathbb{N}_0, C_l \|f''\|_{I_{x,l}} \le \varepsilon \right\}.$$

Denote

$$M(x) = \min\left\{2^l : \frac{n_l^* 2^{-2l}(b-a)^2}{8(n_0-1)(n_0-1-n_l^*)} \left\|f''\right\|_{I_{x,l}} \le \varepsilon, l \in \mathbb{N}_0\right\}$$
$$= \min\left\{2^l : 2^l \ge \sqrt{\frac{n_l^*(b-a)^2 \left\|f''\right\|_{I_{x,l}}}{8(n_0-1)(n_0-1-n_l^*)\varepsilon}}, l \in \mathbb{N}_0\right\}. \tag{3.16}$$

We obtain the upper bound on computational cost of Algorithm `funappx_g`.

Theorem 4. *Let $A(f,\varepsilon)$ be the adaptive linear spline defined by Algorithm* `funappx_g`*, and let n_0, $\boldsymbol{n}^*$, and ε be the inputs and parameters described there. Let $\mathcal{C}_{\boldsymbol{n}^*}$ be the cone of functions defined in* (3.5)*. Let $M(x)$ is defined in* (3.16)*. Then it follows that Algorithm* `funappx_g` *is successful for all functions in $\mathcal{C}_{\boldsymbol{n}^*}$, i.e., $\left\|f - A(f,\varepsilon)\right\|_\infty \le \varepsilon$. Moreover, the cost of this algorithm is bounded above as follows:*

$$\mathrm{cost}(A, f; \mathcal{P}, \varepsilon) \le \frac{n_0 - 1}{b-a}\int_a^b M(x)dx + 1.$$

Proof. For subinterval $I_{x,\ell(x)}$, where $\ell(x)$ is defined in (3.14), which means Algorithm `funappx_g` satisfied error tolerance on $I_{x,\ell(x)}$ with n_0 number of points. If we take out the right end point of the subinterval, then the density of the cost on $I_{x,\ell(x)}$ can be considered as

$$\frac{n_0 - 1}{2^{-\ell(x)}(b-a)}.$$

Thus, we obtain the cost on whole interval $[a,b]$ should be

$$\mathrm{cost}(A, f; \mathcal{P}, \varepsilon) = \int_a^b \frac{n_0 - 1}{2^{-\ell(x)}(b-a)}dx + 1 = \frac{n_0 - 1}{(b-a)}\int_a^b 2^{\ell(x)}dx + 1.$$

If $L(x)$ is defined in (3.15), then we know $\ell(x) \le L(x)$. Therefore we obtain an upper bound

$$\mathrm{cost}(A, f; \mathcal{P}, \varepsilon) \le \frac{n_0 - 1}{(b-a)}\int_a^b 2^{L(x)}dx + 1.$$

As $M(x) = 2^{L(x)}$ by the definition (3.16), we obtain

$$\mathrm{cost}(A, f; \mathcal{P}, \varepsilon) \le \frac{n_0 - 1}{(b-a)}\int_a^b M(x)dx + 1.$$

□

From Theorem 4, we know for very small ε, l tends to ∞, $\|f''\|_{I_{x,l}}$ tends to $|f''(x)|$ and n_l^* tends to n_{lo}. Thus we can have

$$M(x) \lesssim 2\sqrt{\frac{n_{\text{lo}}(b-a)^2|f''(x)|}{8(n_0-1)(n_0-1-n_{\text{lo}})\varepsilon}}.$$

The upper bound on computational cost tends to

$$\begin{aligned}
\text{cost}(A, f; \mathcal{P}, \varepsilon) &\lesssim \frac{n_0-1}{(b-a)} \int_a^b 2\sqrt{\frac{n_{\text{lo}}(b-a)^2|f''(x)|}{8(n_0-1)(n_0-1-n_{\text{lo}})\varepsilon}} dx + 1 \\
&= \sqrt{\frac{n_{\text{lo}}(n_0-1)}{2\varepsilon(n_0-1-n_{\text{lo}})}} \int_a^b \sqrt{|f''(x)|} dx + 1 \\
&= \sqrt{\frac{n_{\text{lo}}(n_0-1)}{2\varepsilon(n_0-1-n_{\text{lo}})}} \left\| \sqrt{|f''|} \right\|_1 dx + 1,
\end{aligned}$$

where $\|g\|_1 = \int_a^b |g''(x)|dx$. However, when ε is not that small, the computational cost does not only depend on $\int_a^b |f''(x)|dx$ but also depends on where and how the peaky parts are located.

To illustrate the computational cost more clearly, we present the following example. We firstly construct a piecewise test function g on $[0, 1]$:

$$g(x) = \begin{cases} x + 8.85 & \text{if } x \in [0, 0.1) \\ 9 - 5(x - 0.2)^2 & \text{if } x \in [0.1, 0.3) \\ -x + 9.25 & \text{if } x \in [0.3, 1] \end{cases}$$

Figure 3.1a) shows the graph of $g(x)$. We also construct two parabolas p_1 and p_2:

$$p_1(x) = -(x - 0.5)^2 + 25 \qquad p_2(x) = -5(x - 0.5)^2 + 25,$$

as shown in Figure 3.1b) and Figure 3.1c). From Figure 3.1d) to Figure 3.1f), we can obtain

$$\|p_1''\|_1 = \|g''\|_1 \qquad \|p_2''\|_\infty = \|g''\|_\infty .$$

Using `funappx_g` to approximate these test functions with different error tolerance, we obtain Table 3.1. We compare g and p_1 at first. From Table 3.1, we can

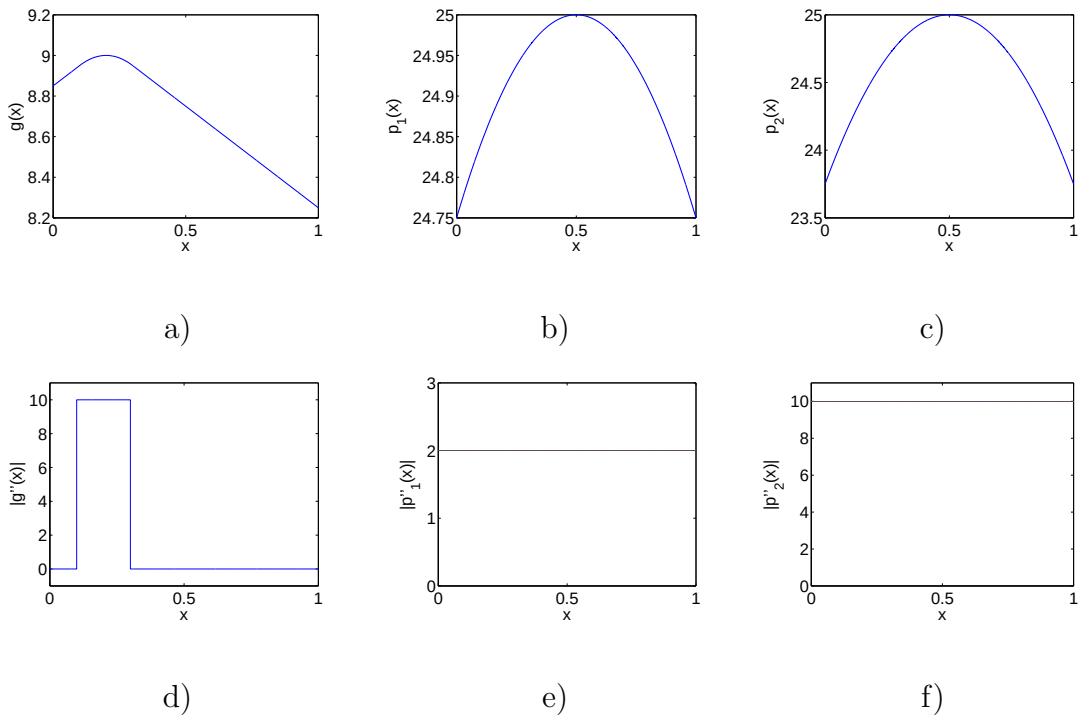


Figure 3.1. Graphs of test functions to explain the upper bound on the computational cost of `funappx_g` a) $g(x)$; b) $p_1(x)$ c) $p_2(x)$; d) $|g''(x)|$; e)$|p_1''(x)|$; f)$|p_2''(x)|$.

find when $\varepsilon = 1e-6$, the number of points required to approximate p_1 is comparable to the number of points required to approximate g. But when ε becomes smaller and smaller, more points are need for p_1 comparing to g. Especially for $\varepsilon = 1e-10$, the number of points of $g(x)$ used is almost one-third of the number of points $p_1(x)$ used, even though the second derivatives of these two functions have the same $\mathcal{L}^1$ norm.

What makes the difference? Let us investigate the second derivatives of both functions more deeply. In Figure 3.1e), we can find p_1 has the same value of $|p_1''(x)|$ on the whole interval $[0,1]$. In other words, $p_1(x)$ is peaky everywhere. This means we are actually using global adaption instead of local adaption to approximate p_1. But for g, g'' reaches its extreme value only on interval $[0.1, 0.3]$. That is, we only need to spend more points on interval $[0.1, 0.3]$. As the $\mathcal{L}^\infty$ norm of g'' is larger than that of p_1'', when ε is relatively large, p_1 doesn't need much more points than g. But when ε is decreasing, the advantage of local adaption shows up.

Table 3.1. Number of points needed for three test functions to reach different error tolerance ε by `funappx_g`

ε	g	p_1	p_2
$1e-6$	$2,401$	$3,201$	$6,401$
$1e-7$	$3,801$	$6,401$	$12,801$
$1e-8$	$11,801$	$25,601$	$51,201$
$1e-9$	$43,001$	$102,401$	$204,801$
$1e-10$	$84,201$	$204,801$	$409,601$

If we compare g with p_2, the result is much clearer. We can find p_2 needs almost five times number of points than g when $\varepsilon \leq 1e-8$ from Table 3.1, even p_2'' has the same $\mathcal{L}^\infty$ norm as g''. The computational cost is proportional to $\int_a^b |f''(x)|dx$, when the peaky parts are much closer and more concentrated. That is, Algorithm `funappx_g` can save a lot of points for functions which have the peaky parts only on some locations of the entire interval.

3.6 Numerical Examples

In this section, we firstly use `funappx_gui_g` to present how `funappx_g` does local adaption, and present three examples of `funappx_g` from computer graphics or geometric modeling. At last, we compare `funappx_g` and `funappxglobal_g` with different kinds of test functions.

Example 3. The next version of GAIL will provide an interactive demo for Algorithm `funappx_g`. The demo can be applied to any univariate function on a finite interval to visualize data points sampled by `funappx_g` from iteration to iteration. The demo is designed akin to Moler's `quadgui` [13], which illustrates adaptive integration.

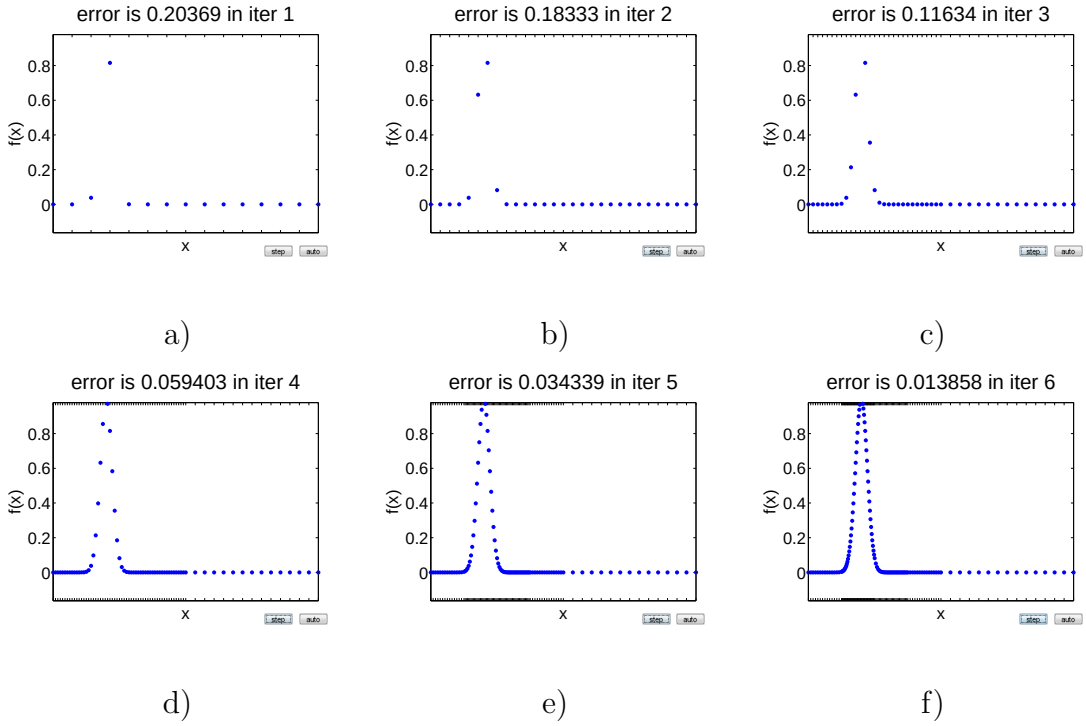


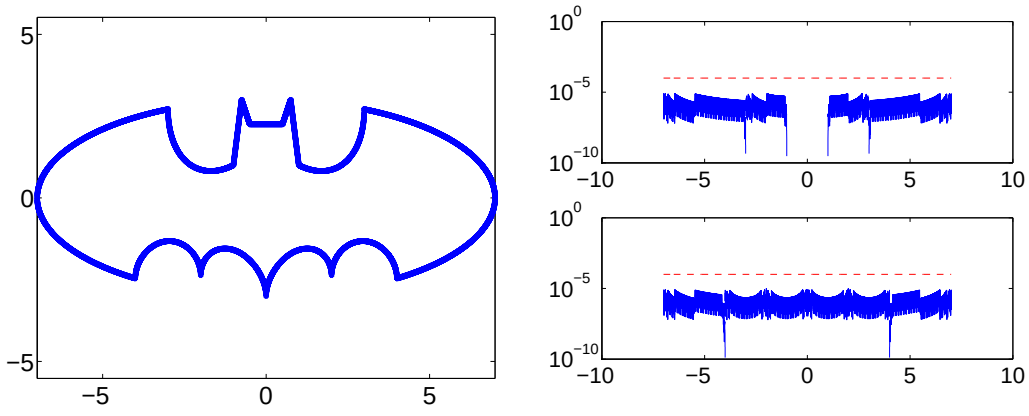
Figure 3.2. Approximation of function (3.17) step by step to show how local adaption works.

Figure 3.2 shows the approximate coordinates of

$$f(x) = \exp\left(-1000(x-0.2)^2\right) \tag{3.17}$$

on $[0, 1]$ step by step to reach the error tolerance 10^{-2}. We can find the number of points increases evenly at the second iteration. But after two iterations, the number of points only increases at the left hand side. From Figure 3.2e) to Figure 3.2f), `funappx_g` starts to sample more and more points around the peaky part. This demo can illustrate how local adaption works in `funappx_g`. For the flat parts, we can use less points. But for the peaky parts, we need more points to approximate.

Example 4. Batman is a super hero with his famous symbol. Here we use `funappx_g` to approximate the curves of the symbol using the following two piecewise functions



a) b)

Figure 3.3. a) Approximate the symbol of batman; b) Approximation error.

from [22]:

$$f_1(x) = \begin{cases} 3\sqrt{1-\frac{x^2}{49}} & 3 < |x| \le 7 \\ \frac{6}{7}\sqrt{10} + (1.5 - 0.5\,|x|) - \frac{3}{7}\sqrt{10(4-(|x|-1)^2)} & 1 < |x| \le 3 \\ 9 - 8\,|x| & 0.75 < |x| \le 1 \\ 3\,|x| + 0.75 & 0.5 < |x| \le -0.75 \\ 2.25 & |x| \le 0.5 \end{cases}$$

and

$$f_2(x) = \begin{cases} -3\sqrt{1-\frac{x^2}{49}} & 4 < |x| \le 7 \\ \frac{|x|}{2} - \frac{3\sqrt{33}-7}{112}x^2 - 3 + \sqrt{1-(||x|-2|-1)^2} & |x| \le 4 \end{cases}.$$

After we use `funappx_g` to approximate f_1 and f_2, we obtain the graphs of $\hat{f}_1$ and $\hat{f}_2$, the approximants of f_1 above the x-axis and f_2 below in Figure 3.3a). In Figure 3.3b), the top part represents the approximation error of $\hat{f}_1$ and the bottom part represents the approximation error of $\hat{f}_2$. The interesting part of Figure 3.3b) is that the errors of $\hat{f}_1$ for $x \in [-0.5, 0.5]$ seem to have disappeared. Actually, they are almost zero because $f_1(x)$ is a constant on interval $[-0.5, 0.5]$. From Figure 3.3,

we note that even though $f_1(x)$ and $f_2(x)$ are not differentiable on $[-7, 7]$, $\hat{f}_1$ and $\hat{f}_2$ satisfy the error tolerance of 10^{-4}.

`funappx_g` is an algorithm for univariate function approximation. We illustrate how to apply `funappx_g` to approximate a 3D curve or 3D surface in the following two examples.

Example 5. In this example, we use `funappx_g` to approximate a set of parametric equations, $(x(t), y(t), z(t))$ in $\mathbb{R}^3$:

$$x(t) = \exp(0.05t)\cos(t),$$
$$y(t) = \exp(0.05t)\sin(t),$$
$$z(t) = \exp(0.05t),$$

where $t \in [0, 16\pi]$. Using `funappx_g` to approximate $x(t)$, $y(t)$, and $z(t)$, we obtain the graphs of original and approximate parametric curves in Figure 3.4(a). Define the approximation error to be

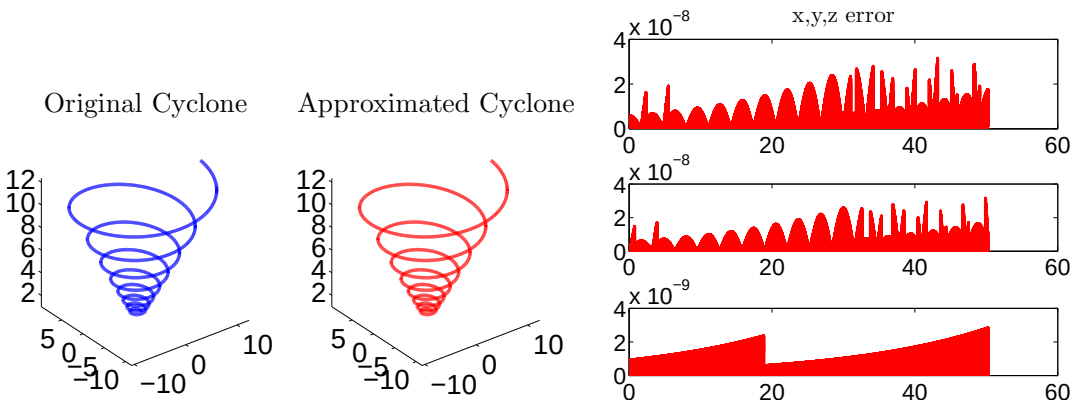
$$\mathscr{E} = \max\left\{\max_{t\in[0,16\pi]}|x(t)-\hat{x}(t)|, \max_{t\in[0,16\pi]}|y(t)-\hat{y}(t)|, \max_{t\in[0,16\pi]}|z(t)-\hat{z}(t)|\right\},$$

where $\hat{x}(t), \hat{y}(t), \hat{z}(t)$ are the approximate functions for $x(t), y(t)$ and $z(t)$ respectively.

Let the error tolerance 10^{-6} in `plotcyclone`. We obtain the approximation error of $\hat{x}(t)$, $\hat{y}(t)$, and $\hat{z}(t)$ in order in Figure 3.4(b). Thus we know $\mathscr{E} < 4 \times 10^{-8}$, which means `funappx_g` reaches the error tolerance.

Example 6. Figure 3.5a) is an approximation of a pretty Florida seashell, portrayed as a parametric surface in 3D. Let $a = -0.2, b = 0.5, c = 0.1, n = 2,\ u, v \in [0, 2\pi]$. The parametric surface is defined by the following equations in [6]:

$$x(u, v) = \left[a\left(1 - \frac{v}{2\pi}\right)(1 + \cos(u)) + c\right]\cos(nv),$$



a) b)

Figure 3.4. a) Graphs of cyclone; b) Approximation error of `funappx_g`.

$$y(u,v) = \left[a\left(1-\frac{v}{2\pi}\right)(1+\cos(u))+c\right]\sin(nv),$$
$$z(u,v) = \frac{bv}{2\pi} + a\left(1-\frac{v}{2\pi}\right)\sin(u).$$

If a function of two variables $f(x,y)$ can be separated, such as

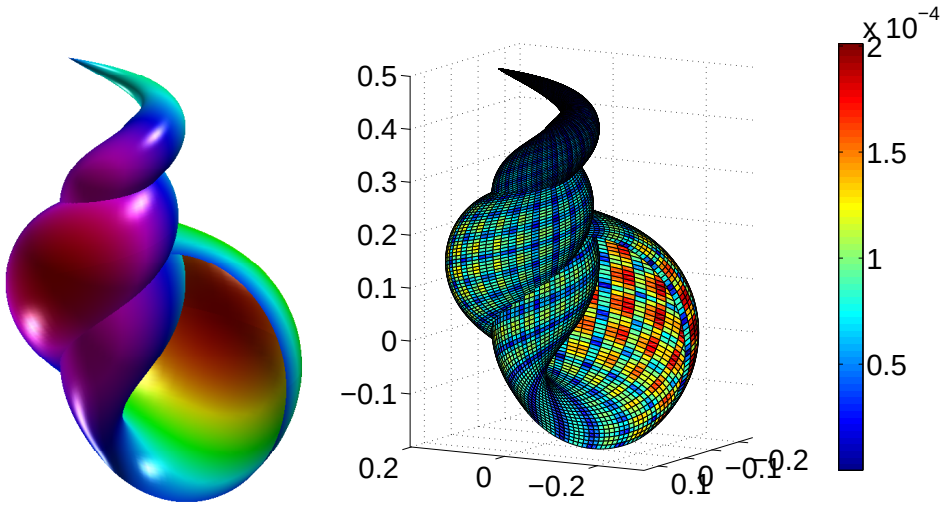
$$f(x,y) = f_1(x) + f_2(y) \text{ or } f(x,y) = f_1(x)f_2(y),$$

we can apply `funappx_g` directly to $f_1(x)$ and $f_2(y)$. However, $x(u,v)$, $y(u,v)$, and $z(u,v)$ can not be represented by the form mentioned above. Instead, we approximate $\sin(x)$ and $\cos(x)$ on the interval $[0,4\pi]$. Denote $sinappx(x)$ and $cosappx(x)$ as the approximate functions of $\sin(x)$ and $\cos(x)$. Then we have

$$\hat{x}(u,v) = \left[a\left(1-\frac{v}{2\pi}\right)(1+cosappx(u))+c\right]cosappx(nv),$$
$$\hat{y}(u,v) = \left[a\left(1-\frac{v}{2\pi}\right)(1+cosappx(u))+c\right]sinappx(nv),$$
$$\hat{z}(u,v) = \frac{bv}{2\pi} + a\left(1-\frac{v}{2\pi}\right)sinappx(u).$$

We evaluate the approximations by the following error function:

$$\mathscr{E} = \max\left\{\max_{u,v\in[0,2\pi]}|x(u,v)-\hat{x}(u,v)|,\right.$$



a) b)

Figure 3.5. a) Approximate seashell; b) Error estimation of seashell with tolerance 0.1.

$$\left.\max_{u,v\in[0,2\pi]} |y(u,v)-\hat{y}(u,v)|, \max_{u,v\in[0,2\pi]} |z(u,v)-\hat{z}(u,v)|\right\},$$

Even if we set the error tolerance as big as 0.1, we still can obtain Figure 3.5a), which is very similar to the original seashell image. The associated error plot is presented in Figure 3.5b), which shows that the approximation error $\mathscr{E} = 2 \times 10^{-4}$.

Example 7. In this example, we want to compare the globally and locally adaptive algorithms by using family of test functions, where

$$f_1(x) = (x-c)^2, \qquad f_2(x) = c\sin(c\pi x), \qquad f_3(x) = 10\exp\left(-1000(x-c)^2\right),$$

with $c \sim U[0,4]$. Figure 3.6 shows the shapes of the test functions. We use `funappx_g` and `funappxglobal_g` to approximate the family of the test functions on interval $[a,b]$, where $a = 0$ and $b = c+1$. Let

$$\varepsilon = 10^{-6}, n_{\text{lo}} = 100, n_{\text{hi}} = 1000.$$

As c is random, we run it 100 times. Thus we can obtain results in Table 3.2. From Table 3.2, we find the locally adaptive method can save computational cost compared

to globally adaptive method in most cases. But if we are dealing with the oscillatory functions, `funappx_g` costs more effort.

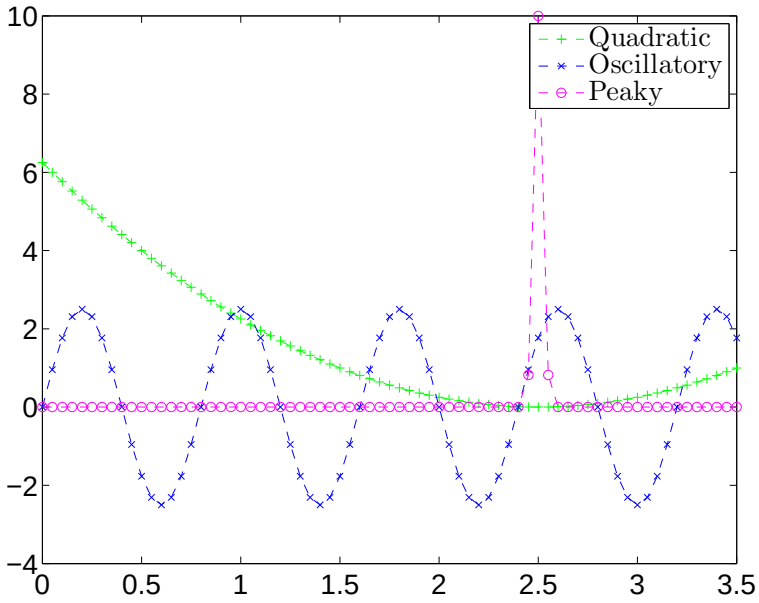


Figure 3.6. Test functions to compare `funappx_g` and `funappxglobal_g`.

For the quadratic functions, as the second derivatives of this kind of functions are the same everywhere, even we use locally adaptive algorithm however we still do it in a global way. But from Table 3.2, we can find `funappx_g` still works better than `funappxglobal_g` due to the small second derivatives.

But if we check the oscillatory functions, we can find `funappx_g` spends more time and points than `funappxglobal_g`. From Figure 3.6, we know that oscillatory functions have many peaks almost everywhere on the whole interval. To reach the error tolerance, `funappx_g` needs to double points and split in almost every subintervals in each iteration. This explains why `funappx_g` need more time than `funappxglobal_g`. Also in this kind of situation, suppose we start with the same initial number of points for both `funappx_g` and `funappxglobal_g`. The number of points `funappx_g` needed is larger or equal to the number of points `funappxglobal_g` needed, because the way of increasing number in `funappx_g` is always a multiplier

Table 3.2. Comparison of number of sample points and computational time between `funappx_g` and `funappxglobal_g`. This table can be conditionally reproduced by `workout_funappx_g`.

Test	Number of Points		Time Used	
Function	Global	Local	Global	Local
f_1: Quadratic	199,374	76,604	0.0250	0.0129
f_2: Oscillatory	440,906	543,268	0.0603	0.1100
f_3: Peaky	2,136,821	222,714	0.2941	0.0516

which is an exponential number with base 2. In other words, if `funappx_g` reaches the error tolerance at $l-$th iteration, the computational cost of `funappx_g` is $2^l(n_0-1)+1$. But for `funappxglobal_g`, there exists a constant C such that

$$2^{l-1} < C \leq 2^l,$$

to make `funappxglobal_g` succeed with computational cost $C(n_0 - 1) + 1$. That's why `funappx_g` can cost more points than `funappxglobal_g`.

For peaky functions, we obtain a totally different result from oscillatory ones. By Table 3.2, we can find `funappxglobal_g` needs almost ten times number of points larger than `funappx_g` to reach the same error tolerance. `funappx_g` also uses one-sixth computational time comparing to `funappxglobal_g`.

From this experiment, we can see globally and locally adaptive methods have their own pros and cons. How to combine their advantages together would be a very interesting problem to investigate.

CHAPTER 4

CONCLUSION

4.1 Summary

In many cases of univariate function approximation, the algorithm can be denoted as $A_n(f)$, where f is the function to approximate and n denotes the number of sample size. With input f and n, the algorithm returns $\hat{f} = A_n(f)$. But we are not satisfied with this kind of algorithm. We want an algorithm which returns $\hat{f}$ within a guaranteed accuracy ε to the input function f, i.e.,

$$\left\| f - \hat{f} \right\|_\infty \leq \varepsilon.$$

In addition, we also want the algorithm to determine the sample size, n, from the error tolerance ε and information obtained by sampling the function f. So we can denote our algorithm as $A(f, \varepsilon)$.

For example, let $A_n(f)$ be the piecewise linear spline. With the condition $\|f''\|_\infty \leq \sigma$, we get

$$n = \left\lceil \sqrt{\frac{(b-a)^2 \sigma}{8\varepsilon}} \right\rceil + 1. \tag{4.1}$$

Thus we can let $A(f, \varepsilon) = A_n(f)$ to obtain the sample size n, where $A(f, \varepsilon)$ has the guaranteed accuracy. However, the sample size is determined by the error tolerance ε and σ, not the size of f''. In other words, the algorithm $A(f, \varepsilon)$ is not adaptive.

For univariate function approximation, adaption means that the algorithm adjusts its computational effort based on information gathered from sampling the input function f. For example, if an initial modest-sized sample of a function determines it to be linear, then no further samples are needed. For the same error tolerance, the number of samples required to approximate the linear function $x \mapsto \alpha + \beta x$ defined on $[0, 1]$ should be much fewer than the number of samples required to approximate

$x \mapsto \sin(1000x)$ on the same domain. Algorithms that satisfy this requirement must generally be adaptive.

The key idea of establishing guaranteed adaptive algorithms is to perform error analysis for cones —instead of balls— of input functions. As we approximate functions via piecewise linear interpolation, we can use the function data to obtain an adaptive upper bound on $\|f''\|_\infty$ under the cone condition. Thus, we can obtain an adaptive error bound by the same data. By letting the error bound satisfy the error tolerance, we can obtain guaranteed adaptive algorithms.

A guaranteed globally adaptive algorithm `funappxglobal_g`, starting with evenly spaced points, can adjust its effort globally based on information about the function obtained through sampling. The algorithm stops when it reaches the error tolerance ε or exceeds the cost budget. The upper bound on the computational cost of `funappxglobal_g` is

$$\text{cost}(A, f; \varepsilon) \leq \left\lceil \sqrt{\frac{n^*(b-a)^2 \|f''\|_\infty}{2\varepsilon}} + 2n^* \right\rceil + 1.$$

This bound is of the same order as bound on the computational cost for which the algorithms knowing $\|f''\|_\infty$ as a priori. The lower bound on the complexity of the problem is also given:

$$\text{comp}(\varepsilon, \mathcal{A}(\mathcal{C}_{n^*}, \mathcal{L}^\infty), \mathcal{B}_s) \geq \left\lceil \sqrt{\frac{(b-a)^2(n^*-1)s}{32n^*\varepsilon}} \right\rceil.$$

As the lower bound on the complexity of the problem is of the same order as the upper bound on the computational cost of `funappxglobal_g`, we can say our algorithm is optimal. Numerical examples are used to illustrate our algorithm can succeed even when the functions are not inside the cone. The adaptive algorithm without rigorous justifications is fooled in Example 2 shows how important the guaranteed accuracy is.

Locally adaptive algorithms, which can adjust sampling density according to

the function data, are more interesting and efficient than global adaptive algorithms. There is some work on locally adaptive algorithms for optimization problems, but less for univariate function approximation, and even less with guaranteed accuracy. In this dissertation, we also establish our guaranteed locally adaptive algorithm `funappx_g`. Still starting with evenly spaced points, `funappx_g` samples more points where the function is peaky. In each iteration, the algorithm refines the partition of the whole interval. Algorithm `funappx_g` stops when the error bounds in all the subintervals in the current partition do not exceed the error tolerance ε. The algorithm also stops when it attempts to exceed the cost budget or the maximum number of iterations.

It is difficult to obtain an explicit formula for the computational cost of `funappx_g`. When we derive the computational cost, we obtain an inequality with l both sides,

$$2^l \geq \sqrt{\frac{n_l^*(b-a)^2 \left\|f''\right\|_{I_{x,l}}}{8(n_0-1)(n_0-1-n_l^*)\varepsilon}}.$$

We cannot solve l in an explicit form from the above inequality. But when ε is close enough to zero, the upper bound on the computational cost of `funappx_g` tends to

$$\mathrm{cost}(\mathcal{A}, f; \mathcal{P}, \varepsilon) \lesssim \sqrt{\frac{n_{\mathrm{lo}}}{2\varepsilon(n_0-1-n_{\mathrm{lo}})}} \left\| \sqrt{|f''|} \right\|_1 dx + 1.$$

We can find the cost does not only depend on how peaky the function is, but also depends on where the peaky part of the function locates.

In addition, a GUI example is used to illustrate how `funappx_g` samples points in each iteration. Numerical examples from approximating piecewise functions to 3D surface are also given for the locally adaptive algorithm. From the comparison between `funappxglobal_g` and `funappx_g`, we find that `funappxglobal_g` has advantages for the functions which have large $\left\|f''\right\|_\infty$ on almost the entire interval, while `funappx_g` has advantages for the functions which have peaky parts on a subinterval of the entire interval.

In our algorithm, we use piecewise linear interpolation to approximate the function. If we want to improve the efficiency of our algorithms, we should consider higher order piecewise polynomial splines. This problem is presented with more detail in the future work.

In conclusion, `funappxglobal_g` and `funappx_g` are adaptive algorithms with guarantees to succeed for the appropriate cones of input functions. Our goal is to offer users a reliable software, which fails with the necessary conditions and sends warning messages to user when it may fail. Users may want to integrate, approximate or optimize some spiky functions. The adaptive algorithms without rigorous guarantees can be easily fooled and don't show any warning message. But in GAIL, we have guarantees to succeed using our algorithms for the input functions inside our cone. Even if the algorithms fail, we know the reason why they fail and what we can do to solve this situation. That's why we believe that there should be more adaptive algorithms with rigorous guarantees of their success and related software. We hope that this will inspire further research in this direction.

4.2 Future Work

The results presented in this dissertation suggest a number of interesting open problems, some of which we are working on. Here is a summary.

4.2.1 Guaranteed accuracy on hybrid error tolerance of guaranteed adaptive algorithms for univariate function approximation. In this dissertation, we investigate the guaranteed accuracy on absolute error for univariate approximation. This analysis should be extended to include *relative* error tolerance ε_r. The error requirement of `funappx_g` can be modified from

$$\|f - \hat{f}\|_\infty \leq \varepsilon,$$

where the tolerance function is simply the absolute error tolerance ε, to a combination

of both ε and ε_r, for example,

$$\|f - \hat{f}\|_\infty \leq \max\{\varepsilon, \varepsilon_r \|f\|_\infty\}.$$

Even though $\|f\|_\infty$ is unknown, it can be easily estimated at increasing accuracy from the data points iteratively. The challenge is to derive results analogous to Theorem 1 and Theorem 4. We can try to apply the same idea as in [11], which is used for multivariate integration.

4.2.2 Develop high order guaranteed adaptive algorithms for univariate function approximation. In this dissertation, we use piecewise linear splines to approximate functions. The algorithms based on piecewise linear splines have low-order convergence. If we want to establish guaranteed adaptive algorithms with *higher order convergence*, we need smoother input functions and we can use high order piecewise splines, such as cubic splines etc. After we pick up an interpolation method, we need to select the appropriate semi-norm and function space for error analysis. Next, we need to identify a suitable weaker semi-norm and construct a cone of input functions with the selected semi-norm and weaker semi-norm. Thus we might obtain an adaptive error bound and establish the guaranteed adaptive algorithm.

We can also consider to use Chebyshev polynomials to approximate function. Chebfun is a widely used toolbox, but lacks rigorous justification We want to extend our existing guaranteed adaptive algorithms to a higher-order convergence guaranteed adaptive algorithm based on Chebyshev points. We would need to figure out the proper semi-norm and function space for error analysis. An alternative would be to look at the decay rate of the Chebyshev coefficients and construct a cone related with the decay rate. This idea is similar to the work of Hickernell and Jiménez Rugama [9, 12].

4.2.3 Parallel guaranteed adaptive algorithms for univariate function ap-

proximation. If we try to approximate functions with very high-accuracy or on a very large interval $[a, b]$ using `funappx_g` or `funappxglobal_g`, the current implementation may run out of memory on a personal machine if more than 10 million data points are required. For such problems, an alternative is to divide the interval $[a, b]$ into some small enough subintervals, then apply `funappx_g` or `funappxglobal_g` on these smaller subintervals. When we approximate the function in all subintervals simultaneously, we can consider using parallel MATLAB to save the time. After we obtain the interpolants, we can try to construct a piecewise function to combine all the interpolants together.

4.2.4 Develop adaptive guaranteed algorithms in a new cone without penalty. The upper bound on the computational cost of `funappxglobal_g` is as following:

$$\text{cost}(A, f; \varepsilon) \leq \left\lceil \sqrt{\frac{n^*(b-a)^2 \left\|f''\right\|_\infty}{2\varepsilon}} + 2n^* \right\rceil + 1.$$

When $\varepsilon \to 0$, the order of the upper bound is $O\left(\sqrt{\frac{n^*(b-a)^2\|f''\|_\infty}{\varepsilon}}\right)$. To eliminate the penalty of n^*, we need to construct a new cone. In the paper for integration [10], Hickernell presents a new way to construct cone, which can be applied to the function approximation. We want to construct a cone similar to the cone in that paper, and modify the order of computational cost without penalty of n^* so that

$$O\left(\sqrt{\frac{n^*(b-a)^2 \left\|f''\right\|_\infty}{\varepsilon}}\right) \text{ becomes } O\left(\sqrt{\frac{(b-a)^2 \left\|f''\right\|_\infty}{\varepsilon}}\right).$$

4.2.5 Develop guaranteed adaptive algorithms for high dimensional function approximation. We can also consider to establish guaranteed adaptive algorithm for high dimensional functional approximation. For $d = 2$, we can consider to construct a triangular mesh or rectangular mesh. We refine the mesh until we reach the error tolerance. For d is large, the recent work of Xuan Zhou and Fred Hickernell for multidimensional-function approximation using radial basis functions [23]. Un-

fortunately, the error bounds are too restrictive in the sense that they are applicable only to tiny domains, and consequently have not made a practical impact.

APPENDIX A

MATLAB CODE OF GUARANTEED GLOBALLY ADAPTIVE ALGORITHM FUNAPPXGLOBAL_G

```
function [pp,out_param]=funappxglobal_g(varargin)
%FUNAPPXGLOBAL_G 1-D guaranteed function recovery on a closed interval
%[a,b]
%
%   pp = FUNAPPXGLOBAL_G(f) approximates function f on the default
%   interval [0,1] by a piecewise polynomial structure pp within the
%   guaranteed absolute error tolerance of 1e-6. Default initial number
%   of points is 100 and default cost budget is 1e7.  Input f is a
%   function handle. The statement y = f(x) should accept a vector
%   argument x and return a vector y of function values that is of the
%   same size as x. Output pp may be evaluated via PPVAL.
%
%   pp = FUNAPPXGLOBAL_G(f,a,b,abstol,nlo,nhi,nmax) for a given
%   function f and the ordered input parameters that define the finite
%   interval [a,b], a guaranteed absolute error tolerance abstol, a
%   lower bound of initial number of points nlo, an upper bound of
%   initial number of points nhi, and a cost budget nmax.
%
%   pp = FUNAPPXGLOBAL_G(f,'a',a,'b',b,'abstol',abstol,'nlo',nlo,'nhi',
%   nhi,'nmax',nmax) recovers function f on the finite interval [a,b],
%   given a guaranteed absolute error tolerance abstol, a lower bound of
%   initial number of points nlo, an upper bound of initial number of
%   points nhi, and a cost budget nmax. All six field-value pairs are
%   optional and can be supplied in different order.
%
%   pp = FUNAPPXGLOBAL_G(f,in_param) recovers function f on the finite
%   interval [in_param.a,in_param.b], given a guaranteed absolute error
%   tolerance in_param.abstol, a lower bound of initial number of points
%   in_param.nlo, an upper bound of initial number of points
%   in_param.nhi, and a cost budget in_param.nmax. If a field is not
%   specified, the default value is used.
%
%   [pp, out_param] = FUNAPPXGLOBAL_G(f,...) returns a piecewise
%   polynomial structure pp and an output structure out_param.
%
%   Input Arguments
%
%     in_param.a --- left end point of interval, default value is 0
%
%     in_param.b --- right end point of interval, default value is 1
%
%     in_param.abstol --- guaranteed absolute error tolerance, default
%     value is 1e-6
%
%     in_param.nlo --- lower bound of initial number of points we used,
%     default value is 10
%
%     in_param.nhi --- upper bound of initial number of points we used,
%     default value is 1000
%
%     in_param.nmax --- cost budget, default value is 1e7
%
%   Output Arguments
%
```

```
%     pp.form --- pp means piecewise polynomials
%
%     pp.breaks --- show the location of interpolation points
%
%     pp.coefs --- coefficients for piecewise linear polynomials
%
%     pp.pieces --- number of piecewise linear polynomials
%
%     pp.order --- be 2 as we use piecewise linear polynomials
%
%     pp.dim --- be 1 as we do univariate approximation
%
%     pp.orient --- always be 'first'
%
%     out_param.exceedbudget --- it is 0 if the number of points used in
%     the construction of pp is less than cost budget, 1 otherwise.
%
%     out_param.ninit --- initial number of points we use
%
%     out_param.npoints --- number of points we need to reach the
%     guaranteed absolute error tolerance
%
%     out_param.errorbound --- an upper bound of the absolute error
%
%     out_param.nstar --- final value of the parameter defining the cone
%     of functions for which this algorithm is guaranteed;
%     nstar = ninit-2 initially and is increased as necessary
%
%     out_param.a --- left end point of interval
%
%     out_param.b --- right end point of interval
%
%     out_param.abstol --- guaranteed absolute error tolerance
%
%     out_param.nlo --- a lower bound of initial number of points we use
%
%     out_param.nhi --- an upper bound of initial number of points we
%     use
%
%     out_param.nmax --- cost budget
%
%  Guarantee
%
%  If the function to be approximated, f, satisfies the cone condition
%                            2 nstar   ||     f(b)-f(a)  ||
%      ||f''||        <=  ----------  ||f'- ------------ ||
%             \infty         b - a    ||       b - a    ||\infty,
%  then the pp output by this algorithm is guaranteed to satisfy
%      ||f-ppval(pp, )||\infty <= abstol,
%  and the upper bound of the cost is
%           ----------------------------
%         / nstar*(b-a)^2 ||f''||\infty
%        / ------------------------------ + 2 nstar + 4
%      \/           2 abstol
```

```
%
%  provided the flag exceedbudget = 0.
%
%
%  Examples
%
%   Example 1:
%
%
%   >> f = @(x) x.^2; [pp, out_param] = funappxglobal_g(f)
%
%     pp =
%       form: 'pp'
%     breaks: [1x5051 double]
%      coefs: [5050x2 double]
%     pieces: 5050
%      order: 2
%        dim: 1
%     orient: 'first'
%
%     out_param =
%                 f: @(x)x.^2
%                 a: 0
%                 b: 1
%           abstol: 1.0000e-06
%              nlo: 10
%              nhi: 1000
%             nmax: 10000000
%            nstar: 100
%            ninit: 102
%     exceedbudget: 0
%          npoints: 5051
%       errorbound: 9.9990e-07
%
%
%   Example 2:
%
%   >> f = @(x) x.^2;
%   >> [pp, out_param] = funappxglobal_g(f,-2,2,1e-7,10,10,1000000)
%
% pp =
%
%       form: 'pp'
%     breaks: [1x38149 double]
%      coefs: [38148x2 double]
%     pieces: 38148
%      order: 2
%        dim: 1
%     orient: 'first'
%
%     out_param =
%                 a: -2
%           abstol: 1.0000e-07
%                 b: 2
```

```
%                f: @(x)x.^2
%              nhi: 10
%              nlo: 10
%             nmax: 1000000
%            nstar: 10
%            ninit: 12
%     exceedbudget: 0
%          npoints: 38149
%       errorbound: 2.7493e-08
%
%
%   Example 3:
%
%  >> f = @(x) x.^2;
%  >>[pp, out_param]=funappxglobal_g(f,'a',-2,'b',2,'nhi',100,'nlo',10)
%
% pp =
%
%       form: 'pp'
%     breaks: [1x31851 double]
%      coefs: [31850x2 double]
%     pieces: 31850
%      order: 2
%        dim: 1
%     orient: 'first'
%
% out_param =
%
%                a: -2
%           abstol: 1.0000e-06
%                b: 2
%                f: @(x)x.^2
%              nhi: 100
%              nlo: 10
%             nmax: 10000000
%            nstar: 64
%            ninit: 66
%     exceedbudget: 0
%          npoints: 31851
%       errorbound: 2.5286e-07
%
%
%   Example 4:
%
%   >> in_param.a = -10; in_param.b = 10;
%   >> in_param.abstol = 10^(-7); in_param.nlo = 10; in_param.nhi = 100;
%   >> in_param.nmax = 10^6; f = @(x) x.^2;
%   >> [pp, out_param] =funappxglobal_g(f,in_param)
%
% pp =
%
%       form: 'pp'
%     breaks: [1x596779 double]
%      coefs: [596778x2 double]
```

```
%     pieces: 596778
%      order: 2
%        dim: 1
%     orient: 'first'
%
%   out_param =
%                a: -10
%           abstol: 1.0000e-07
%                b: 10
%                f: @(x)x.^2
%              nhi: 100
%              nlo: 10
%             nmax: 1000000
%            nstar: 90
%            ninit: 92
%     exceedbudget: 0
%          npoints: 596779
%       errorbound: 2.5274e-08
%
%
%   See also INTEGRAL_G, MEANMC_G, CUBMC_G
%
%  References
%
%   [1]  Nick 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 (2014),
%   pp. 21-45.
%
%
%   [2]  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. Available from
%   http://code.google.com/p/gail/
%
%   If you find GAIL helpful in your work, please support us by citing
%   the above paper and software.
%

% check parameter satisfy conditions or not
[f, out_param] = funappx_g_param(varargin{:});

MATLABVERSION= gail.matlab_version;
if MATLABVERSION >= 8.3
    warning('off', 'Matlab:interp1:ppGriddedInterpolant');
end;

%% main algorithm

% initialize number of points
% initialize nstar
out_param.ninit = out_param.nstar + 2;
n = out_param.ninit;
```

```
% cost budget flag
out_param.exceedbudget = 1;
% tau change flag
tauchange = 0;
% length of interval
len = out_param.b-out_param.a;
% add flag
flag = 0;

while n < out_param.nmax;

    if(flag==0)
        x = out_param.a:len/(n-1):out_param.b;
        y = f(x);
    else
        xnew = repmat(x(1:end-1),m-1,1)...
            +repmat((1:m-1)'*len/(n-1),1,(n-1)/m);
        ynew = f(xnew);
        xnew = [x(1:end-1); xnew];
        x = [xnew(:); x(end)]';
        ynew = [y(1:end-1); ynew];
        y = [ynew(:); y(end)]';
    end;
    diff_y = diff(y);
    %approximate the weaker norm of input function
    gn = (n-1)/len*max(abs(diff_y-(y(n)-y(1))/(n-1)));
    %approximate the stronger norm of input function
    fn = (n-1)^2/len^2*max(abs(diff(diff_y)));

    % Stage 2: satisfy necessary condition
    if out_param.nstar*(2*gn+fn*len/(n-1)) >= fn*len;
        % Stage 3: check for convergence
        errbound = 4*out_param.abstol*(n-1)*(n-1-out_param.nstar)...
            /out_param.nstar/len;
        % satisfy convergence
        if errbound >= gn;
            out_param.exceedbudget = 0; break;
        end;
        % otherwise increase number of points
        m = max(ceil(1/(n-1)*sqrt(gn*out_param.nstar*...
            len/4/out_param.abstol)),2);
        n = m*(n-1)+1;
        flag = 1;
        % Stage2: do not satisfy necessary condition
    else
        % increase tau
        out_param.nstar = ceil(fn/(2*gn/len+fn/(n-1)));
        % change tau change flag
        tauchange = 1;
        % check if number of points large enough
        if n >= out_param.nstar+2;
            % true, go to Stage 3
            errbound = 4*out_param.abstol*(n-1)*(n-1-out_param.nstar)...
                /out_param.nstar/len;
```

```
            if errbound >= gn;
                out_param.exceedbudget = 0; break;
            end;
            m = max(ceil(1/(n-1)*sqrt(gn*out_param.nstar*...
                len/4/out_param.abstol)),2);
            n = m*(n-1)+1;
            flag = 1;
        else
            % otherwise increase number of points, go to Stage 1
            %n = 2 + ceil(out_param.nstar);
            m = ceil((2*out_param.nstar + 1)/(2*n-2));
            n = m*(n-1)+1;
            flag = 1;
        end;
    end;
end;

if tauchange == 1;
    warning('GAIL:funappxglobal_g:peaky',['This function is peaky '...
    'relative to nlo and nhi. You may wish to increase nlo and nhi'...
    ' for similar functions.'])
end;

% Check cost budget flag
if out_param.exceedbudget == 1;
    n = 1 + (n-1)/m*floor((out_param.nmax-1)*m/(n-1));
    warning('GAIL:funappxglobal_g:exceedbudget',['funappxglobal_g '...
    'attempted to exceed the cost budget. The answer may be '...
    'unreliable.'])
    out_param.npoints = n;
    nstar = out_param.nstar;
    out_param.errorbound = gn*len*nstar/(4*(n-1)*(n-1-nstar));
    x1 = out_param.a:len/(out_param.npoints-1):out_param.b;
    y1 = f(x1);
    pp = interp1(x1,y1,'linear','pp');
else
    out_param.npoints = n;
    nstar = out_param.nstar;
    out_param.errorbound = gn*len*nstar/(4*(n-1)*(n-1-nstar));
    pp = interp1(x,y,'linear','pp');
end;

if MATLABVERSION >= 8.3
    warning('on', 'Matlab:interp1:ppGriddedInterpolant');
end;


function [f, out_param] = funappx_g_param(varargin)
% parse the input to the funappx_g function

%% Default parameter values

default.abstol = 1e-6;
default.a = 0;
```

```
default.b = 1;
default.nlo = 10;
default.nhi = 1000;
default.nmax  = 1e7;


if isempty(varargin)
    warning('GAIL:funappxglobal_g:nofunction',['Function f must be '...
    'specified. Now GAIL is using f(x)=x^2 and unit interval [0,1].'])
    help funappxglobal_g
    f = @(x) x.^2;
    out_param.f = f;
else
  if gail.isfcn(varargin{1})
    f = varargin{1};
    out_param.f = f;
  else
    warning('GAIL:funappxglobal_g:notfunction',['Function f must be'...
    ' a function handle. Now GAIL is using f(x)=x^2.'])
    f = @(x) x.^2;
    out_param.f = f;
  end
end;

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

if ~validvarargin
    %if only one input f, use all the default parameters
    out_param.a = default.a;
    out_param.b = default.b;
    out_param.abstol = default.abstol;
    out_param.nlo = default.nlo;
    out_param.nhi = default.nhi;
    out_param.nmax = default.nmax;
else
    p = inputParser;
    addRequired(p,'f',@gail.isfcn);
    if isnumeric(in2)%if there are multiple inputs with
        %only numeric, they should be put in order.
        addOptional(p,'a',default.a,@isnumeric);
        addOptional(p,'b',default.b,@isnumeric);
        addOptional(p,'abstol',default.abstol,@isnumeric);
        addOptional(p,'nlo',default.nlo,@isnumeric);
        addOptional(p,'nhi',default.nhi,@isnumeric);
        addOptional(p,'nmax',default.nmax,@isnumeric);
    else
        if isstruct(in2) %parse input structure
            p.StructExpand = true;
            p.KeepUnmatched = true;
        end
```

```
        addParamValue(p,'a',default.a,@isnumeric);
        addParamValue(p,'b',default.b,@isnumeric);
        addParamValue(p,'abstol',default.abstol,@isnumeric);
        addParamValue(p,'nlo',default.nlo,@isnumeric);
        addParamValue(p,'nhi',default.nhi,@isnumeric);
        addParamValue(p,'nmax',default.nmax,@isnumeric);
    end
    parse(p,f,varargin{2:end})
    out_param = p.Results;
end;

if (out_param.a == inf||out_param.a == -inf||isnan(out_param.a)==1)
    warning('GAIL:funappxglobal_g:anoinfinity',['a can not be '...
    'infinity. Use default a = ' num2str(default.a)])
    out_param.a = default.a;
end;
if (out_param.b == inf||out_param.b == -inf||isnan(out_param.b)==1)
    warning('GAIL:funappxglobal_g:bnoinfinity',['b can not be '...
    'infinity. Use default b = ' num2str(default.b)])
    out_param.b = default.b;
end;

if (out_param.b < out_param.a)
    warning('GAIL:funappxglobal_g:blea',['b can not be smaller '...
    'than a; exchange these two. '])
    tmp = out_param.b;
    out_param.b = out_param.a;
    out_param.a = tmp;
elseif(out_param.b == out_param.a)
    warning('GAIL:funappxglobal_g:beqa',['b can not equal a. '...
    'Use b = ',num2str(out_param.a+1)])
    out_param.b = out_param.a+1;
end;

% let error tolerance greater than 0
if (out_param.abstol <= 0 )
    warning(['GAIL:funappxglobal_g:abstolnonpos ','Error tolerance'...
    'should be greater than 0. Using default error tolerance ',...
    num2str(default.abstol)])
    out_param.abstol = default.abstol;
end

% let cost budget be a positive integer
if (~gail.isposint(out_param.nmax))
    if gail.isposintive(out_param.nmax)
        warning('GAIL:funappxglobal_g:budgetnotint',['Cost budget'...
        ' should be a positive integer. Using cost budget ',...
        num2str(ceil(out_param.nmax))])
        out_param.nmax = ceil(out_param.nmax);
    else
        warning('GAIL:funappxglobal_g:budgetisneg',['Cost budget'...
        ' should be a positive integer. Using default cost budget '...
        int2str(default.nmax)])
        out_param.nmax = default.nmax;
```

```
    end;
end

if (~gail.isposint(out_param.nlo))
        warning('GAIL:funappxglobal_g:lowinitnotint',['Lower bound'...
        ' of initial nstar should be a positive integer.' ...
        ' Using ', num2str(ceil(out_param.nlo)) ' as nlo '])
        out_param.nlo = ceil(out_param.nlo);
end
 if (~gail.isposint(out_param.nhi))
        warning('GAIL:funappxglobal_g:hiinitnotint',['Upper bound '...
        ' of initial nstar should be a positive integer.' ...
        ' Using ', num2str(ceil(out_param.nhi)) ' as nhi' ])
        out_param.nhi = ceil(out_param.nhi);
end

if (out_param.nlo > out_param.nhi)
    warning('GAIL:funappxglobal_g:logrhi',['Lower bound of initial '...
    'number of points is larger than upper bound of initial number '...
    'of points; Use nhi as nlo'])
    out_param.nhi = out_param.nlo;
end;
if (out_param.nlo > out_param.nmax)
    warning('GAIL:funappxglobal_g:logecost',['Lower bound of '...
    'initial number of points should be smaller than cost budget. '...
    'Using ',num2str(ceil(out_param.nmax/2))])
    out_param.nlo = out_param.nmax/2;
end;
if (out_param.nhi > out_param.nmax)
    warning('GAIL:funappxglobal_g:higecost',['Upper bound of '...
    'initial number of points should be smaller than cost budget. '...
    'Using ',num2str(out_param.nlo)])
    out_param.nhi = out_param.nlo;
end;

h = out_param.b - out_param.a;
out_param.nstar = ceil(out_param.nhi*(out_param.nlo/out_param.nhi)...
                  ^(1/(1+h)));
```

APPENDIX B
MATLAB CODE OF GUARANTEED LOCALLY ADAPTIVE ALGORITHM FUNAPPX_G

```
function [fappx,out_param]=funappx_g(varargin)
%funappx_g 1-D guaranteed locally adaptive function approximation (or
%   function recovery) on [a,b]
%
%   fappx = funappx_g(f) approximates function f on the default interval
%   [0,1] by an approximated function handle fappx within the guaranteed
%   absolute error tolerance of 1e-6. When Matlab version is higher or
%   equal to 8.3, fappx is an interpolant generated by
%   griddedInterpolant. When Matlab version is lower than 8.3, fappx is
%   a function handle generated by ppval and interp1. Input f is a
%   function handle. The statement y = f(x) should accept a vector
%   argument x and return a vector y of function values that is of the
%   same size as x.
%
%   fappx = funappx_g(f,a,b,abstol) for a given function f and the
%   ordered input parameters that define the finite interval [a,b], and
%   a guaranteed absolute error tolerance abstol.
%
%   fappx = funappx_g(f,'a',a,'b',b,'abstol',abstol) approximates
%   function f on the finite interval [a,b], given a guaranteed absolute
%   error tolerance abstol. All four field-value pairs are optional and
%   can be supplied in different order.
%
%   fappx = funappx_g(f,in_param) approximates function f on the finite
%   interval [in_param.a,in_param.b], given a guaranteed absolute error
%   tolerance in_param.abstol. If a field is not specified, the default
%   value is used.
%
%   [fappx, out_param] = funappx_g(f,...) returns an approximated
%   function fappx and an output structure out_param.
%
%   Input Arguments
%
%     f --- input function
%
%     in_param.a --- left end point of interval, default value is 0
%
%     in_param.b --- right end point of interval, default value is 1
%
%     in_param.abstol --- guaranteed absolute error tolerance, default
%     value is 1e-6
%
%   Optional Input Arguments
%
%     in_param.nlo --- lower bound of initial number of points we used,
%     default value is 10
%
%     in_param.nhi --- upper bound of initial number of points we used,
%     default value is 1000
%
%     in_param.nmax --- when number of points hits the value, iteration
%     will stop, default value is 1e7
%
%     in_param.maxiter --- max number of iterations, default value is
```

```
%     1000
%
%   Output Arguments
%
%     fappx ---- approximated function handle (Note: When Matlab version
%     is higher or equal to 8.3, fappx is an interpolant generated by
%     griddedInterpolant. When Matlab version is lower than 8.3, fappx
%     is a function handle generated by ppval and interp1.)
%
%     out_param.f ---- input function
%
%     out_param.a ---- left end point of interval
%
%     out_param.b ---- right end point of interval
%
%     out_param.abstol ---- guaranteed absolute error tolerance
%
%     out_param.nlo ---- a lower bound of initial number of points we use
%
%     out_param.nhi ---- an upper bound of initial number of points we
%     use
%
%     out_param.nmax ---- when number of points hits the value, iteration
%     will stop
%
%     out_param.maxiter ---- max number of iterations
%
%     out_param.ninit ---- initial number of points we use for each sub
%     interval
%
%     out_param.exit ---- this is a vector with two elements, defining
%     the conditions of success or failure satisfied when finishing the
%     algorithm. The algorithm is considered successful (with
%     out_param.exit == [0 0]) if no other flags arise warning that the
%     results are certainly not guaranteed. The initial value is [0 0]
%     and the final value of this parameter is encoded as follows:
%
%                     [1 0]   If reaching overbudget. It states whether
%                     the max budget is attained without reaching the
%                     guaranteed error tolerance.
%
%                     [0 1]   If reaching overiteration. It states
%                     whether the max iterations is attained without
%                     reaching the guaranteed error tolerance.
%
%     out_param.iter ---- number of iterations
%
%     out_param.npoints ---- number of points we need to reach the
%     guaranteed absolute error tolerance
%
%     out_param.errest ---- an estimation of the absolute error for the
%     approximation
%
%     out_param.nstar ---- final value of the parameter defining the
```

```
%     cone of functions for which this algorithm is guaranteed for each
%     subinterval; nstar = floor(ninit/2) initially
%
%     out_param.x --- sample points used to approximate function
%
%     out_param.bytes --- amount of memory used during the computation
%
%  Guarantee
%
%  For [a,b] there exists a partition, P={[t_0,t_1], [t_1,t_2], ...,
%  [t_{L-1},t_L]}, where a=t_0 < t_1 < ... < t_L=b. If the function to
%  be approximated, f, satisfies the cone condition
%                            2 nstar    ||     f(t_l)-f(t_{l-1})||
%   ||f''||        <=  ---------------  ||f'- ------------------ ||
%          \infty       t_l - t_{l-1}  ||        t_l - t_{l-1} ||\infty,
%  for each sub interval [t_{l-1},t_l], where 1 <= l <= L, then the
%  output fappx by this algorithm is guaranteed to satisfy
%      ||f-fappx||\infty <= abstol.
%
%   Examples
%
%   Example 1:
%
%   >> f = @(x) x.^2;
%   >> [~, out_param] = funappx_g(f,-2,2,1e-7,10,20)
%
%  out_param =
%
%               a: -2
%          abstol: 1.0000e-***7
%               b: 2
%               f: @(x)x.^2
%         maxiter: 1000
%             nhi: 20
%             nlo: 10
%            nmax: 10000000
%           nstar: [1x1024 double]
%           ninit: 37
%            exit: [2x1 logical]
%            iter: 11
%         npoints: 36865
%          errest: 4.5329e-***8
%               x: [1x36865 double]
%           bytes: 3203626
%
%
%   Example 2:
%
%   >> f = @(x) x.^2;
%   >> [~, out_param] = funappx_g(f,'a',-2,'b',2,'nhi',20,'nlo',10)
%
% out_param =
%
%               a: -2
```

```
%           abstol: 1.0000e-***6
%                b: 2
%                f: @(x)x.^2
%          maxiter: 1000
%              nhi: 20
%              nlo: 10
%             nmax: 10000000
%            nstar: [1x256 double]
%            ninit: 37
%             exit: [2x1 logical]
%             iter: 9
%          npoints: 9217
%           errest: 7.2526e-***7
%                x: [1x9217 double]
%            bytes: 803970
%
%
%   Example 3:
%
%   >> in_param.a = -5; in_param.b = 5; f = @(x) x.^2;
%   >> in_param.abstol = 10^(-6); in_param.nlo = 10; in_param.nhi = 20;
%   >> [~, out_param] = funappx_g(f,in_param)
%
% out_param =
%
%                a: -5
%           abstol: 1.0000e-***6
%                b: 5
%                f: @(x)x.^2
%          maxiter: 1000
%              nhi: 20
%              nlo: 10
%             nmax: 10000000
%            nstar: [1x512 double]
%            ninit: 39
%             exit: [2x1 logical]
%             iter: 10
%          npoints: 19457
%           errest: 9.9555e-***7
%                x: [1x19457 double]
%            bytes: 1690074
%
%
%   See also INTERP1, GRIDDEDINTERPOLANT, INTEGRAL_G, MEANMC_G, FUNMIN_G
%
%
%  References
%
%   [1]  Nick 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, pp. 21-45,
%   2014.
%
%   [2]  Yuhan Ding, Fred J. Hickernell, and Sou-Cheng T. Choi, "Locally
```

```
%   Adaptive Method for Approximating Univariate Functions in Cones with
%   a Guarantee for Accuracy," working, 2015.
%
%   [3]  Sou-Cheng T. Choi, Yuhan Ding, Fred J. Hickernell, Lan Jiang,
%   Lluis Antoni Jimenez Rugama, Xin Tong, Yizhi Zhang and Xuan Zhou,
%   GAIL: Guaranteed Automatic Integration Library (Version 2.1) [MATLAB
%   Software], 2015. Available from http://code.google.com/p/gail/
%
%   [4] Sou-Cheng T. Choi, "MINRES-QLP Pack and Reliable Reproducible
%   Research via Supportable Scientific Software," Journal of Open
%   Research Software, Volume 2, Number 1, e22, pp. 1-7, 2014.
%
%   [5] Sou-Cheng T. Choi and Fred J. Hickernell, "IIT MATH-573 Reliable
%   Mathematical Software" [Course Slides], Illinois Institute of
%   Technology, Chicago, IL, 2013. Available from
%   http://code.google.com/p/gail/
%
%   [6] Daniel S. Katz, Sou-Cheng T. Choi, Hilmar Lapp, Ketan
%   Maheshwari, Frank Loffler, Matthew Turk, Marcus D. Hanwell, Nancy
%   Wilkins-Diehr, James Hetherington, James Howison, Shel Swenson,
%   Gabrielle D. Allen, Anne C. Elster, Bruce Berriman, Colin Venters,
%   "Summary of the First Workshop On Sustainable Software for Science:
%   Practice And Experiences (WSSSPE1)," Journal of Open Research
%   Software, Volume 2, Number 1, e6, pp. 1-21, 2014.
%
%   If you find GAIL helpful in your work, please support us by citing
%   the above papers, software, and materials.
%

% check parameter satisfy conditions or not
[f, out_param] = funappx_g_param(varargin{:});
MATLABVERSION= gail.matlab_version;

%%main algorithm
% initialize nstar
%nstar = ninit - 2;
nstar = out_param.nstar;
% initialize number of points
out_param.ninit = 2 * nstar + 1;
ninit = out_param.ninit;
index = [1 ninit];
% initialize error
abstol = out_param.abstol;
err = abstol + 1;
len = out_param.b - out_param.a;
x = out_param.a:len/(ninit-1):out_param.b;
y = f(x);
iSing = find(isinf(y));
if ~isempty(iSing)
    error('GAIL:funappx_g:yInf',['Function f(x) = Inf at x = ',...
     num2str(x(iSing))]);
end
if length(y) == 1
    % probably f is a constant function and Matlab would
```

```
    % reutrn only a scalar y = f(x) even if x is a vector
    f = @(x) f(x) + 0 * x;
    y = f(x);
end
iter = 0;
exit_len = 2;
%we start the algorithm with all warning flags down
out_param.exit = false(exit_len,1);

while(max(err) > abstol)
    % length of each subinterval
    len = x(index(2:end))-x(index(1:end-1));
    reshapey = reshape(y(1:end-1),ninit - 1, length(index)-1);
    diffy = diff([reshapey;y(index(2:end))]);
    %approximate the weaker norm of input function at different
    %subinterval
    gn = (ninit-1)./len.*max(abs(bsxfun(@minus,diffy,(y(index(2:end))...
        -y(index(1:end-1)))/(ninit-1))),[],1);
    %approximate the stronger norm of input function at different
    %subinterval
    fn = (ninit-1)^2./(len.^2).*max(abs(diff(diffy)),[],1);
    %update cone condition every iteration
    nstar=ceil(out_param.nhi*(out_param.nlo...
        /out_param.nhi).^(1./(1+len)));
    %find nstar not large enough then double it
    smallconeind = find(nstar.*(2*gn+fn.*len/(ninit-1)) <(fn.*len));
    nstarsmallcone = (fn.*len)./(2*gn+fn.*len/(ninit-1)).*...
          (2*ninit-2-(fn.*len)./(2*gn+fn.*len/(ninit-1)))/(ninit-1);
    nstar(smallconeind) = nstarsmallcone(smallconeind);
    iter = iter + 1;
    err = nstar.*len.*gn./(4*(ninit-1).*(ninit-1-nstar));
    %check if error satisfy the error tolerance
    counterr = sum(err > abstol);
    if(length(x) + counterr *(ninit -1) > out_param.nmax)
        out_param.exit(1) = 1;
        warning('GAIL:funappx_g:exceedbudget',['funappx_g attempted'...
        ' to exceed the cost budget. The answer may be unreliable.'])
        break;
    end;
    %if max(err) > abstol;
    if counterr >= 1;
        %flag sub interval error not satisfy error tolerance 1 in whbad
        whbad = err > abstol;
        %add index for bad sub interval
        badind = find(whbad == 1);
        %flag sub interval error satisfy error tolerance 1 in whgood
        whgood = (whbad ==0);
        %add index for good sub interval
        goodind = find(whgood == 1);
        %find # of new sub intervals need to be added at each sub
        %interval
        badcumsum = cumsum(whbad);
        %pickup # of new sub intervals at bad intervals
        cumbad = badcumsum(badind);
```

```
%generate new index of sub intervals splitted from bad intervals
newindex = [badind + [0 cumbad(1:end-1)]; badind + cumbad];
newindex = newindex(:)';
%find the length of each sub interval
%h = len/2/(ninit-1);
h = (out_param.b-out_param.a)/2^iter/(ninit-1);
%reshape x without end point to a matrix of ninit-1 by # of
%intervals
reshapex =  reshape(x(1:end-1),ninit -1,...
    (index(end) - 1)/(ninit -1));
%generate new points newx need to be added
%newx = bsxfun(@plus,reshapex(:,badind),h(badind));
newx = reshapex(:,badind)+h;
%compute value newy of newx
newy = f(newx);
%initialize a zero matrix of 2*(ninit-1) by # of bad
%subintervals to store all the points after splitting bad
%subintervals
badmatrix = zeros(2*(ninit-1),sum(whbad));
%insert x at bad sub intervals in badmatrix as the row 1,
%3,..., end-1
badmatrix(1:2:end-1,:) = reshapex(:,badind);
%insert newx at bad sub intervals in badmatrix as the row 2,
%4,..., end
badmatrix(2:2:end,:) = newx;
%reshape badmatrix to the size of ninit -1 by 2*# of bad sub
%intervals
badmatreshape = reshape(badmatrix, ninit - 1, 2*sum(whbad));
%initialize a matrix of ninit - 1 by # of sub intervals after
%splitting bad sub intervals for x
newreshapex = zeros(ninit - 1, 2*sum(whbad)+sum(whgood));
%insert all the points after splitting bad sub intervals to
%correct column
newreshapex(:,newindex) = badmatreshape;
%insert all the points on good sub intervals to correct column
newreshapex(:,goodind + badcumsum(goodind)) = ...
    reshapex(:,goodind);
%obtain all the points in vector x
x = [newreshapex(:)' x(end)];
%insert y at bad sub intervals in badmatrix as the row 1,
%3,..., end-1
badmatrix(1:2:end-1,:) = reshapey(:,badind);
%insert newy at bad sub intervals in badmatrix as the row 2,
%4,..., end
badmatrix(2:2:end,:) = newy;
%reshape badmatrix to the size of ninit -1 by 2*# of bad sub
%intervals
badmatreshape = reshape(badmatrix, ninit - 1, 2*sum(whbad));
%initialize a matrix of ninit - 1 by # of sub intervals after
%splitting bad sub intervals for y
newreshapey = zeros(ninit - 1, 2*sum(whbad)+sum(whgood));
%insert all the values after splitting bad sub intervals to
%correct column
newreshapey(:,newindex) = badmatreshape;
```

```
        %insert all the original y on good sub intervals to correct
        %column
        newreshapey(:,goodind + badcumsum(goodind)) = ...
            reshapey(:,goodind);
        %obtain all the values in vector y
        y = [newreshapey(:)' y(end)];
        iSing = find(isinf(y));
        if ~isempty(iSing)
            error('GAIL:funappx_g:yInf',['Function f(x) = Inf at'...
            ' x = ', num2str(x(iSing))]);
        end
        %generate error for new sub intervals
        %initialize a vertor of # of sub intervals after splitting
        newerr = zeros(1,2*sum(whbad)+sum(whgood));
        %use the same error for splitted bad interval
        baderr = [err(badind); err(badind)];
        %insert error after splitting bad sub intervals to correct
        %position
        newerr(newindex)=baderr(:)';
        newerr(goodind + badcumsum(goodind)) = err(goodind);
        %obtain error for all sub intervals
        err = newerr;
        %upadte index w.p.t x after splitting
        index = 1:(ninit-1):length(err)*(ninit-1)+1;
    else
        break;
    end;
    if(iter==out_param.maxiter)
        out_param.exit(2) = 1;
        warning('GAIL:funappx_g:exceediter',['Number of iterations'...
        ' has reached maximum number of iterations.'])
        break;
    end;
end;
out_param.iter = iter;
out_param.npoints = index(end);
out_param.errest = max(err);
out_param.nstar = nstar;
out_param.x = x;
w = whos;
out_param.bytes = sum([w.bytes]);
if MATLABVERSION >= 8.3
    fappx = griddedInterpolant(x,y,'linear');
else
    pp = interp1(x,y,'linear','pp');
    fappx =@(x) ppval(pp,x);
end;

function [f, out_param] = funappx_g_param(varargin)
% parse the input to the funappx_g function

%% Default parameter values
default.abstol = 1e-6;
default.a = 0;
```

```
default.b = 1;
default.nlo = 10;
default.nhi = 1000;
default.nmax = 1e7;
default.maxiter = 1000;

MATLABVERSION= gail.matlab_version;
if MATLABVERSION >= 8.3
    f_addParamVal = @addParameter;
else
    f_addParamVal = @addParamValue;
end;

if isempty(varargin)
  warning('GAIL:funappx_g:nofunction',['Function f must be '...
      'specified. Now GAIL is using f(x)=exp(-100*(x-0.5)^2) and '
      'unit interval [0,1].'])
  help funappx_g
  f = @(x) exp(-100*(x-0.5).^2);
  out_param.f = f;
else
  if gail.isfcn(varargin{1})
    f = varargin{1};
    out_param.f = f;
  else
    warning('GAIL:funappx_g:notfunction',['Function f must be a '...
        'function handle. Now GAIL is using f(x)=exp(-100*(x-0.5)^2).'])
    f = @(x) exp(-100*(x-0.5).^2);
    out_param.f = f;
  end
end;

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

if ~validvarargin
    %if only one input f, use all the default parameters
    out_param.a = default.a;
    out_param.b = default.b;
    out_param.abstol = default.abstol;
    out_param.nlo = default.nlo;
    out_param.nhi = default.nhi;
    out_param.nmax = default.nmax ;
    out_param.maxiter = default.maxiter;
else
    p = inputParser;
    addRequired(p,'f',@gail.isfcn);
    if isnumeric(in2)%if there are multiple inputs with
        %only numeric, they should be put in order.
        addOptional(p,'a',default.a,@isnumeric);
```

```
        addOptional(p,'b',default.b,@isnumeric);
        addOptional(p,'abstol',default.abstol,@isnumeric);
        addOptional(p,'nlo',default.nlo,@isnumeric);
        addOptional(p,'nhi',default.nhi,@isnumeric);
        addOptional(p,'nmax',default.nmax,@isnumeric)
        addOptional(p,'maxiter',default.maxiter,@isnumeric)
    else
        if isstruct(in2) %parse input structure
            p.StructExpand = true;
            p.KeepUnmatched = true;
        end
        f_addParamVal(p,'a',default.a,@isnumeric);
        f_addParamVal(p,'b',default.b,@isnumeric);
        f_addParamVal(p,'abstol',default.abstol,@isnumeric);
        f_addParamVal(p,'nlo',default.nlo,@isnumeric);
        f_addParamVal(p,'nhi',default.nhi,@isnumeric);
        f_addParamVal(p,'nmax',default.nmax,@isnumeric);
        f_addParamVal(p,'maxiter',default.maxiter,@isnumeric);
    end
    parse(p,f,varargin{2:end})
    out_param = p.Results;
end;

% let end point of interval not be infinity
if (out_param.a == inf||out_param.a == -inf)
    warning('GAIL:funappx_g:aisinf',['a cannot be infinity.'...
        'Use default a = ' num2str(default.a)])
    out_param.a = default.a;
end;
if (out_param.b == inf||out_param.b == -inf)
    warning(['GAIL:funappx_g:bisinf','b cannot be infinity. '...
        'Use default b = ' num2str(default.b)])
    out_param.b = default.b;
end;

if (out_param.b < out_param.a)
    warning('GAIL:funappx_g:blea',['b cannot be smaller than a;'...
        ' exchange these two. '])
    tmp = out_param.b;
    out_param.b = out_param.a;
    out_param.a = tmp;
elseif(out_param.b == out_param.a)
    warning('GAIL:funappx_g:beqa',['b cannot equal a.'...
        'Use b = ' num2str(out_param.a+1)])
    out_param.b = out_param.a+1;
end;
% let error tolerance greater than 0
if (out_param.abstol <= 0 )
    warning('GAIL:funappx_g:tolneg', ['Error tolerance should be'...
        ' greater than 0. Using default error tolerance ',...
        num2str(default.abstol)])
    out_param.abstol = default.abstol;
end
% let cost budget be a positive integer
```

```
if (~gail.isposint(out_param.nmax))
    if gail.isposintive(out_param.nmax)
        warning('GAIL:funappx_g:budgetnotint',['Cost budget should'...
            ' be a positive integer. Using cost budget '...
            , num2str(ceil(out_param.nmax))])
        out_param.nmax = ceil(out_param.nmax);
    else
        warning('GAIL:funappx_g:budgetisneg',['Cost budget should'...
            ' be a positive integer. Using default cost budget '...
            int2str(default.nmax)])
        out_param.nmax = default.nmax;
    end;
end

if (~gail.isposint(out_param.nlo))
        warning('GAIL:funappxglobal_g:lowinitnotint',['Lower bound'...
        ' of initial nstar should be a positive integer.' ...
        ' Using ', num2str(ceil(out_param.nlo)) ' as nlo '])
        out_param.nlo = ceil(out_param.nlo);
end
 if (~gail.isposint(out_param.nhi))
        warning('GAIL:funappxglobal_g:hiinitnotint',['Upper bound'...
        ' of initial nstar should be a positive integer.' ...
        ' Using ', num2str(ceil(out_param.nhi)) ' as nhi' ])
        out_param.nhi = ceil(out_param.nhi);
end

if (out_param.nlo > out_param.nhi)
    warning('GAIL:funappx_g:logrhi', ['Lower bound of initial number'...
        ' of points is larger than upper bound of initial number of '...
        'points; Use nhi as nlo'])
    out_param.nhi = out_param.nlo;
end;

h = out_param.b - out_param.a;
out_param.nstar = ceil(out_param.nhi*(out_param.nlo/out_param.nhi)...
    ^(1/(1+h)));

if (~gail.isposint(out_param.maxiter))
    if gail.ispositive(out_param.maxiter)
        warning('GAIL:funappx_g:maxiternotint',['Max number of '...
            'iterations should be a positive integer. Using max '...
            'number of iterations as ',...
             num2str(ceil(out_param.maxiter))])
        out_param.nmax = ceil(out_param.nmax);
    else
        warning('GAIL:funappx_g:budgetisneg',['Max number of '...
            'iterations should be a positive integer. Using max '...
            'number of iterations as ' int2str(default.maxiter)])
        out_param.nmax = default.nmax;
    end;
end
```

BIBLIOGRAPHY

[1] R. L. Burden and J. D. Faires. *Numerical Analysis, Ninth Edition*. Brooks/Cole, 2011.

[2] S-C. T. Choi, Y. Ding, F. J. Hickernell, L. Jiang, Ll. A. Jiménez Rugama, X. Tong, Y. Zhang, and X. Zhou. GAIL: Guaranteed Automatic Integration Library (version 2.0). MATLAB software, `http://gailgithub.github.io/GAIL_Dev`, Nov 2014.

[3] S-C. T. Choi, Y. Ding, F. J. Hickernell, L. Jiang, Ll. A. Jiménez Rugama, X. Tong, Y. Zhang, and X. Zhou. GAIL: Guaranteed Automatic Integration Library (version 2.1). MATLAB software, `http://gailgithub.github.io/GAIL_Dev`, Mar 2015.

[4] S-C. T. Choi, Y. Ding, F. J. Hickernell, L. Jiang, and Y. Zhang. GAIL: Guaranteed Automatic Integration Library (version 1.0). MATLAB software, `http://gailgithub.github.io/GAIL_Dev`, Sep 2013.

[5] N. Clancy, Y. Ding, C. Hamilton, F. J. Hickernell, and Y. Zhang. The cost of deterministic, adaptive, automatic algorithms: Cones, not balls. *Journal of Complexity*, 30:21–45, Feb 2014.

[6] T. A. Davis and K. Sigmon. *MATLAB Primer, 7th Edition*. CRC Press, 2005.

[7] N. Hale, L. N. Trefethen, and T. A. Driscoll. *Chebfun Version 5.2*, 2015.

[8] F. J. Hickernell, L. Jiang, Y. Liu, and A. B. Owen. Guaranteed conservative fixed width confidence intervals via Monte Carlo sampling. In G. W. Peter J. Dick, F. Y. Kuo and I. H. Sloan, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2012*, pages 105–128. Springe-Verlag, 2013.

[9] F. J. Hickernell and Ll. A. Jiménez Rugama. Reliable adaptive cubature using digital sequences. Submitted for publication: arXiv:1410.8615, 2014.

[10] F. J. Hickernell, M. Razo, and S. Yun. Reliable adaptive numerical integration. preprint, 2015.

[11] L. Jiang and Ll. A. Jiménez Rugama. personal communication.

[12] Ll. A. Jiménez Rugama and F.J. Hickernell. Adaptive multidimensional integration based on rank-1 lattices. Submitted for publication: arXiv:1411.1966, 2014.

[13] C. B. Moler. *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics, 2004.

[14] E. Novak. On the power of adaption. *J. Complexity*, 12:199–237, 1996.

[15] L. Plaskota and G. W. Wasilkowski. Adaption allows efficient integration of functions with unknown singularities. *Numer. Math.*, pages 123–144, 2005.

[16] L. Plaskota, G. W. Wasilkowski, and Y. Zhao. The power of adaption for approximating functions with singularities. *Math. Comput.*, pages 2309–2338, 2008.

[17] The MathWorks, Inc. *MATLAB 8.1*. Natick, MA, 2013.

[18] The Numerical Algorithms Group. *The NAG Library*. Oxford, Mark 23 edition, 2012.

[19] X. Tong. A guaranteed, adaptive, automatic algorithm for univariate function minimization. Master's thesis, Illinois Insitute of Technology, July 2014.

[20] J. F. Traub, G. W. Wasilkowski, and H. Woźniakowski. *Information-Based Complexity*. Academic Press, Boston.

[21] J. F. Traub and A. G. Werschulz. *Complexity and Information*. Cambridge University Press, Cambridge.

[22] M. Zhang. Batman Equation: The Legend. `http://www.mathworks.com/matlabcentral/fileexchange/33124-batman-equation--the-legend`, 2011.

[23] X. Zhou and F. J. Hickernell. Tractability of the function approximation problem in terms of the kernel's shape and scale parameters. *arXiv preprint arXiv:1411.0790*, 2014.