Automated Generation of Numerical Evaluation Routines for Bivariate Functions via Tensor Product Series

by

Xiang Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis, we present a method for the automated generation of numerical evaluation routines for bivariate functions via tensor product series and develop a toolkit to assist with the generation of the approximations. The final approximations can be evaluated in user-defined precision or in hardware floating point precision by default. The evaluation routines can also be compiled into a C library (or a library in some other language) for more efficient evaluations.

The toolkit can be used for various mathematical functions of two variables, such as Bessel functions or user-defined functions, at any given precision. The method of tensor product series expansion reduces the bivariate approximation problem to a sequence of univariate approximation problems. In order to control the degrees of the approximating functions so that evaluation will be accurate and efficient, we recursively divide the bivariate intervals into subintervals until both the number of terms in the tensor product series and the degrees of the univariate approximations are less than specified bounds. We then generate in each subinterval rational approximations using Chebyshev-Padé approximants or polynomial approximations using Chebyshev series according to the user's specification.

Finally we show the experimental results for a variety of bivariate functions, which achieve a significant speedup over the original Maple functions for evaluation in hardware floating point precision. We also compare the results of choosing polynomial approximations versus rational approximations for the univariate subproblems.

Acknowledgements

My deepest thanks go to my supervisor, Professor Keith Geddes, who shepherded me from the formative stages of this thesis to the final draft and spent a tremendous amount of time editing this thesis. His sage advice, careful guidance and patient explanation aided the writing of this thesis in innumerable ways.

I would like to extend my appreciation to Professor George Labahn and Professor Justin Wan, who are the readers of this thesis.

I would also like to thank all of my labmates and friends who gave me invaluable suggestions and made my life more interesting.

Dedication

I would like to dedicate this thesis to my fiance, Huanyu Chen, who offered me unconditional love and support all the time. This thesis is also dedicated to my dear parents and grandparents for their understanding and encouragement throughout my lifetime.

谨以此论文献给我远在中国的最亲爱的爸爸妈妈和爷爷奶奶！

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis investigates a method for the automated generation of numerical evaluation routines for bivariate functions via tensor-product series. The goal of this thesis is to develop a Maple toolkit that automatically generates evaluation routines which can be compiled into a C library (or into a library in some other language) for efficient evaluation in hardware floating point precision.

1.1 Previous Work

Thomas A. Robinson in his Masters thesis [9] developed a toolkit to assist with the generation of numerical evaluation routines for approximations of univariate functions at any fixed precision. He investigated various approximation methods to improve the efficiency of the Maple implementation, including Chebyshev series expansions and minimax approximation. The approximation process of the toolkit is to divide the real line interval on which it is approximating into subintervals and then generate polynomial or rational approximations on each subinterval.

In his chapter of future work, Thomas A. Robinson considered the possibility of adapting the method of tensor product series, which was developed by Frederick W. Chapman in his PhD thesis [3], to the case of generating approximations for bivariate functions. The approach used by Robinson was not very successful and therefore further study was warranted.

As a continuation of Thomas A. Robinson's work, Jingchi Chen in his Masters essay [4] developed a method to approximate bivariate functions, such as the Bessel function $J_v(x)$. The method uses Chapman's tensor product series expansion to approximate a bivariate function into a sequence of univariate functions, and then uses truncated Chebyshev series for the univariate approximations. However, as mentioned in Chen's last chapter, his method only works on a fixed-sized single

interval and this limits the applicability of the method. Another limitation is that the final approximation is a single bivariate polynomial and as the number of terms gets large, numerical accuracy problems arise.

1.2 Motivation

In mathematical software, as well as in many non-mathematical applications, a common requirement is to be able to evaluate functions at numerical (floating point) values. For the most common elementary functions (such as e^x, $\cos x$, etc.), numerical libraries have long ago been developed based on approximations (typically polynomial or rational function approximations) valid for a specified fixed precision [5]. In a more general mathematical software environment such as Maple, routines have been developed for arbitrary-precision evaluation in Maple's software floating point mode. For the most common univariate functions, Maple is able to dispatch to efficient compiled code in numerical libraries if the requested precision is not too high (e.g. hardware floating point precision). However, for bivariate functions such numerical libraries have not been developed.

This thesis is considered as the continuation of the work done by Thomas A. Robinson and Jingchi Chen. In this thesis, we investigate the generation of approximations for bivariate functions in the x-y plane. Our toolkit generates piecewise polynomial or rational approximations. When we talk about generation of an approximation here, we mean to generate a new function which is sufficiently close to the original function for a specified fixed precision. The goal is to have the approximating function be significantly faster to evaluate than the original function when evaluated at the specified precision.

Our overall plan is to use the toolkit to generate routines for efficient numerical evaluation at any point in a specified region of the x-y plane in advance and to build a library in Maple (or other languages) to be used when needed. So we are not overly concerned about the time required to generate the approximation functions. What we mostly care about is the time efficiency and accuracy when we use the approximations to obtain values of the function at various points.

Before proceeding, here is an outline of the rest of this thesis. Chapter 2 presents some definitions and describes the three modes for evaluating functions in Maple as technical background. We then look at various techniques for approximation of univariate functions, including polynomial approximations in Chapter 3 and rational approximations in Chapter 4. Chapter 5 then discusses the approximation of bivariate functions using tensor product series. This provides background to continue to Chapter 6 where we present our method and package for automated generation of numerical evaluation routines for bivariate functions. Chapter 7 presents the

corresponding experimentation and results for some approximations computed. Finally, in Chapter 8 possible future work and in Chapter 9 the general conclusion of this thesis are stated.

Chapter 2

Technical Background

In this chapter we explain the definition of some terms which we will use later.

2.1 Expression vs. Procedure

In mathematics, an expression is a term consisting of a single symbol, such as x, or a well-formed combination of mathematical symbols, such as $3x^2 + y - 1$. In Maple we usually assign an expression to a name so that we can use the expression again more conveniently thereafter. For example,

$$> \quad expn := 3x^2 + y - 1;$$

is a very common use in Maple. In this case $expn$ is an expression in terms of the variables x and y. Then we may use $expn$ instead of $3x^2 + y - 1$ whenever the latter is needed.

In Maple, a procedure can be assigned to a name. A procedure, which is also called functional operator form, can be invoked by a function call. We can change an expression to a function in procedure form. For example, a procedure

$$> \quad f := (x, y) \rightarrow 3x^2 + y - 1;$$

can be derived from expression $expn$ by using the Maple command `unapply`:

$$> \quad f := unapply(expn, (x, y));.$$

We may also convert procedure form f to expression form $expn$ by using the `apply` command in Maple

$$> \quad expn := apply(f, (x, y));.$$

which is equivalent to the function invocation $f(x, y)$.

In particular, if f is a functional operator in Maple taking two arguments then

$$> \quad unapply(f(x, y), (x, y));$$

yields a functional operator equivalent to f. Analogously,

$$> \quad unapply(expn, (x, y))(x, y);$$

yields an expression equivalent to $expn$.

An expression is usually evaluated by the `eval` command. For example, $expn$ can be evaluated at the point $(x, y) = (2, 3)$ by using

$$> \quad eval(expn, [x = 2, y = 3]);.$$

Unlike expressions, a procedure can be simply evaluated by a function call, such as $f(2, 3)$.

Our toolkit takes an expression as input for the function to be approximated. We will convert between the expression form and procedure form in our code when necessary.

2.2 Norms

If we have a function f and its approximation q which we want to use for obtaining values of f, then we are interested in knowing the maximum error of the approximation for all the points on the interval.

A commonly used norm is the maximum (or infinity) norm which is defined as

$$\|f - q\|_\infty = \max_{a \le x \le b} |f(x) - q(x)| \tag{2.1}$$

for a univariate function $f(x)$ and its approximation $q(x)$ on the interval $[a, b]$.

For a bivariate function $f(x, y)$ and its approximation $q(x, y)$ in the region $R = [a, b] \times [c, d]$, the maximum norm is defined as

$$\|f - q\|_\infty = \max_{(x,y) \in R} |f(x, y) - q(x, y)| . \tag{2.2}$$

2.3 Absolute Error vs. Relative Error

There are two common measures of error in experimental science: absolute and relative error. In mathematics, absolute error is the difference between the exact value and the approximation value. The relative error is the absolute error divided by the exact value, indicating how many significant digits are correct in the approximation.

Let us use $f(x, y)$ to denote the original bivariate function which needs to be approximated and use $q(x, y)$ to denote an approximation of $f(x, y)$. We also use $p(x, y)$ and $r(x, y)$ to denote the polynomial and rational function respectively. In our work, $q(x, y)$ is a collection of $p_i(x, y)$ or $r_i(x, y)$, or possibly a combination of both $p_i(x, y)$ and $r_i(x, y)$. Then the absolute error at a point (x, y) is

$$\epsilon = |f(x, y) - q(x, y)| \tag{2.3}$$

and if $f(x, y) \neq 0$, the relative error is

$$\eta = \frac{|f(x, y) - q(x, y)|}{|f(x, y)|}. \tag{2.4}$$

We can see that the absolute error is associated with the number of correct decimal places in the decimal representation of $q(x, y)$, while the relative error can be associated with the number of correct significant digits in the decimal representation of $q(x, y)$ [5].

As mentioned in Chapter 1, our goal is to generate an approximation $q(x, y)$ which is sufficiently close to $f(x, y)$. When we say sufficiently close here, we mean that the relative error of $q(x, y)$ as an approximation of $f(x, y)$ is equal or less than the desired accuracy which is hardware floating point precision by default.

However, when we give a specified error tolerance as an argument to the procedure for tensor product series generation, the measure is not treated as a pure relative error tolerance. Additionally, when we use Chebyshev series expansion with a given error tolerance on a specified interval to estimate the degree of the approximation (by which we then know whether it is necessary to divide the interval into subintervals), again it is not treated as a pure relative error tolerance.

In the following section, we will discuss the various measures of error.

2.4 Measures of Error

For a numerical library in a floating point environment, it is generally desired that the relative error should be small. For a bivariate function $f(x, y)$ and its approximation $q(x, y)$ in the region R, the pointwise relative error is

$$relerr = \max_{(x,y)\in R^*} \frac{|f(x,y) - q(x,y)|}{|f(x,y)|} \tag{2.5}$$

where $R^* = \{(x,y) \in R : f(x,y) \neq 0\}$.

In our project, we are using previously-written code for generating tensor product series expansions. In order to explain the measure of accuracy achieved by an approximation generated by this code, we first define what we call global relative error:

$$global\_relerr = \frac{\|f - q\|_\infty}{\|f\|_\infty} \tag{2.6}$$

which is the absolute error $\|f - q\|_\infty$ divided by $\|f\|_\infty$, the maximum absolute value of $f(x,y)$.

Note that if there are points $(x,y) \in R$ where $f(x,y)$ is much smaller than $\|f\|_\infty$, then $global\_relerr < tol$ does not imply that $relerr < tol$. Therefore, $global\_relerr$ is a weaker measure than $relerr$ (the pure pointwise relative error measure).

Finally, the actual measure of error used in our project (which is the actual measure of accuracy achieved by the tensor product code) is a further modification of relative error, namely:

$$modified\_relerr = \frac{\|f - q\|_\infty}{\max(\|f\|_\infty, 1)}. \tag{2.7}$$

For a given region $R = [a,b] \times [c,d]$, we can see that $modified\_relerr$ reduces to a pure absolute error measure if $\|f\|_\infty \leq 1$. Otherwise $modified\_relerr$ is equivalent to the global relative error measure $global\_relerr$.

2.5 Numerical Evaluation in Maple

In Maple, there are three environments in which we may numerically evaluate a function : `evalf` (or software floats), `evalhf` (or hardware floats), and compiled code (which may be generated by the Maple compiler).

In this section we discuss these three modes.

2.5.1 `evalf`

The `evalf` command in Maple is used to numerically evaluate expressions including constants and mathematical functions. What we especially care about in our work is its use in evaluating functions. The `evalf` command uses Maple's software floating-point arithmetic. The precision of the computation can be controlled by the environment variable `Digits`, which is set to 10 by default. So we may compute the value of an expression to any precision by assigning the corresponding positive integer value n to `Digits`. Alternatively, we could call `evalf[n](expression)` to specify the numeric precision for an `evalf` computation without changing the value of `Digits`.

Maple's initially known mathematical functions, such as $sin(x)$ or $J_v(x)$, can be evaluated with `evalf`. Additionally, these functions automatically invoke the corresponding `evalf` subfunction if they are passed a floating-point argument. Maple dispatches to a numerical library if the precision is not greater than hardware floating point precision and if the numerical library includes the given function. Otherwise, a general arbitrary-precision routine is used and it executes in software floating point mode (which is significantly slower).

2.5.2 `evalhf`

Maple has another useful command `evalhf` which can numerically evaluate a function in double precision using the floating-point hardware of the underlying system. The argument evaluated by `evalhf` can be either a standard function, such as $sin(x)$, or a user-defined Maple procedure (as in our work). However, such a user-defined procedure must be "purely numerical" which excludes many Maple programming constructs. In order to avoid the conversion overhead it is recommended to do all possible computations within one single call to `evalhf`.

Using `evalhf` for evaluations is usually much faster than using `evalf`. This is because the former evaluates functions in hardware floats while the latter works in software floats. However, the precision of the results when using `evalhf` is limited by the number of `Digits` in the hardware on which Maple is run. For example, if `Digits` is set to the integer part of the value `evalhf(Digits)` (usually 14 or 15 on 32-bit architectures) then `evalhf` and `evalf` should produce similar results. Because of the limitation, it is more difficult to guarantee a specified high accuracy by using `evalhf`. Some special functions are also more difficult to implement in the hardware floating point environment than in the software floating point environment. There are only 79 functions, such as $sin(x)$ and $cos(x)$, that can be handled directly by `evalhf`, and consequently executed without using Maple code, simply by calling the corresponding mathematical function from the C library or executing C code [7]. Our toolkit can create approximations for various special

bivariate functions and add them to the set of 79 functions. Then we will be able to evaluate these special functions by `evalhf` instead of `evalf`, which can speed up the evaluation. For example, using hardware numerical routines for the Bessel functions will accelerate their plotting and other relevant applications within hardware floating point accuracy.

Our toolkit is able to generate the approximations to any user-specified level of accuracy or hardware floating point accuracy by default. For example, if the user just wishes to create a simpler approximation than hardware accuracy for plotting or if the user wants to generate an approximation to full hardware precision, then efficient evaluation will be achieved in hardware floats, which is considerably faster than in software floats.

As will be illustrated by an example in Chapter 7, the function to be approximated may be defined implicitly, for example, by a procedure which numerically solves a boundary value problem. In such a case, the accuracy of the function definition may be limited and therefore the approximations can be generated only to that limited accuracy.

2.5.3 Compiler

Maple's `Compiler:-Compile` command can convert "numerical" Maple procedures to native code. The Maple procedure is first translated to C language code and compiled by an external C compiler, and then the compiled code is dynamically linked into Maple so that we can call the procedure directly in Maple.

All the computation in a compiled procedure is carried out in hardware floating point precision, as is the case when using `evalhf`. The compiled procedure could possibly run hundreds of times faster than the original Maple code. The compiled procedure also runs faster than executing the code in `evalhf` mode in most cases.

Unfortunately there is only a very limited subset of Maple procedures that can be translated by the compiler. Note that none of the following can be translated: nested procedures, modules, exception handling (other than merely raising an exception), procedures that return arrays or allocate new arrays, and procedures that call other procedures about which insufficient type information can be obtained at compile time [7]. For example, the Maple procedures for evaluating the Bessel functions cannot be compiled using the `Compiler:-Compile` command. But the approximations generated by our toolkit are compilable and thus are able to improve the speed of evaluations.

Chapter 3

Polynomial Approximations

There are two common types of approximation techniques for univariate functions: polynomial and rational approximations. In this chapter we discuss three polynomial approximation techniques, including Taylor series, minimax polynomial approximation, and Chebyshev series, and introduce their corresponding commands in Maple.

3.1 Taylor Approximations

3.1.1 Taylor Series

A Taylor series is a series expansion of a function about a point. If a function $f(x)$ is infinitely differentiable in a neighborhood of a point a, then the Taylor series of $f(x)$ about point $x = a$ can be given in the following fashion:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \cdots \quad (3.1)$$

where $n!$ is the factorial of n and $f^{(n)}(a)$ denotes the nth derivative of $f(x)$ at point a.

If $a = 0$ in the particular case, then the expansion is known as a Maclaurin series [11].

Taylor series expansion is one of the important techniques for univariate function approximation because $f(x)$ can often be approximated to a specified accuracy by the partial sums of the series in equation (3.1), i.e., the order k Taylor series expansion, with sufficiently many terms k.

3.1.2 The `taylor` Command

Maple has the command `taylor(f(x),x=a,n)` which computes the Taylor series expansion of $f(x)$ about the point a to the expansion order n, namely

$$\sum_{k=0}^{n-1} \frac{f^{(n)}(a)}{n!}(x-a)^n + O((x-a)^n). \tag{3.2}$$

The third argument n (the truncation order) may be omitted in which case the order is determined by the environment variable `Order` (which is set to 6 by default).

If we need to convert the Taylor series to ordinary polynomial form, we can use the command `convert(s, polynom)`, where `s` is the Taylor series in equation (3.2).

Maple has the more general command `series(f(x),x=a,n)`. If the function $f(x)$ has a Taylor series expansion about the point $x = a$ then the result of the `series` command is identical to `taylor`. Otherwise, a generalized (non-Taylor) series expansion may be returned.

3.2 Minimax Polynomial Approximations

It is often desirable in applications to minimize the maximum absolute or relative error of a polynomial approximation in order to reduce the computational expense of evaluation. The polynomial of best approximation of a given degree, called the minimax polynomial, is defined to be the one that has the smallest maximum deviation from the true function.

According to Chebyshev's theorem on polynomial approximations [5], we know that if $u(x)$ denotes a function continuous in a closed, finite interval $[a, b]$, $v(x)$ denotes a function continuous and nonzero in $[a, b]$, and V_n denotes the set of polynomials of degree$\leq n$, then there exists a unique polynomial $p_n^*(x)$, i.e., the minimax polynomial, in V_n such that

$$\max_{a \leq x \leq b} \left| \frac{p_n^*(x)}{v(x)} - u(x) \right| = \min_{p_n(x)\ in\ V_n} \max_{a \leq x \leq b} \left| \frac{p_n(x)}{v(x)} - u(x) \right|. \tag{3.3}$$

Generally there are two different $p_n^*(x)$ of interest that approximate $f(x)$ with the two error types in the interval $[a, b]$:

- If $v(x) = 1$ and $u(x) = f(x)$ in equation (3.3), the minimax polynomial $p_n^*(x)$ is with respect to absolute error.

- If $v(x) = f(x)$ and $u(x) = 1$ in equation (3.3), then $p_n^*(x)$ is with respect to relative error.

Maple has a `minimax` command in the **numapprox** package which may be used to compute the minimax polynomial. In our project, we will use the computationally less expensive `chebyshev` command which computes a "near-minimax" polynomial approximation as discussed in the next section.

3.3 Chebyshev Approximations

3.3.1 Chebyshev Polynomials

In mathematics, there is a well-known Chebyshev's differential equation which is defined as:

$$(1 - x^2)y'' - xy' + n^2 y = 0 \tag{3.4}$$

where n is a real number.

If n is a non-negative integer, the solutions to this equation are often referred to as Chebyshev polynomials of the first kind, or Chebyshev polynomials, denoted by $T_n(x)$. The Chebyshev polynomials here are a sequence of orthogonal polynomials. There are two common ways to define Chebyshev polynomials $T_n(x)$ of degree n. One way is by the recurrence relation:

$$\begin{aligned} T_0(x) &= 1, \quad T_1(x) = x, \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x), \quad n = 1, 2, 3, ... \end{aligned} \tag{3.5}$$

An alternative way is by the trigonometric identity:

$$T_n(x) = cos(n \arccos x) \quad \text{for } -1 \leq x \leq 1. \tag{3.6}$$

3.3.2 Chebyshev Series

As described in [5], if a function $f(x)$ has a continuous first derivative in $[-1, 1]$, then it has a Chebyshev series expansion:

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x) = \frac{1}{2}a_0 T_0(x) + a_1 T_1(x) + a_2 T_2(x) + \cdots \tag{3.7}$$

which converges uniformly in $[-1, 1]$, where

$$a_k = \frac{2}{\pi} \int_{-1}^{1} f(x) T_k(x) (1 - x^2)^{-\frac{1}{2}} dx. \tag{3.8}$$

Since the Chebyshev series is "infinite", we usually use a truncated Chebyshev series of degree n as the approximation for $f(x)$ instead. In [8] it is proved that if $q_n(x)$ denotes the truncated Chebyshev series of degree n and $p_n^*(x)$ denotes the best minimax polynomial approximation of degree n then

$$\|f - q_n\|_\infty \leq \gamma_n \|f - p_n^*\|_\infty \tag{3.9}$$

where $[a, b]$ is the interval of the approximation, γ_n are known as the Lebesgue constants and $\gamma_n < 5$ for $n \leq 1000$.

Therefore we conclude that the truncated Chebyshev series of degree n closely approximates the minimax polynomial approximation of $f(x)$. Since it is computationally less expensive to compute the truncated Chebyshev series (via a fast discrete cosine transform), it is our method of choice.

The main difference between Taylor series and Chebyshev series is that the former expansions are often convenient for theoretical work but less useful for practical applications, while the latter is better for numerical computation since the Chebyshev polynomials, $T_n(x)$, are better behaved than the monomials, x^n, on an interval $[a, b]$ of the real line. As stated in [10], since the Chebyshev series has superior performance to that of the Taylor series in terms of convergence, it is expected to exhibit better performance in terms of approximation error, when the approximation interval is large. Hence we will use Chebyshev series for polynomial approximation in our toolkit.

3.3.3 The `chebyshev` Command

Maple's **numapprox** package has the command `chebyshev(f(x),x=a..b,eps)` which computes a truncated Chebyshev series expansion, in terms of the Chebyshev polynomials $T_k(x)$, for a given function $f(x)$ on a specified interval $[a, b]$, in the form:

$$c_0 T_0(v) + c_1 T_1(v) + \cdots + c_n T_n(v) \tag{3.10}$$

where v is an expression in x of the form $v = Dx + E$ representing the linear transformation of the specified interval $[a, b]$ to the standard interval $[-1, 1]$.

Such an approximation is computed to a specified accuracy `eps`, which is achieved by discarding all the terms with coefficients smaller (in magnitude) than a particular tolerance criterion.

We may use the Maple command `T(k,x)` in the **orthopoly** package to expand the approximation (3.10) into standard polynomial form in terms of powers of x, although such a conversion is generally not advisable for numerical evaluation.

In our package we added three optional arguments `('output=Vector', size, FixedSize)` to the `numapprox:-chebyshev` procedure.

If `'output=Vector'` is specified then the returned value from `chebyshev` is a vector representation of the Chebyshev series. The vector representation is an expression sequence $(coef, v)$ where $coef$ is a Maple vector of the floating point coefficients $c_k, k = 0 \ldots n$ in the corresponding Chebyshev series (3.10), and v is the linear expression in x described above. The vector representation can be evaluated more efficiently and accurately than the sum-of-products form in (3.10).

If we want to evaluate the vector representation at some point x, for example, $x = 2.5$, we can use the Maple command `chebeval` in the **numapprox** package for this purpose:

$$> numapprox : -chebeval(coef, eval(v, x = 2.5)); . \tag{3.11}$$

The integer argument `size` is used to specify the dimension of the local array allocated for Chebyshev coefficients, which can specify the number of function evaluations to be used. Note that the number of function evaluations actually used is a number of the form $2 \times 3^M + 1$ due to the algorithm invoked in `numapprox:-chebyshev` and therefore reasonable values might be 163, 487, 1459, or 4375. `size` is initialized to 487 by default; however, if the convergence is not achieved and `size`$\leq$487 and `FixedSize` is false, then `size` is reset to 4375.

The boolean argument `FixedSize`, which is false by default, may be used to specify whether `size` may be increased in case of non-convergence. If `FixedSize` is true then the number of function evaluations is limited to the value specified by `size`.

Note that the `numapprox:-chebyshev` command does not break an interval into subintervals, so it computes a single polynomial approximation on the given interval no matter how large the degree of the approximation is. Due to the near-minimax property of the truncated Chebyshev series, the degree of the approximation computed by the `chebyshev` command is a very good estimate of the minimum degree required to approximate the function on the given interval to the specified accuracy. In other words, it is a measure of the difficulty of approximating the function.

As will be discussed later, our toolkit will break intervals into subintervals as necessary to achieve approximations with degrees that do not exceed user-specified bounds.

Chapter 4

Rational Approximations

Another class of very important approximation techniques for univariate functions is rational approximations. In this chapter we discuss three techniques for rational approximation: Padé approximation, minimax rational approximation, and Chebyshev-Padé approximation. In doing so, we will compare polynomial and rational approximations and describe how they are useful for developing our toolkit.

4.1 The Padé Method

4.1.1 Padé Approximation

The method of Padé approximation is used widely in computer calculations. It approximates a function $f(x)$ by a ratio of two polynomials $R_{m,n}(x) = \frac{P_m(x)}{Q_n(x)}$ such that the Taylor series expansion of $R_{m,n}(x)$ has maximal initial agreement with the Taylor series expansion of $f(x)$. If we let $f(x)$ denote a function having the Maclaurin series expansion $f(x) = \sum_{k=0}^{\infty} a_k x^k$, from [5] we know that the Padé approximation of order (m, n) to $f(x)$ is defined to be a rational function $R_{m,n}(x)$ of the form:

$$R_{m,n}(x) = \frac{P_m(x)}{Q_n(x)} = \frac{p_0 + p_1 x + p_2 x^2 + \cdots + p_m x^m}{q_0 + q_1 x + q_2 x^2 + \cdots + q_n x^n} \tag{4.1}$$

where $P_m(x)$ and $Q_n(x)$ are polynomials whose coefficients satisfy the set of equations :

$$
\begin{aligned}
a_0 q_0 &= p_0 \\
a_1 q_0 + a_0 q_1 &= p_1 \\
&\vdots \\
a_m q_0 + a_{m-1} q_1 + \cdots + a_{m-n} q_n &= p_m \\
a_{m+1} q_0 + a_m q_1 + \cdots + a_{m-n+1} q_n &= 0 \\
a_{m+2} q_0 + a_{m+1} q_1 + \cdots + a_{m-n+2} q_n &= 0 \\
&\vdots \\
a_{m+n} q_0 + a_{m+n-1} q_1 + \cdots + a_m q_n &= 0.
\end{aligned} \tag{4.2}
$$

Note that these equations are derived from the defining property:

$$
f(x) \cdot Q_n(x) - P_m(x) = O(x^{m+n+1}). \tag{4.3}
$$

4.1.2 The `pade` Command

Maple's **numapprox** package has the command `pade(f(x),x=a,[m,n])` which computes a Padé approximation of degree (m, n) for the function $f(x)$. It first expands $f(x)$ in a Taylor (or Laurent) series about the point $x = a$ to order $m+n+1$, and then computes the Padé rational approximation [7]. If n is 0 or not specified, then it simply computes the Taylor polynomial of degree m.

4.2 The Minimax Method

4.2.1 Minimax Rational Approximations

Similar to Chebyshev's theorem on polynomial approximation introduced in section 3.2, we have Chebyshev's theorem on rational approximation. Let us continue using $u(x)$ and $v(x)$ defined in equation (3.3). We also let $V_{m,n}$ denote the family of rational functions which is expressible as an irreducible fraction $\frac{P_m(x)}{Q_n(x)}$, where $Q_n(x) \neq 0$ in $[a, b]$. Then [5] states that there exists a unique rational function $R^*_{m,n}(x)$ in $V_{m,n}$ such that

$$
\max_{a \le x \le b} \left| \frac{R^*_{m,n}(x)}{v(x)} - u(x) \right| = \min_{R_{m,n}(x) \; in \; V_{m,n}} \max_{a \le x \le b} \left| \frac{R_{m,n}(x)}{v(x)} - u(x) \right|. \tag{4.4}
$$

As with polynomial approximations, there are two corresponding cases of minimax absolute error and minimax relative error:

- When $v(x) = 1$ and $u(x) = f(x)$, we have

$$\left| \frac{R_{m,n}(x)}{v(x)} - u(x) \right| = |R_{m,n}(x) - f(x)|$$

  which approximates $f(x)$ with minimax absolute error in $[a, b]$.

- When $v(x) = f(x) \neq 0$ and $u(x) = 1$, we have

$$\left| \frac{R_{m,n}(x)}{v(x)} - u(x) \right| = \left| \frac{R_{m,n}(x) - f(x)}{f(x)} \right|$$

  which approximates $f(x)$ with minimax relative error in $[a, b]$.

4.2.2 The `minimax` Command

The `minimax` command in the **numapprox** package of Maple can be used to compute the best minimax rational approximation $\frac{p(x)}{q(x)}$ of degree (m, n) for a specified function $f(x)$ on the interval $[a, b]$, where $p(x)$ is of degree$\leq m$ and $q(x)$ is of degree$\leq n$. The $\frac{p(x)}{q(x)}$ returned by `numapprox:-minimax` is the one that minimizes

$$w(x) \left\| f - \frac{p}{q} \right\|_\infty \tag{4.5}$$

where $w(x)$ is a positive weight function, which is set to 1 corresponding to absolute error measure by default and set to $\left| \frac{1}{f(x)} \right|$ for relative error measure if specified.

As mentioned in the case polynomial approximation, in our project we avoid using the `minimax` method. For rational approximations, the Chebyshev-Padé method is computationally less expensive and yields similar results.

4.3 The Chebyshev-Padé Method

4.3.1 Chebyshev-Padé Approximation

Chebyshev-Padé approximation is a combination of Chebyshev approximation and Padé-like rational approximation. It approximates a given function $f(x)$ as a rational function in which the numerator and denominator are both in Chebyshev series

forms as follows:

$$\frac{\sum_{k=0}^{m} a_k T_k(x)}{\sum_{k=0}^{n} b_k T_k(x)} \tag{4.6}$$

where $T_k(x)$ is the Chebyshev polynomial of degree k. Note that the Chebyshev-Padé approximation of $f(x)$ is defined by the property that the Chebyshev series expansion of the rational function in (4.6) has maximal initial agreement with the Chebyshev series expansion of $f(x)$. In standard cases, the expansions agree up to the term of degree $m+n$.

As proved by equation (3.9), the Chebyshev series has a very good near-minimax performance and small, uniform errors. Likewise the Chebyshev-Padé approximation described is a near-minimax rational approximation. This is achieved by using the method described in [6] as implemented in the `chebpade` command in Maple. As previously mentioned, the Chebyshev-Padé method is a relatively inexpensive method.

4.3.2 The `chebpade` Command

Given a function $f(x)$, the command `chebpade(f(x),x=a..b,[m,n])` in the **numapprox** package of Maple computes a Chebyshev-Padé approximation of degree (m,n) on the interval $[a,b]$.

Similar to `numapprox:-chebyshev`, we added one more argument `'output=Vector'` to `numapprox:-chebpade` so that when we use `chebpade` it returns a vector representation $(numcoef, dencoef, v)$ of the Chebyshev-Padé approximation. Both of $numcoef$ and $dencoef$ are Maple vectors of floating point coefficients of dimensions $m+1$ and $n+1$ respectively, such that the corresponding Chebyshev-Padé approximation is as in (4.6), where $a = numcoef$, $b = dencoef$, and v is a linear expression in x exactly as described in section 3.3.3.

As in the case of `numapprox:-chebyshev`, if $f(x)$ is an expression, we may numerically evaluate the approximation at some point x, say, $x = 2.5$, by the Maple command `numapprox:-chebeval`

$$\begin{aligned} > \quad & numapprox := -chebeval(numcoef, eval(v, x = 2.5)) \\ & /numapprox := -chebeval(dencoef, eval(v, x = 2.5)); . \end{aligned}$$

Using the method described in [6], the `numapprox:-chebpade` command first expands $f(x)$ in a Chebyshev series on the interval $[a,b]$ which is converted to a

power series with the same coefficients. It computes a Padé approximation for the power series and then transforms to the Chebyshev-Padé rational approximation on the specified interval $[a, b]$. Since our toolkit always uses `numapprox:-chebyshev` to estimate the degree of approximation, it is more time-efficient if we can use the output from `numapprox:-chebyshev` as input to `numapprox:-chebpade` so that we can avoid re-computing the Chebyshev series in the first step of computation in `numapprox:-chebpade`. Hence we altered the `chebpade` command so that it can accept as an argument the vector representation sequence $(coef, var)$ of a Chebyshev series, as returned by `chebyshev` with the option `'output=Vector'`.

Note that if `Digits`$\leq$`evalhf(Digits)` then we will specify the option `'datatype = float[8]'` for the vectors $numcoef$ and $dencoef$ in order to have efficient processing by procedure `numapprox:-chebeval`. Specifically, the type `float[8]` in Maple refers to hardware floating point numbers.

4.4 Polynomial Approximation vs. Rational Approximation

In our toolkit we first expand a given bivariate function in a tensor product series so that we can reduce a bivariate approximation problem to a sequence of univariate approximation problems. Then we use polynomial or rational approximations for the univariate functions.

To approximate a function within the specified (fixed) accuracy on a given interval, we consider two choices: polynomial approximation and rational approximation. Since the error tolerance is the same for the two methods in such case, we would prefer to use the one with faster speed for evaluation.

As shown by experiments in [9], it takes essentially the same amount of time to evaluate a polynomial of degree n as to evaluate a rational function of degree $[\frac{n}{2}, \frac{n}{2}]$. If the degree of a rational function can be reduced, which is possible for some functions, we will then be able to evaluate the approximation faster by using rational approximations. Generally, if the function being approximated is polynomial in nature, such as the Bessel function of the first kind $J_v(x)$, using polynomial approximation should be a better choice since rational approximation is of no benefit over polynomial approximation in such cases. However, in the case when the function is not polynomial in nature, such as the modified Bessel function of the second kind $K_v(x)$ which has a singularity at $x = 0$, a rational approximation can achieve a better result with a smaller total degree.

In our toolkit, by default we attempt to use rational approximations if possible unless the user specifies `'rational=false'` as an optional argument in which case only polynomial approximations will be used. In the default case, we prefer to use

rational approximation even if the total degree is a little higher than that of polynomial approximation. This is because in our data structure (described later) for storing the various approximations, there is an efficiency advantage if all approximations are similar (rather than having, for example, a rational approximation of degree $(4, 4)$ in one case and a polynomial approximation of degree 7 in another case). If we fail to achieve the accuracy by using rational approximation in default case, we will revert to use polynomial approximation.

Chapter 5

Approximation Using Tensor Product Series

Traditionally, as in the case of univariate function approximations, the approximations for bivariate or multivariate functions are achieved by fixing the basis functions first, such as trigonometric basis functions or polynomial basis functions. Frederick W. Chapman introduced a method in his PhD thesis [3] to approximate functions by using tensor product series, and he gave the name Geddes series to the particular tensor product series generated by his method. Orlando A. Carvajal summarized the relevant work and explained how to use tensor product series to approximate symmetric functions in his Masters thesis [2], which dealt with the application to multidimensional integration.

We will start by giving a brief overview of definitions of tensor products and the splitting operator. Many of the concepts presented here appear in [1], [3] and [2].

5.1 Tensor Products

A tensor product is a finite sum of the terms, where each term is a product of univariate functions:

$$s_n(x,y) = \sum_{i=1}^{n} g_i(x)h_i(y). \tag{5.1}$$

It is easy to see that many bivariate functions can be written in terms of tensor products, for example,

$$e^{x+y} = e^x e^y \tag{5.2}$$

$$\sin(x+y) = \sin x \cos y + \cos x \sin y. \tag{5.3}$$

The minimum number of terms among all the equivalent expressions of a tensor product is called the *rank* of the tensor product. A tensor product is *natural* if the factors of each term can be derived from the original function by a finite linear combination of linear functionals. Hence in the above examples, (5.2) has rank 1 and (5.3) has rank 2, and they are both natural tensor products.

5.2 The Splitting Operator

The splitting operator $\Upsilon_{(a,b)}$ at the point (a, b) is defined as in [3]:

$$\Upsilon_{(a,b)} f(x,y) = \lim_{\hat{x}\to a}\left(\lim_{\hat{y}\to b}\left(\frac{f(x,\hat{y})\cdot f(\hat{x},y)}{f(\hat{x},\hat{y})}\right)\right). \tag{5.4}$$

If f is continuous in the interval and $f(a, b) \neq 0$, we then have the splitting operator as

$$\Upsilon_{(a,b)} f(x,y) = \frac{f(x,b)\cdot f(a,y)}{f(a,b)}. \tag{5.5}$$

The point (a, b) here is called a *splitting point.*

There are two important properties of $\Upsilon_{(a,b)} f(x, y)$:

- $\Upsilon_{(a,b)} f(x, y)$ interpolates $f(x, y)$ on the lines $x = a$ and $y = b$. Thus $\Upsilon_{(a,b)} f(a, y)$ $\equiv f(a, y)$ and $\Upsilon_{(a,b)} f(x, b) \equiv f(x, b)$.
- If there is a value $\bar{x}$ such that $f(\bar{x}, y) \equiv 0$, it follows that $\Upsilon_{(a,b)} f(\bar{x}, y) \equiv 0$ as well. Likewise for a $\bar{y}$ such that $f(x, \bar{y}) \equiv 0$.

5.3 Tensor Product Series

Not all bivariate functions $f(x, y)$ can be expressed as a tensor product of finite rank such as in examples (5.2) and (5.3), whereas we can approximate f on the region $R = [a, b] \times [c, d]$ by a tensor product series $s_n(x, y)$ as in equation (5.1):

$$f(x,y) = s_n(x,y) + r_n(x,y) \tag{5.6}$$

where $r_n(x, y)$ is the n-th remainder term for the series. $|r_n(x, y)|$ is also called the absolute error for the approximation s_n of f on R.

Orlando A. Carvajal, Frederick W. Chapman and Keith O. Geddes described an algorithm NTPS to compute a tensor product series approximation of a bivariate function f with absolute error $\epsilon > 0$ on region R in their ISSAC paper [2]. Their interest in that paper is multidimensional integration. They first approximated the integrand $f(x, y)$ by a tensor product series expansion and then integrated term by term. Our interest is to apply their algorithm to generate an approximation for a bivariate function by a tensor product series. The algorithm NTPS works as follows:

1. Define the initial remainder by $r_0 := f$ and initialize the counter $i := 1$.
2. While the uniform error satisfies $\|r_{i-1}\|_\infty > \epsilon$, iterate the following two steps:
   (a) Choose a splitting point $(a_i, b_i) \in R$ such that $|r_{i-1}(a_i, b_i)| = \|r_{i-1}\|_\infty$.
   (b) Let $r_i := r_{i-1} - \Upsilon_{(a_i,b_i)} r_{i-1}$ and $i := i + 1$.
3. After exiting the loop, let $n := i - 1$ and $s_n := f - r_n$. Return s_n as the desired series approximation of f with uniform error $\|r_n\|_\infty \leq \epsilon$ over R.

The resulting series s_n is the natural tensor product series expansion of f to n terms with respect to the splitting points $\{(a_i, b_i)\}_{i=1}^n$.

The Algorithm NTPS in its most general form as presented above has one expensive step. Specifically, step 2a calls for a two-dimensional search of the region $[a, b] \times [c, d]$ for the point where the numerical maximum is attained for the absolute remainder $|r_{i-1}(x, y)|$. It turns out to be sufficient to replace step 2a by two one-dimensional searches (or a single one-dimensional search in the case of symmetric functions). Moreover, for the one-dimensional searches it is sufficient to sample a discrete set of points defined by the midpoints of previous splitting points. In doing so we can achieve more efficient computation; however, this is based on computational experience and some proofs are still needed here.

We now present a simple example from [2] which illustrates how to reduce the bivariate function approximation problem to a sequence of univariate function approximation problems.

Example 5.1

The following function is defined on the unit square $[0, 1] \times [0, 1]$:

$$f(x, y) = e^{x^2 y^2} \cos(x + y).$$

The tensor product series to three terms is as follows:

$$s_3(x,y) = \sum_{i=1}^{3} c_i g_i(x) h_i(y)$$

where the splitting points are (a,a) for a =1, 0, 0.616336, and

$$\begin{aligned} c_1 &= -0.884017, \\ g_1(x) &= e^{x^2}\cos(x+1). \\ h_1(y) &= e^{y^2}\cos(y+1); \end{aligned}$$

$$\begin{aligned} c_2 &= 0.794868, \\ g_2(x) &= \cos x + 0.477636\, e^{x^2}\cos(x+1), \\ h_2(y) &= \cos y + 0.477636\, e^{y^2}\cos(y+1); \end{aligned}$$

$$\begin{aligned} c_3 &= -9.83284, \\ g_3(x) &= e^{0.379870\,x^2}\cos(x+0.616336) - \\ &\quad 0.356576\, e^{x^2}\cos(x+1) - 0.623342\ \cos x, \\ h_3(y) &= e^{0.379870\,y^2}\cos(y+0.616336) - \\ &\quad 0.356576\, e^{y^2}\cos(y+1) - 0.623342\ \cos y. \end{aligned}$$

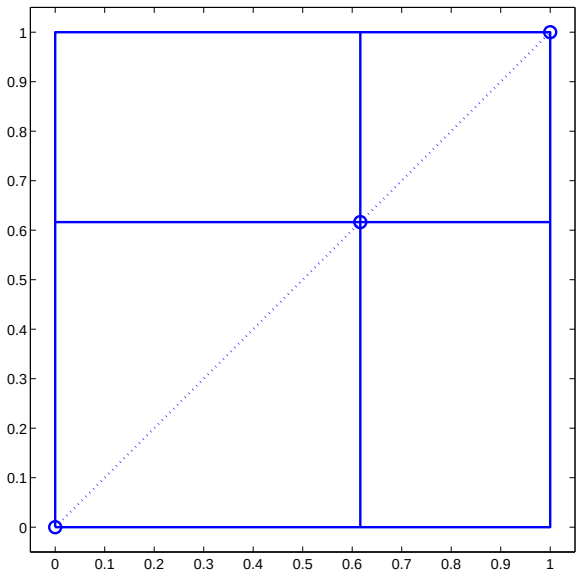

Figure 5.1: Lines of Interpolation for 3 Splitting Points

As shown in Figure 5.1, all the splitting points in Example 5.1 are located on the diagonal $y = x$. However, due to the interpolating properties of the splitting operator described in section 5.2, unlike ordinary interpolation which produces zero error exactly at the interpolation points, with our tensor product series we get zero error at all points on the lines passing through the splitting points. With this strong interpolation property, the approximation $s_3(x, y)$ agrees exactly with the function $f(x, y)$ at all points on the horizontal and vertical lines passing through the three splitting points. Therefore, we find that even with only three splitting points, the maximum error over the unit square for the approximation in Example 5.1 is $\|f - s_3\|_\infty \approx 3.0 \times 10^{-3}$.

5.4 Vector-Matrix Representation

5.4.1 Symmetric Case

The series generated by algorithm NTPS in section 5.3 has the following form:

$$s_n(x, y) = \sum_{i=1}^{n} \left(c_i \left(\sum_{j=1}^{i} k_{i,j} f(x, b_j) \right) \left(\sum_{j=1}^{i} l_{i,j} f(a_j, y) \right) \right) \tag{5.7}$$

where c_i, $k_{i,j}$, $l_{i,j} \neq 0$ are real coefficients, and (a_i, b_i) are the splitting points.

In the symmetric case, i.e., $f(x, y) = f(y, x)$ for all $(x, y) \in R$, we require the splitting points to be either on the diagonal $y = x$ or selecting a point (a, b) as a splitting point implies that the next splitting point must be (b, a). According to the symmetry property of this criterion for selecting splitting points, we have $\{a_i\}_{i=1}^n = \{b_i\}_{i=1}^n$, i.e., the set of x-coordinates of the splitting points is the same as the set of y-coordinates. Hence for the symmetric case, we can present the series in (5.7) by using matrices and vectors as

$$s_n(x, y) = V^T(x) \cdot L^T \cdot P \cdot D \cdot L \cdot V(y) \tag{5.8}$$

where

- $V(x)$ is the column vector of dimension n whose elements are the univariate functions $f(x, a_i)$.
- D is an $n \times n$ diagonal matrix whose diagonal elements correspond to the coefficients $c_i = 1/r_{i-1}(a_i, b_i)$.
- P is an $n \times n$ permutation matrix that allows the coefficients $k_{i,j}$ to be obtained from the coefficients $l_{i,j}$ via $[k_{i,j}] = P \cdot [l_{i,j}]$.

- $L = [l_{i,j}]$ is an $n \times n$ unit lower triangular matrix.

More details can be found in [2] and [1].

5.4.2 Asymmetric Case

For general asymmetric functions, we have the matrix-vector representation of (5.8) rewritten as:

$$s_n(x, y) = V^T(x) \cdot U \cdot D \cdot L \cdot W(y) \tag{5.9}$$

where

- $V(x)$ is the column vector of dimension n whose elements are the univariate functions $f(x, b_i)$.
- $U = [k_{i,j}]^T$ is an $n \times n$ unit upper triangular matrix.
- D is an $n \times n$ diagonal matrix whose diagonal elements correspond to the coefficients $c_i = 1/r_{i-1}(a_i, b_i)$.
- $L = [l_{i,j}]$ is an $n \times n$ unit lower triangular matrix.
- $W(y)$ is the column vector of dimension n whose elements are the univariate functions $f(a_i, y)$.

In particular, if we are working with symmetric functions then $U = L^T$ and $V(x)$ is identical to $W(y)$ with just the variable name changed, i.e., $W(y) = V(y)$.

We will adopt this updated vector-matrix representation in our package for approximation of general functions, including both symmetric and asymmetric cases. Note that recently Frederick W. Chapman and Keith O. Geddes have theoretically proved the convergence of the tensor product series expansion for a symmetric positive definite kernel, where the kernel means the bivariate function $f(x, y)$ [see Appendix A]. We do not have mathematical proofs for the convergence in the general case yet; however, the practical results of our toolkit show the promise of the method as will be seen in Chapter 7.

Now we show an example of how to represent the tensor product series generated in Example 5.1 by vector-matrix form.

Example 5.2

As defined in Example 5.1, we have the following function on the unit square $[0,1] \times [0,1]$:

$$f(x,y) = e^{x^2 y^2} \cos(x+y),$$

and the tensor product series to three terms:

$$s_3(x,y) = \sum_{i=1}^{3} c_i g_i(x) h_i(y),$$

where the splitting points are (a,a) for a =1, 0, 0.616336.

By converting $s_3(x,y)$ to vector-matrix representation we get:

$$V(x) = \begin{bmatrix} e^{x^2}\cos(x+1) \\ \cos x \\ e^{0.379870\,x^2}\cos(x+0.616336) \end{bmatrix},$$

$$U = \begin{bmatrix} 1 & 0.477636 & -0.356576 \\ 0 & 1 & -0.623342 \\ 0 & 0 & 1 \end{bmatrix},$$

$$D = \begin{bmatrix} -0.884017 & 0 & 0 \\ 0 & 0.794868 & 0 \\ 0 & 0 & -9.83284 \end{bmatrix},$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.477636 & 1 & 0 \\ -0.356576 & -0.623342 & 1 \end{bmatrix},$$

$$W(y) = \begin{bmatrix} e^{y^2}\cos(y+1) \\ \cos y \\ e^{0.379870\,y^2}\cos(y+0.616336) \end{bmatrix}.$$

We can obviously see that $W(y)$ is the same as $V(x)$ with x replaced by y, and U is the transpose of L, i.e., $U = L^T$, in this example. This is simply because $f(x,y)$ is a symmetric function in this case.

Now we can compute the tensor product series $s_3(x,y)$, and $s_1(x,y)$, $s_2(x,y)$ as well, by using the above vector-matrix representation as follows:

Approximation with rank 1:

$$\begin{aligned} s_1(x,y) &= V^T(x)[1]\cdot U[1,1]\cdot D[1,1]\cdot L[1,1]\cdot W(y)[1] \\ &= -0.884017\, e^{x^2}\cos(x+1)\, e^{y^2}\cos(y+1). \end{aligned}$$

Approximation with rank 2:

$$\begin{aligned} s_2(x,y) &= V^T(x)[1..2]\cdot U[1..2,1..2]\cdot D[1..2,1..2]\cdot L[1..2,1..2]\cdot W(y)[1..2] \\ &= e^{y^2}\cos(y+1)\left(-0.702679\, e^{x^2}\cos(x+1)+0.379658\ \cos x\right)+ \\ &\quad \cos y\left(0.379658\, e^{x^2}\cos(x+1)+0.794868\ \cos x\right). \end{aligned}$$

Approximation with rank 3:

$$s_3(x,y) = V^T(x)\cdot U\cdot D\cdot L\cdot W(y)$$

$$= e^{y^2}\cos(y+1)$$

$$\left(-1.95289\, e^{x^2}\cos(x+1)-1.80587\ \cos x+3.50615\, e^{0.379870\,x^2}\cos(x+0.616336)\right)$$

$$+\cos y$$

$$\left(-1.80587\, e^{x^2}\cos(x+1)-3.02573\ \cos x+6.12922\, e^{0.379870\,x^2}\cos(x+0.616336)\right)$$

$$+e^{0.379870\,y^2}\cos(y+0.616336)$$

$$\left(3.50615\, e^{x^2}\cos(x+1)+6.12922\ \cos x-9.83284\, e^{0.379870\,x^2}\cos(x+0.616336)\right).$$

5.4.3 Benefits of the Vector-Matrix Representation

There are several reasons why we choose the vector-matrix representation rather than the explicit expression of tensor product series $s_n(x,y)$ as in (5.7). We can see that the vector-matrix representation improves the time efficiency and accuracy of the approximation from the following points of view:

- When using the explicit expression, the size of the expression grows exponentially as the number of terms increases. However, we can avoid such expression growth by adopting the representation in vector-matrix form.

- The vector-matrix representation reduces the cost of handling the complex expressions for r_n and s_n from $O(n^2)$ to $O(n)$ evaluations of the original function $f(x, y)$.
- Applying built-in linear algebra operations for vector-matrix multiplication and inner product make the computation more efficient.
- It is usually impossible to evaluate to full accuracy with the explicit expression. Because the remainders r_i get successively smaller, the $i-$th diagonal elements $D[i, i]$ in matrix D, which are equal to $\frac{1}{r_i}$, get larger as i increases (though this does not really show up in Example 5.2 because of the small number of terms). As a result we will produce a sum of terms with large values adding up to a relatively small final result which leads to a loss of accuracy. However, using vector-matrix representation can prevent this problem and maintain a stable evaluation as will be seen in Chapter 6.

Note that in our package we evaluate the expressions of x and y, which only appear in the vectors $V(x)$ and $W(y)$, to numerical values before performing the vector-matrix multiplications. We will show the details of coding in Chapter 6.

5.5 Translation of Variables

The original implementation of generating tensor product series for a bivariate function f was specifically on the unit square region $R = [0, 1] \times [0, 1]$. Hence there has been some modification to the code so that the new implementation can accept any region $R = [a, b] \times [c, d]$ and automatically transform the specified region to the unit square $[0, 1] \times [0, 1]$. After the computation is done, the inverse transformation is then used to get back to the original region.

If the function to be approximated is $f(x, y)$ and we want to translate the original variables (x, y) on the region $[a, b] \times [c, d]$ to new variables (s, t) on the unit square $[0, 1] \times [0, 1]$, then it works as follows:

1. Apply the change of variables which maps the region $[0, 1]\times[0, 1]$ to $[a, b]\times[c, d]$

$$\begin{aligned} x &= a \times (1 - s) + b \times s, \\ y &= c \times (1 - t) + d \times t; \end{aligned} \tag{5.10}$$

where $0 \leq s \leq 1$ and $0 \leq t \leq 1$.

2. Now we have a modified function $g(s,t)$ transformed from $f(x,y)$ such that $g(s,t) = f((b-a)\times s+a,(d-c)\times t+c)$. Use $g(s,t)$ directly in the old code to generate the tensor product series on $[0,1]\times[0,1]$.

3. Once we have created the approximation, apply the inverse transformation

$$\begin{aligned} s &= \frac{x-a}{b-a}, \\ t &= \frac{y-c}{d-c}; \end{aligned} \tag{5.11}$$

which maps the region $[a,b]\times[c,d]$ to $[0,1]\times[0,1]$. Then we have the tensor product series in terms of (s,t) changed back to the approximation with respect to (x,y).

By using the translation of variables as described above, we are now able to compute tensor product series expansions for any given regions.

5.6 `TensorProductInt`

The Maple module `TensorProductInt` developed by Frederick W. Chapman, Orlando A. Carvajal and Keith O. Geddes can be used to generate tensor product series for a given function $f(x,y)$.

As mentioned in section 5.4.2, the old version of the module only worked for symmetric functions and for the unit square region; however, it can now deal with asymmetric functions on any region since it has been recently updated. A new routine has been added in the module to deal with general cases. If the function to be approximated is symmetric, this Maple module will call the subroutines which were already in the old one; otherwise, the module will use the newly developed code for the asymmetric case.

The flow chart of the module `TensorProductInt` is shown in Figure 5.2.

We can use the command `` `evalf/int/TensorProductInt`:-TensorInt2D `` to invoke the new procedure `TensorInt2D`. This procedure accepts as input arguments the given bivariate function `expr`, the ranges of the two variables in the list `region`, and the requested accuracy `epsilon`, and then returns the following values:

`TPSeriesRep` The natural tensor product series approximation of `expr`.

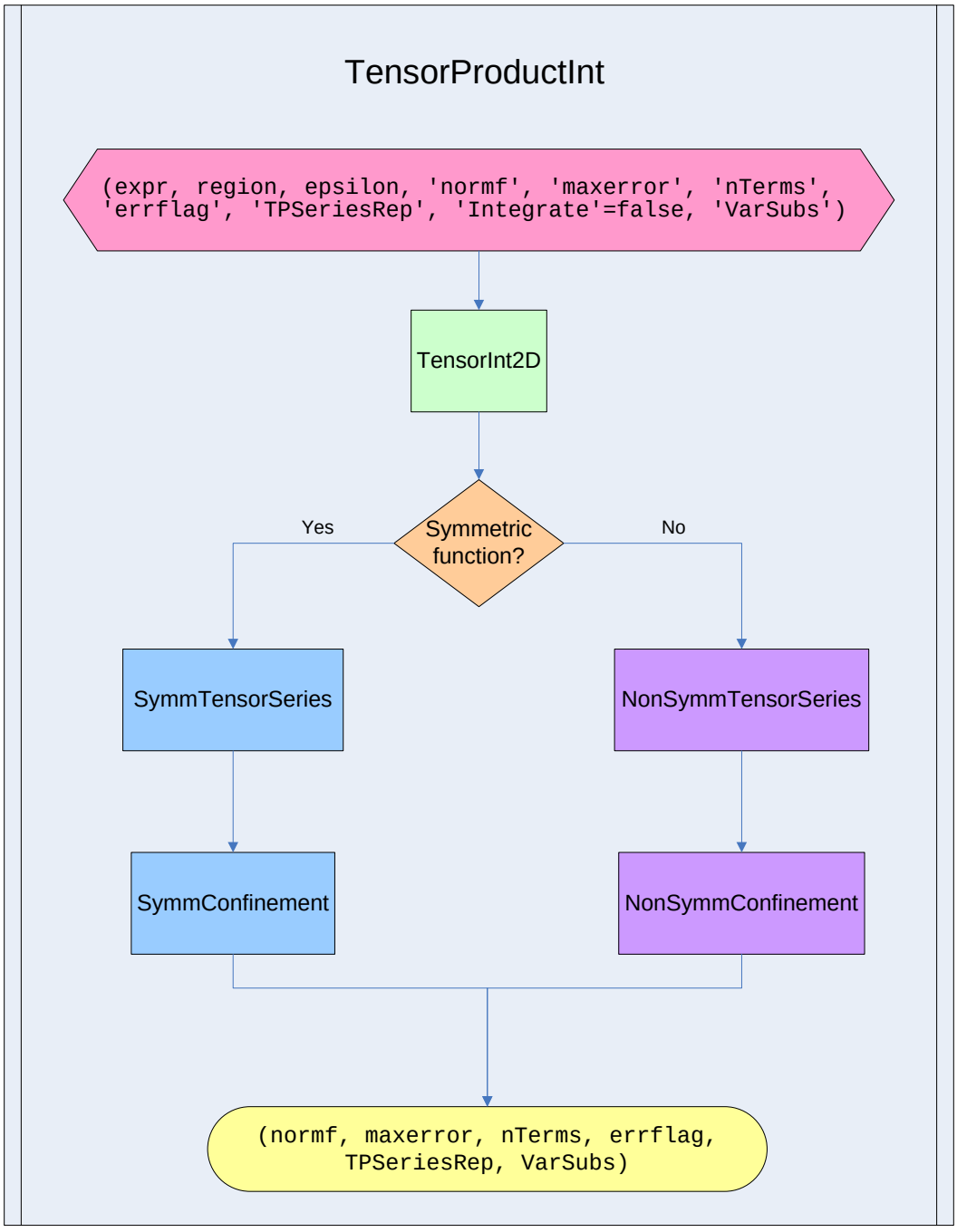

Figure 5.2: Structure of `TensorProductInt`

`nTerms` The number of terms, i.e., the rank, in the tensor product series approximation.

`maxerror` The estimated maximum error which is less than `epsilon`.

`normf` The maximum absolute value of `expr` in the given region.

`VarSubs` A list $[s, t]$ specifying the change of variables $x \to s$, $y \to t$ which transforms the original region into the unit square $[0, 1] \times [0, 1]$.

The value of `TPSeriesRep` here is a sequence `(TPSeries, Vx, U, D, L, Wy)` where `TPSeries` is an explicit expression of the desired approximation of `expr` in sum-of-products form, i.e., $s_n(x, y)$ in equation (5.9), and `(Vx, U, D, L, Wy)` are defined as in section 5.4.2 with n specified by `nTerms`.

In order to have this procedure work properly for our package, we also added an optional argument `errflag` whose value is assigned 0 for a normal return and assigned 1 if the requested accuracy failed to be achieved before the maximum number of iterations without improvement was reached. This allows our toolkit to treat the case `errflag=1` in the same way that we treat the case where `nTerms` is too large, namely, subdividing the region into smaller subregions.

Chapter 6

General Approximation Method for Bivariate Functions

6.1 Purpose of Our Maple Package

Some functions currently provided by Maple, such as `chebyshev` and `minimax`, are useful for generating approximations on a fixed small region. Unfortunately, when the region gets too large on the real line, the degree of the approximation may become large which is an impediment for efficient and accurate evaluations.

The solution that we use in our package, which is a straightforward way, is to divide the region on which we need to do the whole approximation of large degree into subregions on which we can use rational or polynomial approximation of smaller degree instead. In section 6.4, we describe the scheme used by our toolkit in Maple to divide the region into subregions.

There are some built-in functions in Maple, specifically bivariate functions of our interest such as the Bessel function of the first kind $J_v(x)$, which evaluate relatively slowly (e.g., for plotting or integrating over a two-dimensional region). This is because the numerical evaluation routine is a general-purpose routine operating in software floating point for any user-specified precision. We wish to find a better solution to improve the efficiency of the evaluations in hardware floating point precision without losing accuracy.

Our approach here is to generate efficient numerical evaluation routines for a given bivariate function by first converting the problem of approximating a bivariate function to the problem of approximation for a sequence of univariate functions, and then using polynomial or rational approximations for the univariate functions.

Generally there are four goals that we want to achieve when generating an evaluation routine for the original function $f(x, y)$:

- The maximum error of the numerical evaluation routine $fapprox$ on the region of interest $R = [a, b] \times [c, d]$ must be less or equal than the user-specified precision (or hardware floating point precision by default), i.e., $\frac{\|fapprox - f\|_\infty}{\max(\|f\|_\infty, 1)} \leq finaltol$ as discussed in section 2.4.

- The new evaluation routine should be efficient, i.e., significantly faster than Maple's current method. Note that the speed here refers to the execution time when using the generated approximation to evaluate at any point in the specified rectangular region of R, not the time required to generate the approximation.

- The process of generating approximations should be automated, such as by automatically subdividing the region R and then generating approximations on each subregion. The details of automated region selection will be explained in section 6.4.

- The generated routine must be numerical in the sense that it can be translated into the C language (or other languages) and compiled into a numerical library.

In the following sections, we describe how our toolkit meets these goals in practice and discuss the algorithms that are used in the package.

6.2 Use of the Toolkit

We describe the use of our toolkit in this section.

The following parameters are required to be specified by the user:

f — The bivariate function to be approximated, represented as a Maple expression (see section 2.1 for details).

`region` — A list defining the two-dimensional region of approximation with each dimension specified as an equation: `var=a..b`.

The following parameters are optional:

`rationalEqn` — An equation `'rational=true'` (the default case) or `'rational=false'` indicating whether rational approximations will be attempted. If the argument `'rational=false'` is specified then only polynomial approximations will be used.

`finaltol` The accuracy tolerance desired for the final approximation. If it is not specified, we use hardware floating point precision as the default value.

The value returned by the toolkit is a sequence `(hash, arrayRep)` which is specified as follows:

`hash` A Maple piecewise expression which, when evaluation is required at a specific point `(xval, yval)`, returns the integer `index` such that `index := hash(xval, yval)` defines the index in `arrayRep` corresponding to the approximation for the subregion containing the point `(xval, yval)`.

`arrayRep` A Maple table with two dimensions, the first being the integer `index` identifying a subregion. It stores the Matrix values `LDU`, `V` and `W` representing the tensor product approximation in that subregion (see section 6.10 for details).

Given a point (x, y) for evaluation, we can extract the corresponding information of the approximation from `(hash, arrayRep)` and compute the numerical result.

6.3 Algorithm

The following simplified algorithm explains how our main procedure for approximation generation works.

If the given bivariate function is denoted by f, the maximum error of the current approximation is denoted by δ, and the desired accuracy is denoted by ϵ, then our approach generally consists of three phases:

1. Invoke `` `evalf/int/TensorProductInt`:-TensorInt2D `` as introduced in section 5.6 to generate a tensor product series expansion for the given bivariate function, and then check the maximum error. If necessary, divide the current region of the x-y plane into subregions, call the main procedure with the new subregions, and exit the current procedure.
2. Set δ to `Float(infinity)`. As long as $\delta > \epsilon$ repeat the following steps:
   (a) Increase `Digits` for a higher working precision.
   (b) For each univariate function in the vector $V(x)$ do the following steps:
      i. Call `numapprox:-chebyshev` to get the truncated Chebyshev series expansion of the entry.

ii. If the user has specified `'rational=false'`, directly go to step 2c; otherwise, use `numapprox:-chebpade` to get the Chebyshev-Padé approximation of the entry. If rational approximation fails, we revert to using polynomial approximation, i.e., truncated Chebyshev series.

(c) Break the current x region into subregions if needed, invoke the main procedure again with new subregions, and exit the current procedure.

(d) For each univariate function in the vector $W(y)$ do the iterations as in step 2b.

(e) Break the current y region into subregions if necessary, invoke the main procedure once again with new subregions, and exit the current procedure.

(f) Randomly select $2\times$`nTerms` points on each x and y region, where `nTerms` is as defined in section 5.6. Check the errors of the approximation on these points, and then assign the maximum error to δ.

3. Since we have now achieved an approximation which satisfies $\delta \leq \epsilon$, we will store all the information of the approximation for the current region in `arrayRep`, an array with two dimensions.

By applying the above algorithm, we obtain the desired piecewise approximation in the form of a piecewise function `hash` for index association and an array `arrayRep`, as defined in section 6.2, which stores the approximation for each subregion.

The above algorithm description does not cover every part of the code and we will discuss each phase in more detail throughout this chapter.

6.4 Region Subdivision Process

For each region R_i there is a user-specified maximum degree of approximation, denoted by the global variable `MAXDEGREE`. There is also a user-specified maximum number of terms, denoted by `MAXTERMS`, for the tensor product series expansion. In experimenting with our package, we set `MAXDEGREE` and `MAXTERMS` to various values according to empirical results for different functions. Further experimentation will be needed to find the best values of `MAXDEGREE` and `MAXTERMS` automatically in the process of generating approximations.

In order to calculate the degree of the approximation so that we can know if it is necessary to subdivide the region, we use Maple's `numapprox:-chebyshev` command which returns the Chebyshev series truncated at a finite degree based on the requested accuracy for polynomial approximation, and we use the `numapprox:-che-`

`bpade` command which returns a Chebyshev-Padé approximation of specified degree (m, n) for rational approximation. The usage of these two commands are described in section 3.3.3 and 4.3.2.

Suppose $R = [0, 5] \times [0, 5]$ is the whole region on x and y where we wish to approximate the bivariate function f. The following algorithm describes when and how the toolkit breaks the region into subregions based on `MAXTERMS` and `MAXDEGREE`.

1. R_0 is passed into the main procedure, where $R_0 = R$ initially.

2. If the rank of the tensor product series is too large, i.e., `nTerms`>`MAXTERMS`, or `errflag`$\neq 0$, or the maximum error is too large, then we split the x region into two equal subregions, and split the y region into two equal subregions. Otherwise we skip to step 4. Thus we have split the rectangular region R_0 into four smaller regions $\{R_i\}_{i=1}^4$, where the area of each R_i is one-quarter that of R_0. Figure 6.1 illustrates the relationship of R_0 and $\{R_i\}_{i=1}^4$.

3. Call the main procedure again four times with a different $R_0 = R_i$ passed in each time. Exit from the current procedure since we do not need to compute approximations on the larger region R_0 any more.

4. If the degree of approximation for the functions of x is too large, i.e., `x_maxdeg` >`MAXDEGREE`, then split the x region into smaller subregions where each subregion is reduced to half size. Otherwise go to step 6 directly. Note that the y region is left unchanged. Thus we have R_0 split into two subregions $\{R_i\}_{i=1}^2$ this time, where the area of each R_i is half of that of R_0. The relationship of R_0 and $\{R_i\}_{i=1}^2$ is shown as in Figure 6.2.

5. For each R_i set $R_0 = R_i$ and return to step 1.

6. If `y_maxdeg`>`MAXDEGREE`, i.e., the degree of approximation for the functions of y is too large, we then split the y region into two smaller subregions. Otherwise there is no need to break the current region and we will just continue to compute the approximation for the current region. Similarly as in step 4, we only divide the y region but do not split the x region here. Again we will get two subregions R_1 and R_2 separated from R_0 by dividing R_0 by half on the y region, which is shown in Figure 6.3.

7. As in step 5, return to step 1 and invoke the main procedure twice with $R_0 = R_i$. Exit from the current procedure because it is not necessary to continue working on the larger region R_0 now.

Note that when we use rational approximation of degree (m_1, m_2) for the functions of x in step 4, the value of `x_maxdeg` is not $m_1 + m_2$ but $\max(m_1, m_2)$ instead. The same rule applies to `y_maxdeg` in step 6.

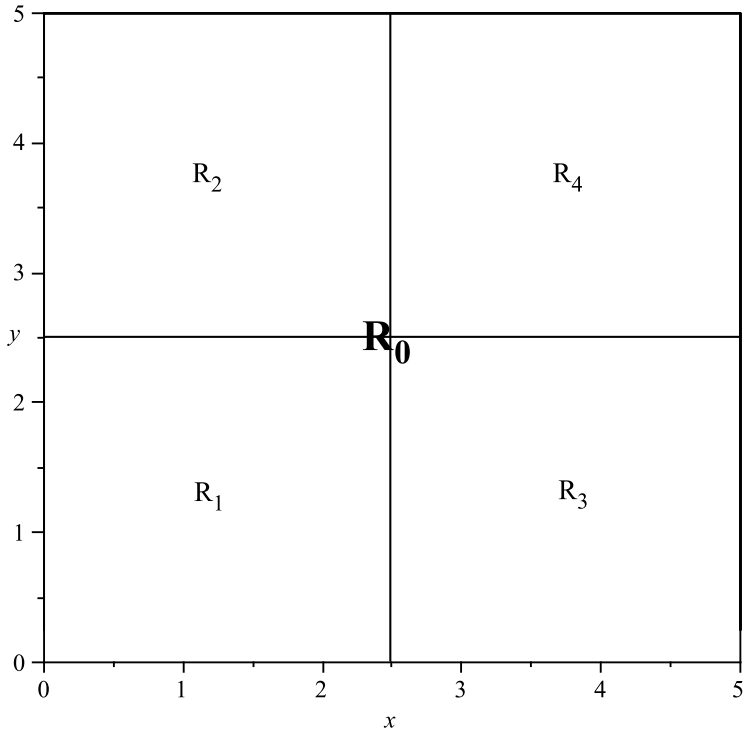

Figure 6.1: Relationship of R_0 and $\{R_i\}_{i=1}^{4}$

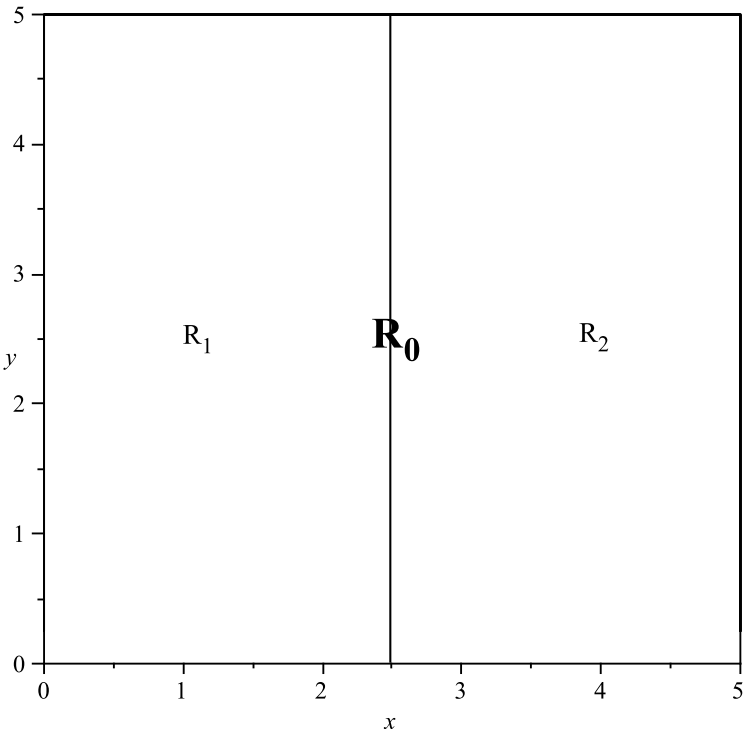

Figure 6.2: Relationship of R_0 and $\{R_i\}_{i=1}^{2}$ Splitting x Region

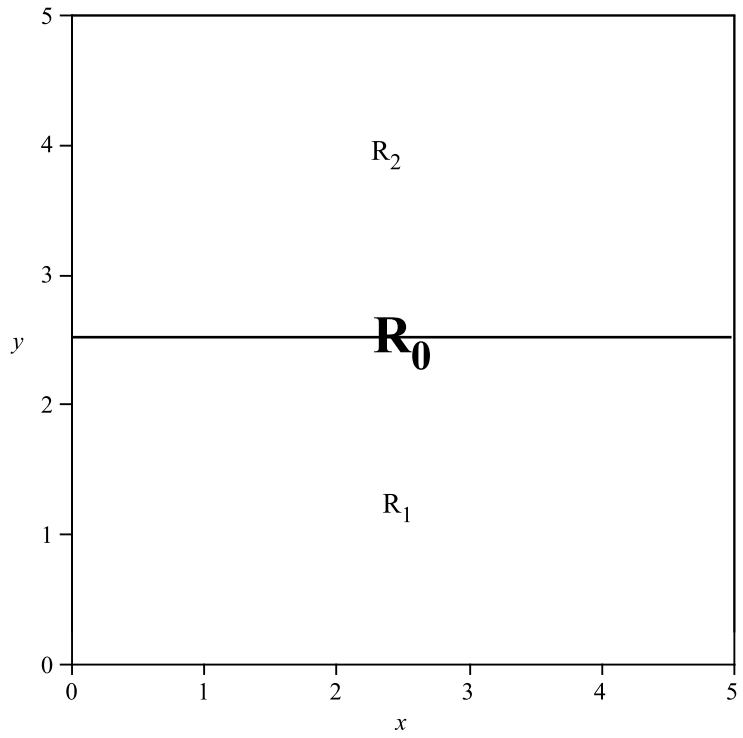

Figure 6.3: Relationship of R_0 and $\{R_i\}_{i=1}^{2}$ Splitting y Region

The above algorithm is actually a part of the algorithm in section 6.3. The steps 2 and 3 in this algorithm explain how the step 1 of the algorithm in section 6.3 works concretely. Analogously, the steps 4 and 5, and steps 6 and 7 here are more detailed versions of step 2c and step 2e in the algorithm of section 6.3 respectively.

Now we illustrate the idea of subdividing the region into subregions in our package by exhibiting an example of approximation in a large region.

Example 6.1

Suppose we wish to approximate the Bessel function $K_v(x)$ on the region $R = [0, 6] \times [1, 7]$, and we have `MAXTERMS` and `MAXDEGREE` set to 15 and 8 respectively. Then the piecewise expression `hash`, defined in section 6.2, returned by our toolkit is as follows:

$$
\begin{cases}
0 & v < 0.0 \ \ or \ \ x < 1.0 \\
1 & 0.0 \le v \ \ and \ \ v \le 1.5 \ \ and \ \ 1.0 \le x \ \ and \ \ x \le 1.75 \\
2 & 0.0 \le v \ \ and \ \ v \le 1.5 \ \ and \ \ 1.75 \le x \ \ and \ \ x \le 2.5 \\
3 & 0.0 \le v \ \ and \ \ v \le 1.5 \ \ and \ \ 2.5 \le x \ \ and \ \ x \le 4.0 \\
4 & 1.5 \le v \ \ and \ \ v \le 3.0 \ \ and \ \ 1.0 \le x \ \ and \ \ x \le 2.5 \\
5 & 1.5 \le v \ \ and \ \ v \le 3.0 \ \ and \ \ 2.5 \le x \ \ and \ \ x \le 4.0 \\
6 & 0.0 \le v \ \ and \ \ v \le 3.0 \ \ and \ \ 4.0 \le x \ \ and \ \ x \le 7.0 \\
7 & 3.0 \le v \ \ and \ \ v \le 4.5 \ \ and \ \ 1.0 \le x \ \ and \ \ x \le 2.5 \\
8 & 3.0 \le v \ \ and \ \ v \le 4.5 \ \ and \ \ 2.5 \le x \ \ and \ \ x \le 4.0 \\
9 & 4.5 \le v \ \ and \ \ v \le 6.0 \ \ and \ \ 1.0 \le x \ \ and \ \ x \le 2.5 \\
10 & 4.5 \le v \ \ and \ \ v \le 6.0 \ \ and \ \ 2.5 \le x \ \ and \ \ x \le 4.0 \\
11 & 3.0 \le v \ \ and \ \ v \le 6.0 \ \ and \ \ 4.0 \le x \ \ and \ \ x \le 7.0 \\
12 & otherwise
\end{cases}
$$

where the first column contains the indices and the second column are the subregions associated with each index.

The resulting subdivisions for the region R are illustrated in Figure 6.4.

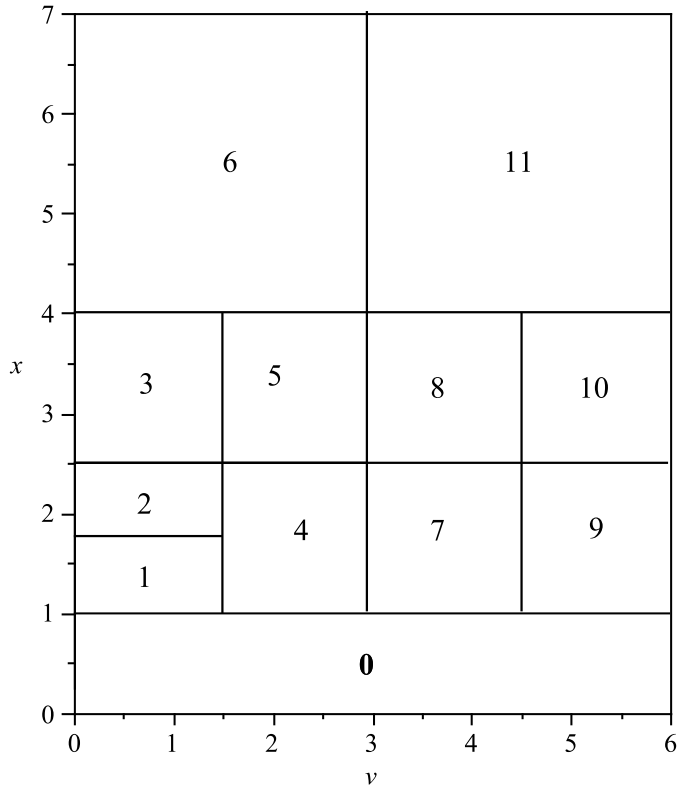

Figure 6.4: Subregions for Example 6.1

Hence given a point (v_i, x_j) at which to evaluate, we can pass it into `hash` which returns the associated index k that we can use to find the corresponding approximation on the k-th subregion stored in `arrayRep`. For example, for the point $(4.6, 2.9)$ invoking `hash(4.6,2.9)` will return 10, meaning that we should evaluate the approximation of the subregion indexed by 10.

6.5 Structure of the Toolkit

Figure 6.5 illustrates the working structure of the three-phase algorithm of section 6.3 as well as the region subdivision algorithm of section 6.4.

In this figure, `generateApprox` is the procedure that the user invokes in order to use the toolkit. After reading in, analyzing and processing the parameters `(f, region, rationalEqn, finaltol)`, `generateApprox` passes them into the procedure `generatePwlist`, which invokes all the procedures inside the big green box in the figure, for the main work of creating approximations for the bivariate function f over the (possibly large) specified `region` of the x-y plane. Eventually `generateApprox` returns `(hash, arrayRep)` as defined in section 6.2.

All the blue arrows show how the generation of the approximation goes through successfully to the third phase which ultimately sends the approximation results back to `generateApprox` implicitly.

Each red arrow shows when the region will be possibly subdivided after implementing the procedure which it points from. The region is subdivided as necessary to achieve approximations on each subregion satisfying the constraints specified by the global variables `MAXTERMS` and `MAXDEGREE`. We can see that all the red arrows point to `generatePwlist`; hence this procedure works in a recursive way and we apply `generatePwlist` to each subregion in order to get different approximations because of region subdivision.

The pink arrows are relevant only when the user has specified `'rational=false'`, in which case we skip the two steps of invoking `chebpade on x` and `chebpade on y` as shown in the figure.

The black arrow represents that when the error is too large we will increase the value of `Digits` and repeat the current phase. Note that increasing the working precision specified by `Digits` implies that `chebyshev` will return a higher-degree approximation.

Note that the number of subregions corresponds to the number of indices in `hash`. Generally the more indices we have, the more distinct approximations are generated and the longer execution time for generation is needed.

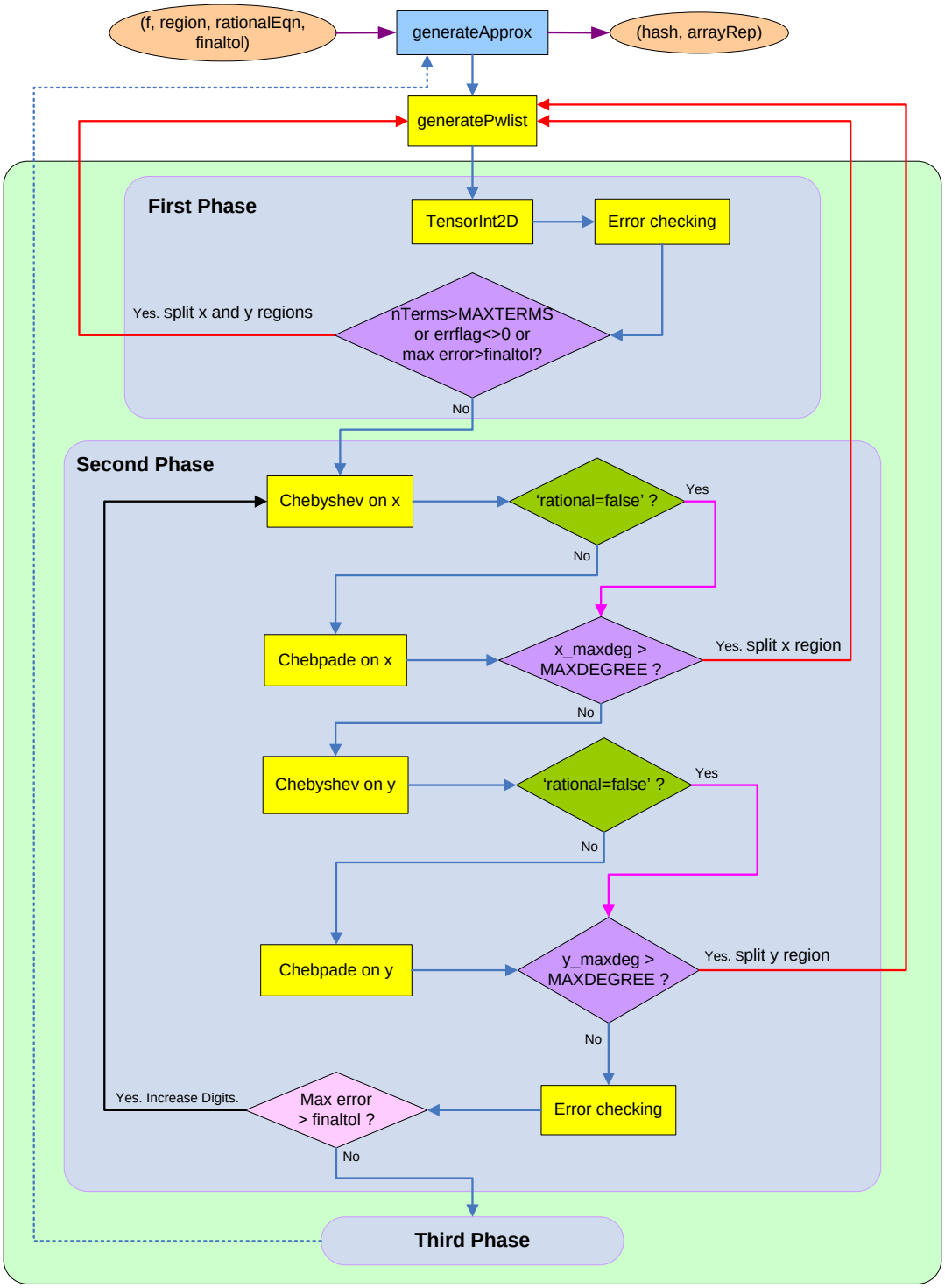

Figure 6.5: Structure of the Toolkit

6.6 Singularities

A singularity is in general a point at which the given mathematical function is not defined or is unbounded. For example, the function

$$f(x) = \frac{1}{x} \tag{6.1}$$

on the real line has a singularity at $x = 0$ since it tends to $\pm\infty$ and is not defined at this point.

More generally, we consider a point x_0 to be a singularity if the function does not have a Taylor series expansion about $x = x_0$. For example, $f(x) = x^{\frac{1}{2}}$ has a "square root singularity" at $x = 0$.

It is possible that there exists such a singularity x_0 for a function $f(x)$ that we wish to approximate, which will cause troubles due to two reasons:

- It becomes impossible to get a polynomial approximation that is correct when arbitrarily close to x_0. Depending on the type of the singularity, using rational approximation might get a better result than polynomial approximation in such case, in the sense of being able to approximate closer to the singularity.
- It tends to require a very high degree when we approximate $f(x)$ near the singularity even on a quite small interval, both for polynomial and rational approximation cases. This may result in highly inefficient evaluation or an infinite loop during the approximation generation.

In Thomas A. Robinson's Masters thesis [9], he attempted to automatically remove the singularities before approximating the function and then replace them once the approximation has been completed for the modified function. He tried two methods, multiplying and subtracting the singularity, with some success. For our current project on bivariate approximation, we have not attempted automatic removal of singularities except to the extent discussed below.

In our generation of the bivariate approximation, we artificially keep the region away from the singularity to avoid such problems. For example, if the singularity is at $x_0 = 0$ we will restrict $x \geq x_{min}$ for the specified interval used in our toolkit, where x_{min} may be chosen as small as $\frac{1}{1000}$.

Additionally, in order to minimize the degree of the approximation when near singularities, we will factor out a power of y for some functions.

Let us first see an example of the Bessel function of the second kind, $Y_x(y)$, so that we can illustrate this idea more intuitively.

Example 6.2

Suppose we are approximating $Y_x(y)$ on the region $R = [0,2] \times [\frac{1}{10}, 1]$. In this example, we concentrate on approximating in the variable y, with x fixed.

Since $\lim_{y \to 0^+} Y_x(y) = -\infty$ for any x, $y = 0$ is a singularity for $Y_x(y)$. If x is a non-integer such as $\frac{1}{4}$, the series expansion for $Y_x(y)$ at $y = 0$ is

$$Y_{\frac{1}{4}}(y) = -\frac{2^{\frac{3}{4}}}{\Gamma\left(\frac{3}{4}\right) y^{\frac{1}{4}}} + \frac{2^{\frac{5}{4}} \Gamma\left(\frac{3}{4}\right) y^{\frac{1}{4}}}{\pi} + \frac{1}{3} \cdot \frac{2^{\frac{3}{4}} y^{\frac{7}{4}}}{\Gamma\left(\frac{3}{4}\right)} - \frac{2}{5} \cdot \frac{2^{\frac{1}{4}} \Gamma\left(\frac{3}{4}\right) y^{\frac{9}{4}}}{\pi} - \frac{1}{42} \cdot \frac{2^{\frac{3}{4}} y^{\frac{15}{4}}}{\Gamma\left(\frac{3}{4}\right)} + O\left(y^{\frac{17}{4}}\right). \tag{6.2}$$

Actually for any value of x, no matter integer or non-integer, the expansion of $Y_x(y)$ about $y = 0$ starts with a term in y^{-x}. It is obvious that evaluation at points (x, y) with y close to 0 will pose difficulties.

Let us try to compute the truncated Chebyshev series for $Y_{\frac{1}{4}}(y)$ to the hardware floating point precision `hfeps`:

```
> with(numapprox):
> chebseries := chebyshev(BesselY(1/4, y), y = 1/10 ..  1, hfeps):
> deg := chebdeg(chebseries); .
```

The `deg` obtained here is 44 which is quite large.

However, if we first multiply $Y_{\frac{1}{4}}(y)$ by $y^{\frac{1}{4}}$ we get a function that is better-behaved at $y = 0$. The series expansion for $y^x Y_x(y)$ with $x = \frac{1}{4}$ at $y = 0$ is

$$-\frac{2^{\frac{3}{4}}}{\Gamma\left(\frac{3}{4}\right)} + \frac{2^{\frac{5}{4}} \Gamma\left(\frac{3}{4}\right) y^{\frac{1}{2}}}{\pi} + \frac{1}{3} \cdot \frac{2^{\frac{3}{4}} y^2}{\Gamma\left(\frac{3}{4}\right)} - \frac{2}{5} \cdot \frac{2^{\frac{1}{4}} \Gamma\left(\frac{3}{4}\right) y^{\frac{5}{2}}}{\pi} - \frac{1}{42} \cdot \frac{2^{\frac{3}{4}} y^4}{\Gamma\left(\frac{3}{4}\right)} + O\left(y^{\frac{9}{2}}\right). \tag{6.3}$$

When we compute the Chebyshev series for this expansion we get `deg`=38 which is smaller than the previous result.

We can notice that this series expansion has a square root singularity at $y = 0$. For the case of integer x, it reveals a logarithmic singularity. Therefore rational approximation will be advantageous over polynomial approximation in both of the two cases.

After trying different degrees we find that `hfeps` can be sufficiently achieved by rational approximation q of degree (15,15), a great reduction from degree 44, in the following form:

$$q = \frac{\sum_{k=0}^{15} c_k T_k\left(\frac{20}{9} y - \frac{11}{9}\right)}{\sum_{k=0}^{15} d_k T_k\left(\frac{10}{9} y - \frac{11}{9}\right)} \tag{6.4}$$

where c_k and d_k are the numerator coefficients and denominator coefficients respectively.

In the end we can get an approximation for the original function $Y_{\frac{1}{4}}(y)$ by forming $y^{-\frac{1}{4}}\,q$.

In order to have more efficient approximations, as shown in the above example for $Y_x(y)$, we will always factor out y^{const} when $f(x,y)$ is Bessel function $J_x(y)$ or $I_x(y)$ for which the series expansion about $y=0$ has leading term $f(const,y) = coef \times y^{const} + ...$ Similarly for $f(x,y) = Y_x(y)$ or $K_x(y)$, the series expansion about $y=0$ has leading term $f(const,y) = coef \times y^{-const} + ...$ To avoid the singularity at $y=0$ we will factor out y^{-const} .

6.7 Preparation Work

Starting from this section we are going to see how our package works in more detail.

We will begin with the procedure `generateApprox` and the arguments processing stage first.

6.7.1 `generateApprox`

After reading in the arguments `(f, region, rationalEqn, finaltol)`, the procedure `generateApprox` first checks if the quantities of the arguments and their format are correct.

If `rationalEqn` is specified, it determines the value of `doRational`:

```
> doRational := rhs(rationalEqn); .
```

Otherwise we always try rational approximations by default:

```
> doRational := true; .
```

If the optional argument `finaltol` isn't specified, we use hardware floating point precision as the accuracy tolerance in the package.

```
> hfDigits := trunc(evalhf(Digits));
> tol := Float(5,-hfDigits); .
```

We then analyze the `region` of x-y and set the corresponding values to `(xrange, yrange)`. Note that we must avoid approximating on the singularity area; however,

for some functions such as $J_x(y)$, specifying `xmin` $= 0$ is allowed but we must specify `ymin` > 0 due to the singularity at (0,0).

```
> (x_orig, y_orig) := (lhs(region[1]), lhs(region[2]));
> (x, y) := (x_orig, y_orig);
> (xmin, xmax) := op(rhs(region[1]));
> (ymin, ymax) := op(rhs(region[2]));
> (xrange, yrange) := (xmin..xmax, ymin..ymax); .
```

We also need to initialize `pwlist` which is a global variable to procedures `generateApprox` and `generatePwlist`.

```
> pwlist := []; .
```

After processing all the arguments, `generateApprox` passes the arguments `(f, x, y, xrange, yrange, tol, doRational)` into `generatePwlist`, of which there is no returned value but instead it updates the global variables `pwlist` and `arrayRep`.

Note that in our package we have some code to print related information for the user; however, for simplification we will not display this code or any insignificant code here. This rule applies to all the Maple code we will show in this thesis.

6.7.2 `pwlist`

Before getting down to `generatePwlist`, we will first introduce the definition of the construct named `pwlist` in our package.

As described in the Maple help pages, the command `convert(piecewise, pwlist, x)` in Maple will convert a piecewise expression, such as `piecewise(cond1, f1, cond2, f2,...)`, into an ordered list $[f_1, t_1, f_2, t_2, ..., f_n, t_n, f_{n+1}]$ which represents the piecewise-defined function:

$$f_i \text{ when } t_{i-1} < x < t_i, \quad i = 1, 2, ..., n+1$$

where $t_0 = -\infty$ and $t_{n+1} = \infty$ implicitly.

In our package we use an extension of this idea for univariate functions to our case of bivariate functions.

For example, if we have the piecewise function `piecewise`($x < 0.0$ or $y < 0.0$, 0, $0.0 \le x \le 1.7$ and $0.0 \le y \le 2.3$, 1, $0.0 \le x \le 1.7$ and $2.3 \le y \le 3.5$, 2, 3), then the corresponding representation of `pwlist` is [0, [0.0, 0.0], 1, [0.0, 1.7, 0.0, 2.3], 2, [0.0, 1.7, 2.3, 3.5], 3] which associates index 0 with $x < 0.0$ or $y < 0.0$ (outside the region of interest), associates index 1 with x in [0.0, 1.7] and y in [0.0, 2.3], associates index 2 with x in [0.0, 1.7] and y in [2.3, 3.5], and associates index 3 otherwise.

Since we specify closed intervals, there will be some redundancy for the index association. For example, the point (1.7, 2.3) could be associated with index 1 or 2. In such a case, the piecewise construct will choose the first condition which is satisfied.

As we process each subregion, we build up the information for the particular piecewise function in `pwlist`.

6.8 Tensor Product Series Approximation

Now let us look at the first phase in the procedure `generatePwlist` which is for generating a tensor product series .

After reading in the arguments `(f, x, y, curXrange, curYrange, finaltol, rational)`, `generatePwlist` first defines the function f in procedure form (see section 2.1 for details) which is useful for error checking after we have the approximation.

```
> fproc := unapply(f, (x,y)); .
```

Then we enter the first phase and we increase `Digits` by the global variable `GUARD_TensorInt2D` which specifies the guard digits for the `TensorInt2D` computation. We will set the value of `GUARD_TensorInt2D` in the testing setup of section 7.1.

Note that the accuracy `sftol` used for `TensorInt2D` is separate from `finaltol`. `sftol` changes as `Digits` increases while `finaltol` is fixed as specified by the calling routine.

As introduced in section 5.6, by invoking `TensorInt2D` we are able to compute a natural tensor product series expansion `TPSeriesRep` of rank `nTerms` with computed value `maxerr` less than the specified `sftol`. We then extract components `(TPSeries, Vx, U, D, L, Wy)` from the variable `TPSeriesRep`.

The corresponding Maple code is as follows:

```
> orig_Digits := Digits;
> maxrelerr := Float(infinity);
> Digits := Digits + GUARD_TensorInt2D;
> sftol := Float(5,-Digits);
> `evalf/int/TensorProductInt`:-TensorInt2D(f, [x=curXrange,
      y=curYrange], sftol, 'normf', 'maxerr', 'nTerms', 'errflag',
      'TPSeriesRep', 'Integrate'=false, 'VarSubs');
> (TPSeries, Vx, U, D, L, Wy) := TPSeriesRep; .
```

As discussed in section 5.4.3, if we compute with the diagonal entries of matrix `D` corresponding to $r_n <$ `sftol`, we would likely have inaccurate numerical results because of roundoff error contamination. Hence we will trim `nTerms` to `nTerms1` based on roundoff error level corresponding to `finaltol` (if `normf`≤ 1) or `normf`$\times$`finaltol` (if `normf`>1). Recall from equation (5.6), the remainder $|r_n(x,y)|$ is the absolute error for the approximation. Hence we can estimate the error of the approximation by using `D[nTerms1+1,nTerms1+1]` since it is the reciprocal of the remainder.

Here is the Maple code:

```
> if nTerms <= MAXTERMS and errflag = 0 then
>   cutoff := `if`(normf > 1.0, 1/(normf*finaltol), 1/finaltol);
>   for i from nTerms by -1 to 1 while abs(D[i,i]) > cutoff do
>   end do;
>   nTerms1 := i;
>   if nTerms1 < nTerms then
>      maxabserr := 1/abs(D[nTerms1+1,nTerms1+1]);
>      maxerr := maxabserr/max(1.0,normf);
>   else
>      errflag := 1;
>   end if;
> end if; .
```

In `generatePwlist`, the current subregion is specified by `x=curXrange` and `y=curYrange`. If `nTerms` is too large, or `errflag` is not 0 indicating that `TensorInt-2D` failed to achieve the requested accuracy `sftol`, or the maximum error `maxerr` in the first phase is not smaller than `finaltol`, we split both the current x and y regions into subregions, call `generatePwlist` with each subregion and then return, i.e., exit the current procedure `generatePwlist`. Otherwise we continue to the second phase.

```
> if nTerms > MAXTERMS or errflag <> 0 or maxerr > finaltol then
>   Digits := orig_Digits;
>   aa[0] := evalf(op(1, curXrange));
>   aa[2] := evalf(op(2, curXrange));
>   aa[1] := (aa[0] + aa[2])/2.0;
>   cc[0] := evalf(op(1, curYrange));
>   cc[2] := evalf(op(2, curYrange));
>   cc[1] := (cc[0] + cc[2])/2.0;
>   for i from 1 to 2 do
>      for j from 1 to 2 do
>         generatePwlist(f, x, y, aa[i-1]..aa[i], cc[j-1]..cc[j],
            finaltol, rational);
>      end do;
>   end do;
```

```
>   return;
> end if; .
```

6.9 Univariate Function Approximation

In the second phase of `generatePwlist`, we compute the approximations for the univariate functions in the vectors `Vx` and `Wy`.

6.9.1 Initialization

At the beginning we initialize the values for some variables:

```
> Digits := orig_Digits;
> (maxerr, prevmaxerr) := (Float(infinity), Float(infinity));
> count := 0; .
```

Then we repeat the main loop in this phase until the desired accuracy `finaltol` is reached.

The first step in the main loop is to set accuracy tolerance `sftolcheb` which determines the degree of the Chebyshev series approximation for f (represented by the vector `chebcoef`). We also use a higher working precision `sftol` similarly as in the first phase; however, the difference is that `sftol` will be changed here each time we repeat since `Digits` is increased each time.

```
> count := count + 1;
> sftolcheb := Float(5, -(Digits + count*GUARD_chebyshev));
> Digits := Digits + (2*count+1)*GUARD_chebyshev;
> sftol := Float(5,-Digits); .
```

The approximation degree determined by `sftolcheb` is required for the given function f and we do not want the degree to be higher than necessary. Hence, `GUARD_chebyshev` should be relatively smaller than `GUARD_TensorInt2D`, both of which are global variables. As for `GUARD_TensorInt2D`, we will also set the value of `GUARD_chebyshev` in the testing setup of section 7.1.

We have already obtained `nTerms1` from `nTerms` based on roundoff error level in the first phase. We now assign the value of `nTerms1` to `nTerms2`, which will be used as the number of entries in `Vx` and `Wy` later:

```
> nTerms2 := nTerms1; .
```

6.9.2 Approximation for `Vx`

Now we compute Chebyshev approximations for the univariate functions in vector `Vx`. Again there are some initializations first:

```
> fromChebseries_x := NULL;
> x_degreelist := NULL;
> x_maxdeg := 0; .
```

We should keep in mind that the vector `Vx` contains entries of the form $f(x, b_i)$ for various values of b_i and similarly `Wy` contains entries of the form $f(a_i, y)$, where a_i and b_i are as defined in equation (5.7). So when we use polynomial or rational approximation for the univariate functions, we need to do the univariate approximation for each entry separately in `Vx` and `Wy`. Using polynomial approximation for one entry in `Vx` does not necessarily mean that we need to use the same type of approximation for another entry in `Vx` (unless it is required by the user); and vice versa. Therefore we are theoretically using a mixture of polynomial and rational approximations for `Vx` (and similarly for `Wy`) in our method, though it is often seen in practice that we are using the same kind of univariate approximation for all the entries in `Vx` and `Wy` .

For each entry i in `Vx`, the total number of which is `nTerms2`, we need to implement the following steps:

We first compute the Chebyshev series to a higher degree based on `sftol`. See the details for usage of command `numapprox:-chebyshev` in section 3.3.3. The higher-degree Chebyshev series (represented by `cheb_accuratecoef`) will be used for computing an estimate of the error in `chebcoef`. In case the function is only defined for limited accuracy, we use a try-catch clause. If the `numapprox:-chebyshev` command fails, try again with `cheb_accuratetol` increased to be between `sftol` and `sftolcheb`, which decreases the Chebyshev degree.

```
> h := Vx[i];
> cheb_accuratetol := sftol;
> cheb_success := false;
> for trycount from 1 to 2 while not cheb_success do
>    try
>       (cheb_accuratecoef, var) := numapprox:-chebyshev(h,
            x=curXrange, cheb_accuratetol, 'output=Vector', 487,
            true);
>       cheb_success := true;
>    catch "singularity in or near interval":
>       cheb_success := false;
>       if trycount = 1 then
>          cheb_accuratetol := 10^((log[10](sftol)+log[10](
             sftolcheb))/2.0);
```

```
>       end if;
>    end try;
> end do; .
```

If the above steps work successfully, we then truncate the Chebyshev series according to the accuracy `sftolcheb`. Otherwise, we set the values of `degree_x` and `maxerr` to infinity so that we know that the current region needs subdividing.

```
> if cheb_success then
>    dim := LinearAlgebra:-Dimension(cheb_accuratecoef);
>    varproc := unapply(var, x);
>    maxcoef := LinearAlgebra:-VectorNorm(cheb_accuratecoef);
>    chebtol := 1/5*maxcoef*sftolcheb;
>    for K from dim by -1 to 1 while abs(cheb_accuratecoef[K])<
       chebtol do end do;
>    chebcoef := cheb_accuratecoef[1..K];
>    degree_x := K-1;
>    if K < dim then
>       maxerr := LinearAlgebra:-VectorNorm(cheb_accuratecoef[
         K+1..dim], 1) /maxcoef;
>    else
>       maxerr :=`if`(K>1, chebcoef[K]/chebcoef[K-1]*chebcoef[K],
         sftolcheb);
>    end if;
> else
>    chebcoef := Vector([0.0]);
>    degree_x := infinity;
>    maxerr := Float(infinity);
> end if;
> fromChebseries_x := fromChebseries_x, degree_x; .
```

If `cheb_success` and the argument `rational` are both true, we will attempt to use rational approximation. Note that for some functions, such as `LegendreP(x,y)`, particular cross-sections $f(x, const)$ or $f(const, y)$ may degenerate to polynomials of low degree. Therefore, we require `degree_x` to be larger than 5 before using rational approximation, otherwise we will use polynomial approximation.

The maximum error of the approximation p here is estimated by

$$\frac{\|f-p\|_\infty}{\|f\|_\infty} = \frac{\|\text{cheb\_accuratecoef} - \text{approxcoef}\|_\infty}{\text{maxcoef}} \tag{6.5}$$

where the norm of `cheb_accuratecoef-approxcoef` is bounded by the absolute sum of the coefficients.

Note that for many functions, one finds that if polynomial approximation requires degree n then to achieve an approximation of the same accuracy by rational

approximation, typically the degree will be (m_1, m_2) with $m_1 + m_2 < n$. We choose to try rational approximations by starting with degree $(\frac{n}{2}+1, \frac{n}{2}+1)$ and decreasing both components by 1 until discovering the minimum possible degree.

If we are unable to compute the Chebyshev-Padé approximation, we then try with denominator degree smaller than numerator degree since it may work for some functions in this way. If we still fail in doing so, we catch the error and revert to use polynomial approximation.

Here is the corresponding Maple code:

```
> halfdeg := iquo(degree_x, 2);
> (m, n) := (halfdeg+2, halfdeg+2);
> while m+n-2 >= dim do (m, n) := (m-1, n-1) end do;
> (numcoef, dencoef) := ('numcoef', 'dencoef');
> maxerr := 0.0;
> for trycount from 1 to halfdeg while maxerr < sftolcheb do
>   (m, n) := (m-1, n-1);
>   try
>      prev := (numcoef, dencoef, var);
>      (numcoef, dencoef, var) := numapprox:-chebpade(
         cheb_accuratecoef, var, [m,n], 'output=Vector');
>      chebratproc := t -> numapprox:-chebeval(numcoef,
         varproc(t))/numapprox:-chebeval(dencoef, varproc(t));
>      (approxcoef, dummy) := numapprox:-chebyshev(chebratproc,
         curXrange, sftol, 'output=Vector');
>      dimapprox := LinearAlgebra:-Dimension(approxcoef);
>      if dimapprox > dim then
>         approxcoef := approxcoef[1..dim];
>      elif dimapprox < dim then
>         approxcoef := Vector(dim, [seq(approxcoef[k], k = 1
           ..dimapprox), seq(0.0, k = dimapprox+1 ..  dim)]);
>      end if;
>      maxerr := LinearAlgebra:-VectorNorm(cheb_accuratecoef-
         approxcoef,1) /maxcoef;
>   catch "Chebyshev-Pade approximation of given degree is
           not well-defined",
         "singularity in or near interval":
>      (m, n) := (m+2, n+1);
>      while m+n-2 >= dim do (m, n) := (m-1, n-1) end do;
>   catch:
>      maxerr := sftolcheb;
>   end try;
>   if trycount = 1 and maxerr >= sftolcheb then
>      (m, n) := (m+2, n+2);
>      if m+n-2 < dim then
```

```
>         (numcoef, dencoef) := ('numcoef', 'dencoef');
>         maxerr := 0.0;
>      end if;
>   end if;
> end do;
> (numcoef, dencoef, var) := prev;
> if type(numcoef,'Vector') and type(dencoef,'Vector') then
>   (numdeg, dendeg) := ( LinearAlgebra:-Dimension(numcoef)-1,
     LinearAlgebra:-Dimension(dencoef)-1 );
> else
>   (numdeg, dendeg) := (degree_x, 0);
>   (numcoef, dencoef) := (chebcoef, Vector([0.0]));
> end if;
> x_degreelist := x_degreelist, [numdeg, dendeg];
> x_maxdeg := max(x_maxdeg, numdeg, dendeg); .
```

If `rational` is false, we simply use polynomial approximation:

```
> (numcoef, dencoef) := (chebcoef, Vector([0.0]));
> x_degreelist := x_degreelist, [degree_x, 0];
> x_maxdeg := max(x_maxdeg, degree_x); .
```

Since we have achieved the approximation for the entry now, we will store the corresponding information into array `xUnivarApprox`. If `x_maxdeg` of the current entry is already too large, we simply break out of the loop since we are definitely going to subdivide the current x region and there is no need to continue the loop:

```
> xUnivarApprox[i] := (1, numcoef, dencoef, var);
> if x_maxdeg > MAXDEGREE then break; end if; .
```

After computing approximations for all the entries in `Vx`, we compare the degree for approximation of x and `MAXDEGREE`. If the former is too large then split the x region into smaller subregions and recall `generatePwlist`:

```
> if x_maxdeg > MAXDEGREE then
>    Digits := orig_Digits;
>    aa[0] := evalf(op(1, curXrange));
>    aa[2] := evalf(op(2, curXrange));
>    aa[1] := (aa[0] + aa[2])/2;
>    for i from 1 to 2 do
>       generatePwlist(f, x, y, aa[i-1]..aa[i], curYrange,
            finaltol, rational);
>    end do;
>    return;
> end if; .
```

6.9.3 Approximation for `Wy`

We next compute Chebyshev approximations for the functions in vector `Wy`, which is very similar as we did for `Vx`.

Note that we factor out a power of y for some functions as discussed in section 6.6. We also need to apply `fnormal` to `op(2,h)` to the level of `orig_Digits` since `op(2,h)` may be in a form such as $1.0\,y + 2.3 \times 10^{-17}$, though we expect `op(2,h)`=$y$. By using `fnormal` we are able to remove the phantom zero of `op(2,h)` and strip the constant 1.0 from y.

Therefore, we need to add the following at the beginning of approximation for the univariate functions in each entry of `Wy`:

```
> fact := 1;
> if member(op(0,h), {BesselJ,BesselI,BesselY,BesselK}) then
>   arg2 := fnormal(op(2,h), orig_Digits);
>   if type(arg2,`*`) and abs(op(1,arg2)-1.0) <
        Float(5,-orig_Digits) then
>     arg2 := mul(op(i,arg2), i=2..nops(arg2));
>   end if;
>   if op(0,h) = BesselJ or op(0,h) = BesselI then
>     fact := arg2^op(1,h);
>   elif op(0,h) = BesselY or op(0,h) = BesselK then
>     fact := arg2^(-op(1,h));
>   end if;
> end if; .
```

Remember that we must use `h/fact` instead of `h` in `numapprox:-chebyshev` for computing approximations for the functions in `Wy`:

```
> (cheb_accuratecoef, var) := numapprox:-chebyshev(h/fact,
    y=curYrange, cheb_accuratetol, 'output=Vector', 487, true); .
```

Just as for the approximation of x, if the degree for approximation of y is too large we then divide the y region and exit `generatePwlist`:

```
> if y_maxdeg > MAXDEGREE then
>   Digits := orig_Digits;
>   cc[0] := evalf(op(1,curYrange));
>   cc[2] := evalf(op(2,curYrange));
>   cc[1] := (cc[0]+cc[2])/2.0;
>   for j from 1 to 2 do
>     generatePwlist(f, x, y, curXrange, cc[j-1]..cc[j],
          finaltol, rational);
>   end do;
>   return;
> end if; .
```

6.9.4 Error Checking

In order to check whether the approximation computed in this phase satisfies the required accuracy, we first need to get new `L` and `U` matrices, both of which are global variables for error checking of the second phase:

```
> L_new := Matrix(nTerms2,nTerms2,datatype=sfloat);
> U_new := Matrix(nTerms2,nTerms2,datatype=sfloat);
> for i from 1 to nTerms2 do
>    for j from i to nTerms2 do
>       U_new[i,j] := U[i,j];
>    end do;
>    for j from 1 to i do
>       L_new[i,j] := L[i,j];
>    end do;
> end do; .
```

Afterwards we invoke the procedure `absErrorCheck2` to check the error of the approximation that we get in this phase. The input of `absErrorCheck2` is as follows:

`fproc` The bivariate function $f(x, y)$ in procedure form.

`x=a..b` The variable name x and the range of values for x.

`y=c..d` The variable name y and the range of values for y.

`npts` The number of points to be randomly selected in each region of x and y for testing errors.

`absErrorCheck2` uses the global variables `nTerms2`, `L_new` and `U_new`, computes the absolute errors at the `npts`×`npts` points and outputs `(maxabserr, avgabserr, normf)` where `maxabserr` is the maximum absolute error, `avgabserr` is the average absolute error, and `normf` is the maximum absolute value of $f(x, y)$.

```
> (maxabserr, avgabserr, normf) := absErrorCheck2(fproc,
      x=curXrange, y=curYrange, 2*nTerms);
> maxerr := maxabserr/max(1.0,normf); .
```

If `maxerr` is larger than `finaltol` we will repeat the main loop in the second phase again until we achieve the desired accuracy. Otherwise we will continue to the third phase.

6.10 Store of the Approximation

At the completion of the procedure `generatePwlist`, we have obtained the desired approximation for the current subregion. The final phase is to store the approximation in `arrayRep` so that we can compile the information into a numerical library and retrieve it for applications such as evaluation or plotting.

We first introduce the procedure `pwlistUpdate` that we developed to be used in this phase.

6.10.1 `pwlistUpdate`

There are three parameters for the procedure `pwlistUpdate`:

`pwlist` A list (initially empty) representing a piecewise function, as described in section 6.7.2.

`xRange` A new range such as `xleft..xright`.

`yRange` A new range such as `yleft..yright`.

Then `pwlistUpdate` outputs the following:

`Count` The index associated with the new ranges of x and y.

`pwlist` The updated list with the new ranges added.

As discussed in section 6.2, we define the function `hash` as a piecewise function. After we process each particular subregion, we build up the information for the `hash` function in `pwlist` by adding the current region to `pwlist`.

```
> (index, pwlist) := pwlistUpdate(pwlist, curXrange, curYrange); .
```

6.10.2 Approximation Store in `arrayRep`

Recall from equation (5.9) that the bivariate approximation $p(x, y)$ for the current subregion satisfies

$$p(x, y) = V^T(x) \cdot U \cdot D \cdot L \cdot W(y). \tag{6.6}$$

Note that the univariate approximation for each entry in `Vx` (and similarly in `Wy`) is of the following form

$$f(x, const) = x^{\tau} \frac{\sum_{k=0}^{MAXDEGREE} c_k T_k(\alpha\, x + \beta)}{\sum_{k=0}^{MAXDEGREE} d_k T_k(\alpha\, x + \beta)} \tag{6.7}$$

where α, β and τ are constants.

If we are using polynomial approximation here we will have no denominator term and hence $d[0] = 1$ and $d[k] = 0$ for $k > 0$.

We choose to represent `Vx` and `Wy` in floating point matrices `V` and `W` respectively. The i-th row of `V` corresponds to the i-th entry $f(x, const)$ in `Vx`, which is stored in the following fashion:

$$\alpha, \beta, \tau, c[0], c[1], \ldots, c[MAXDEGREE], d[0], d[1], \ldots, d[MAXDEGREE]$$

where the number of columns required is M = 2×`MAXDEGREE` + 5. This form also holds for `Wy`.

Therefore the whole approximation for the current subregion will be represented by three floating-point matrices as follows:

`LDU`	nTerms-by-nTerms Matrix storing `L`, `D` and `U`.
`V`	nTerms-by-M Matrix representing `Vx`.
`W`	nTerms-by-M Matrix representing `Wy`.

The following code stores `L`, `D` and `U` into a single Matrix `LDU`:

```
> LDU := Matrix(nTerms, nTerms, datatype=float[8], L);
> for i from 1 to nTerms do
>    LDU[i,i] := D[i,i];
>    for j from i+1 to nTerms do
>       LDU[i,j] := U[i,j];
>    end do;
> end do; .
```

Note that since `L` is a unit lower triangular matrix and `U` is a unit upper triangular matrix, their diagonal entries are all 1.0. Hence we do not store the diagonal entries of `L` and `U` into `LDU` in order to save the space for `D`. When we extract `L`, `D` and `U` from `LDU` for evaluations we will add those diagonal entries 1.0 back to `L` and `U`.

We also should note that storing `L`, `D` and `U` in the single Matrix `LDU` does not mean that we need to form the matrix product `U`·`D`·`L`. We avoid forming the product because, as already discussed in section 5.4.3, the diagonal entries in `D` increase in magnitude from approximately 1 to 10^{15} (assuming the hardware floating `Digits` is 15), while `L` and `U` have entries of magnitude approximately 1. For numerical stability, we must evaluate formula (6.6) in a right-to-left order.

The following code shows how we store the corresponding information of approximation in `V`. Note that we use `nTerms2`, not `nTerms`, for the number of entries here. Diagonal entries in `D` beyond position `nTerms2` would be larger than 10^{15} (i.e., the reciprocal of `hfeps`) indicating terms beyond the roundoff error threshold.

Column 1 of `V` stores *tau* where xfact = x^{tau} (or yfact = y^{tau} for `W`). Columns 2 and 3 store alpha and beta where xvar = alpha·x + beta (or yvar = alpha·y + beta).

If xdencoef[1] = 0.0 which implies we are using polynomial approximation for this entry, then the remaining columns store the Chebyshev coefficients. Otherwise the remaining columns store the Chebyshev-Padé coefficients.

Note that for the polynomial approximation case, default entries 0.0 are correct for the remaining columns except for the entry corresponding to the denominator $d[0] = 1$. However, here we store 0.0 instead of 1 for $d[0]$ as a flag to indicate that it is a polynomial being represented.

```
> for k from 1 to nTerms2 do
>    (xfact, xnumcoef, xdencoef, xvar) := xUnivarApprox[k];
>    if type(xfact,`^`) and op(1,xfact) = x then
>       V[k,1] := op(2,xfact);
>    elif xfact = x then
>       V[k,1] := 1;
>    elif xfact = 1 then
>       V[k,1] := 0;
>    else
>       error "the factor %1 was expected to be a power of %2",
        xfact, x;
>    end if;
>    if not type(xvar,'linear'(x)) then
>       error "expression %1 was expected to be linear in %2",
        xvar, x;
>    end if;
>    V[k,2] := coeff(xvar, x, 1);
>    V[k,3] := coeff(xvar, x, 0);
>    for j from 1 to LinearAlgebra:-Dimension(xnumcoef) do
>       V[k, j+3] := xnumcoef[j];
>    end do;
```

```
>    if xdencoef[1]=0.0 then
>       V[k, M-MAXDEGREE] := 0.0;
>    else
>       for j from 1 to LinearAlgebra:-Dimension(xdencoef) do
>          V[k, j+4+MAXDEGREE] := xdencoef[j];
>       end do;
>    end if;
> end do; .
```

For succinctness we do not show the code for storing the approximation for `W`, which is actually very similar to that of `V`.

At last, we store the three floating-point arrays `LDU`, `V` and `W` for the current subregion into `arrayRep` as follows:

```
> arrayRep[index, 1] := LDU;
> arrayRep[index, 2] := V;
> arrayRep[index, 3] := W; .
```

6.10.3 `piecewiseCreate`

Given the arguments `(pwlist, x, y)`, the procedure `piecewiseCreate` forms the corresponding Maple piecewise construct as described in section 6.7.2.

After we have computed all the approximations for each subregion, indicating that `generatePwlist` has completed updating the global variables `pwlist` and `arrayRep`, we will finish our execution of `generatePwlist` and return to `generateApprox` which returns `(hash, arrayRep)` by the following code:

```
> return (piecewiseCreate(pwlist, (x,y)), eval(arrayRep)); .
```

Chapter 7

Experimental Results

We now present experimental results of using our approximation method for several bivariate functions in this chapter, and in doing so we can evaluate how well our toolkit works.

7.1 Testing Setup

Before proceeding to the concrete testing cases, we first see how to set up the coding as preparation work.

7.1.1 `fapprox`

The procedure `fapprox` is used to evaluate the approximation at a given point (x, y) and return the numerical result.

In `fapprox` we have the following global variables: `nTermsMax`, `maxDegActual`, `LDUarray`, `Varray`, `Warray`, `numcoef`, `dencoef`, `Vx`, `Wy`. We will need to initialize them before we call `fapprox` in the testing cases. We first determine the index of (x, y) in the approximation representation `arrayRep`. Note that `_condExpr` is a placeholder which has been replaced by the piecewise expression `hash` before executing this code.

```
> ind := _condExpr;
> maxDegPlus1 := maxDegActual + 1;
> nTermsVx := nTermsMax; .
```

We will now evaluate every entry, of which the total number is `nTermsMax`, for `Vx`. We need to extract the related information for the approximation, i.e., α, β,

τ, the coefficients for numerator and denominator, from `Varray`. Since the value of $0.0^{0.0}$ is not defined in Maple, we have to check if there is such special case and handle it. If in `Varray` there is a special value 0.0, as discussed in section 6.10.2, indicating the denominator of the approximation is 1, we then know that we should use polynomial approximation; otherwise we will numerically compute the entry as a rational approximation.

The following is the corresponding Maple code:

```
> for k from 1 to nTermsMax do
>   tau := Varray[ind,k,1];
>   alpha := Varray[ind,k,2];
>   beta := Varray[ind,k,3];
>   if alpha=0.0 and beta=0.0 then
>      nTermsVx := k-1;
>      break;
>   end if;
>   if tau = 0.0 then fact := 1.0 else fact := x^tau end if;
>   var := alpha*x + beta;
>   if Varray[ind, k, maxDegActual+5] = 0.0 then
>      for i from 1 to maxDegPlus1 do
>         numcoef[i] := Varray[ind, k, i+3];
>      end do;
>      Vx[k] := fact * `numapprox/evalhf/chebeval2`(numcoef,
            maxDegPlus1, var);
>   else
>      for i from 1 to maxDegPlus1 do
>         numcoef[i] := Varray[ind, k, i+3];
>         dencoef[i] := Varray[ind, k, maxDegPlus1+i+3];
>      end do;
>      Vx[k] := fact * `numapprox/evalhf/chebeval2`(numcoef,
            maxDegPlus1, var) /`numapprox/evalhf/chebeval2`(
            dencoef, maxDegPlus1, var);
>   end if;
> end do; .
```

We also evaluate `Wy` in a similar way.

Afterward we perform the matrix-vector multiplications such that `approx(x,y)` = $\texttt{Vx}^{\%\texttt{T}}\cdot\texttt{U}\cdot\texttt{D}\cdot\texttt{L}\cdot\texttt{Wy}$. We implement the multiplications in a right-to-left order of evaluation.

The dimension of the two vectors `numcoef` and `dencoef` is max(`maxDegActual`+1, `nTermsMax`). Size `maxDegActual`+1 is required for storing the numerator and denominator Chebyshev coefficients, and size `nTermsMax` is required because these

vectors are used for temporary storage during the matrix-vector multiplications for space efficiency.

We can simply use the Maple commands `LinearAlgebra:-MatrixVectorMultiply` and `LinearAlgebra:-DotProduct` when we evaluate points for error checking in the second phase; however, we are not allowed to use them for `fapprox` since the Maple compiler does not accept them. Therefore we have to use lower-level code for the multiplications as below. Note that the diagonal entries in `L` and `U` are implicitly 1.0.

```
> for i from 1 to nTermsVx do
>   numcoef[i] := Wy[i];
>   for j from i-1 by -1 to 1 do
>      numcoef[i] := numcoef[i] + LDUarray[ind,i,j] * Wy[j];
>   end do;
> end do;
> for i from 1 to nTermsVx do
>   numcoef[i] := LDUarray[ind,i,i] * numcoef[i];
> end do;
> for i from 1 to nTermsVx do
>   dencoef[i] := numcoef[i];
>   for j from i+1 to nTermsVx do
>      dencoef[i] := dencoef[i] + LDUarray[ind,i,j] * numcoef[j];
>   end do;
> end do;
> result := 0.0;
> for i from 1 to nTermsVx do
>   result := result + Vx[i] * dencoef[i];
> end do;
> return result; .
```

We will invoke `fapprox` in various modes for our tests: `evalf`, `evalhf`, and compiled. Since the compiler accepts an 'if' expression but not a piecewise expression, we developed a procedure `convertToIf` to convert a Maple piecewise expression into a nested 'if' expression. `convertToIf` also changes all constants to floats at hardware precision since the Maple compiler cannot handle fractions.

7.1.2 Preparation

At the beginning for testing we define the value of `hfDigits` corresponding to hardware floats and set `finaltol` for generating the approximations. We use `hfDigits` typically for the value of `Digits` here since we will test many functions with hardware floating point precision, but any other user-specified precision may be used.

```
> hfDigits := trunc(evalhf(Digits));
> Digits := hfDigits;
> finaltol := Float(5,-Digits); .
```

We also initialize the four global variables introduced in Chapter 6: `MAXDEGREE`, which controls the degree of polynomial and rational approximation; `MAXTERMS`, which controls the number of terms in a tensor product series; `GUARD_TensorInt2D`, the guard digits for the `TensorInt2D` computation; and `GUARD_chebyshev`, the guard digits for Chebyshev series computation. The values of `(MAXTERMS,MAXDEGREE)` here are just examples from the first testing case in this chapter. We may alter their values for different regions and functions as will be shown soon.

```
> MAXTERMS := 20;
> MAXDEGREE := 20;
> GUARD_TensorInt2D := 4;
> GUARD_chebyshev := 2; .
```

We then define the function and the region of approximation by the following code. In addition we make the protected function name of f be an inert name. We then invoke the main procedure `generateApprox` to get the approximation and the number of indices. Note that `evalhf` accepts a piecewise expression, but not an 'if' expression; while the compiler is just the opposite. Hence we will use the piecewise expression for `evalf` and `evalhf` mode, but use the 'if' expression for compiled mode. We take the example of Maple's `BesselJ`(x, y) function for f and $[0, 1] \times [\frac{1}{1000}, 1]$ for the region here since it will be our first experimental case to be shown in the next section.

```
> fname := BesselJ:
> unprotect(fname);
> unassign(fname);
> f := fname(x,y);
> fproc := unapply(f, (x,y));
> region := [x = 0 ..  1, y = 1/1000 ..  1];
> (hash, arrayRep) := generateApprox(f, region, 'rational=true',
      finaltol):
> Nindices := op(-1,hash) - 1;
> fapprox := subs(_condExpr = hash, eval(fapprox)):
> fapprox_compile := subs(_condExpr = convertToIf(hash),
      eval(fapprox_compile)): .
```

The Maple command `unprotect` is used to override the protection on the function name, and the command `unassign` sets the name given as input to an "unassigned name", which is a name that has no value other than its own name. Note that applying `evalf` to the function will continue to invoke the numerical evaluation code.

We then extract `LDU`, `V` and `W` from `arrayRep` into a numerical array datatype. We determine the actual number of terms for index `ind` by looking at the diagonal values of `LDUarrayLarge`. We also examine all denominator and numerator degrees to determine `maxDegActual` among all the numerator and denominator polynomials.

```
> maxDeg := MAXDEGREE;
> maxDegPlus1 := maxDeg + 1;
> nTerms := MAXTERMS;
> M := 2*maxDegPlus1 + 3;
> LDUarrayLarge := Array(1..Nindices,1..nTerms,1..nTerms,
      datatype=float[8]):
> VarrayLarge := Array(1..Nindices,1..nTerms,1..M,
      datatype=float[8]):
> WarrayLarge := Array(1..Nindices,1..nTerms,1..M,
      datatype=float[8]):
> (nTermsMax, maxDegActual) := (0, 0):
> for ind from 1 to Nindices do
>   LDUarrayLarge[ind, 1..nTerms, 1..nTerms] := arrayRep[ind, 1];
>   VarrayLarge[ind, 1..nTerms, 1..M] := arrayRep[ind, 2];
>   WarrayLarge[ind, 1..nTerms, 1..M] := arrayRep[ind, 3];
>   for k from nTerms to 1 by -1 while LDUarrayLarge[ind,k,k] =
      0.0 do end do;
>   nTermsMax := max(nTermsMax, k);
>   for k from 1 to nTermsMax do
>      for i from M by -1 to M-maxDeg while VarrayLarge[ind,k,i]
        = 0.0 and WarrayLarge[ind,k,i] = 0.0 do end do;
>      maxDegActual := max(maxDegActual, i-maxDegPlus1-4);
>      for i from M-maxDegPlus1 by -1 to 4 while VarrayLarge[ind,
        k,i] = 0.0 and WarrayLarge[ind,k,i] = 0.0 do end do;
>      maxDegActual := max(maxDegActual, i-4);
>   end do;
> end do:
> Mmax := 2*maxDegActual + 5; .
```

There may be some waste of space and time if we directly use the arrays `LDUarrayLarge`, `VarrayLarge` and `WarrayLarge` for procedure `fapprox` since these arrays will likely have a lot of zero entries if we just use `nTerms` and `M` for their sizes. In order to avoid this wasted space, we allocate smaller arrays `LDUarray`, `Varray` and `Warray` based on `nTermsMax` and `Mmax`, and shift the remaining elements for denominator coefficients in the arrays accordingly. We also initialize some vectors required within `fapprox`.

```
> LDUarray := Array(1..Nindices,1..nTermsMax,1..nTermsMax,
   LDUarrayLarge[1..Nindices,1..nTermsMax,1..nTermsMax],
   datatype=float[8]):
```

```
> Varray := Array(1..Nindices,1..nTermsMax,1..Mmax, VarrayLarge[1
   ..Nindices,1..nTermsMax,1..Mmax],datatype=float[8]):
> Warray := Array(1..Nindices,1..nTermsMax,1..Mmax, WarrayLarge[1
   ..Nindices,1..nTermsMax,1..Mmax],datatype=float[8]):
> for ind from 1 to Nindices do
>   for k from 1 to nTermsMax do
>      Varray[ind,k,(Mmax-maxDegActual)..Mmax] := VarrayLarge[ind,
         k,(M-maxDeg)..(M-maxDeg+maxDegActual)];
>      Warray[ind,k,(Mmax-maxDegActual)..Mmax] := WarrayLarge[ind,
         k,(M-maxDeg)..(M-maxDeg+maxDegActual)];
>   end do;
> end do;
> numcoef := Vector( max(maxDegActual+1,nTermsMax),
   datatype=float[8] ):
> dencoef := Vector( max(maxDegActual+1,nTermsMax),
   datatype=float[8] ):
> Vx := Vector( nTermsMax, datatype=float[8] ):
> Wy := Vector( nTermsMax, datatype=float[8] ): .
```

All of our experiments are run in Maple 11 on the scg.cs linux server, a Sun V40Z - quad AMD operating system with 16G ram and three 144G SCSI drives, of the University of Waterloo. Since each evaluation of a single point takes a relatively small amount of time, it is not reliable to use the CPU timing of only one computation. Thus we randomly select 70 points on each x and y region and evaluate $70 \times 70 = 4900$ (a quite large number) points for our experiments. We also insert the four boundary points of the rectangular region into the list of evaluation points.

```
> (Nx, Ny) := (70, 70);
> (xmin, xmax) := op(rhs(region[1]));
> (ymin, ymax) := op(rhs(region[2]));
> xvector := LinearAlgebra:-RandomVector( Nx, 'generator'=evalf(
   xmin ..  xmax), 'outputoptions'=[datatype=float[8]] ):
> yvector := LinearAlgebra:-RandomVector( Ny, 'generator'=evalf(
   ymin ..  ymax), 'outputoptions'=[datatype=float[8]] ):
> xvector[1] := evalf(xmin):
> xvector[Nx] := evalf(xmax):
> yvector[1] := evalf(ymin):
> yvector[Ny] := evalf(ymax):
```

We then define procedure `testEval` for the timing tests. In `testEval` we repeatedly invoke `fapprox` to numerically evaluate each point and store the values in a matrix `Values` to allow accuracy checking later. The main loop in `testEval` is as follows:

```
> for i from 1 to Nx do
```

```
>   for j from 1 to Ny do
>      Values[i,j] := fapprox(xvector[i], yvector[j]);
>   end do;
> end do;
```

For comparison purposes, we test four timings for evaluating f at the 4900 points: the original function in Maple, `evalf` mode, `evalhf` mode, and compiled code. We are mainly concerned about how much faster we can evaluate by using compiled code than by using the original function in Maple.

When testing in `evalf` mode, we use extra guard digits to achieve full accuracy since evaluation speed is not sensitive to this change in software floats. We also create arrays with `sfloat` entries for a fair timing of `evalf` mode.

Note that when we test in `evalhf` mode, we should not simply put the statement `evalhf(fapprox(xvector[i], yvector[j]))` in the procedure `testEval`. This is because the `for` loops would be done in the default Maple mode while `evalhf` mode is run in hardware floats, thus the above code would cause conversions between software and hardware float environments which takes a lot of extra time. Therefore we should use one single call of `evalhf(testEval)` to avoid the conversion overhead.

In order to see if our approximation satisfies the required precision in each mode, we also check the errors at all the evaluation points between Maple's function, which we assume to be the correct value, and each of the three different modes.

```
> (maxabserr, normf) := (0.0, 0.0):
> for i from 1 to Nx do
>   for j from 1 to Ny do
>      val := Values[i,j];
>      val_correct := evalf( fproc(xvector[i], yvector[j]) );
>      abserr := abs(val_correct - val);
>      maxabserr := max(maxabserr, abserr);
>      normf := max(normf, abs(val_correct));
>   end do;
> end do;
> maxerr := maxabserr/max(1.0,normf); .
```

7.2 Results for $J_x(y)$

We use the Bessel function of the first kind, $J_x(y)$, as our first example. The corresponding command for this function in Maple is `BesselJ(x,y)`.

As discussed before, we will stay away from singularities in our approximation since special code such as series expansion would be used for evaluation near a

singularity. Therefore, among all the points that we evaluated in our experiments the closest one to the singularity $(0,0)$ of `BesselJ(x,y)` is $(0, \frac{1}{1000})$.

We tested various regions to prove that our toolkit is scalable to work on any region. Table 7.1 shows the timings and corresponding maximum errors for rational approximations of $J_x(y)$ on these regions.

For each test, we recorded the timing of the original function `BesselJ`, the timing of `evalf` mode, the timing of `evalhf` mode and the timing of compiled code for our approximation. We also show the speedup factors between the timings so that we can know how much improvement we have obtained by using our toolkit. Since different values of `MAXTERMS` and `MAXDEGREE` may result in different errors and timings, we also included their values for each test.

Although what we really care about is the timings for evaluating points, we also included the timing for generating the approximations and the number of subregions for each test in the table so that readers can see the difference more clearly. Theoretically, the smaller values of `MAXTERMS` and `MAXDEGREE` that we set, the larger number of subregions we will have and the more time we need to spend on the approximation generation.

Table 7.1 includes the results of 5 testing cases, for which we use the hardware floating point precision as the desired accuracy. The `hfDigits` is 15 in our machine, hence the desired precision is 5×10^{-15}. As we can see in the table, all the maximum errors are smaller than the precision as expected and the overall speedup factor of our toolkit is around 11.

In the first two cases, we tried different values of `MAXTERMS` and `MAXDEGREE` for the same small region $[0,1] \times [\frac{1}{1000}, 1]$. When `(MAXTERMS, MAXDEGREE)` $= (10, 10)$, the time it took to generate the approximation is almost 5 times as that of using `(MAXTERMS, MAXDEGREE)` $= (20, 20)$ since the former divided the region into 22 subregions while the latter did not have any subdivision at all. However, the second case is faster than the first one in evaluating points, hence using smaller values of `MAXTERMS` and `MAXDEGREE` should achieve better results in terms of evaluation time.

Since this region is already fairly close to the singularity $(0,0)$, we did not expect to achieve the accuracy; however, the result turned out to be quite good. The second test proves that using our toolkit in compiled code is around 12 times faster than Maple's `BesselJ` method.

The third test shows how our toolkit works on another small region which is away from the singularity. Again the compiled code achieves faster speed than the original function.

The last two tests show that our toolkit works properly on large regions. Note that the two cases have successive regions, which indicates that we can partition the x-y plane into some large regions and use our toolkit on each large region.

Regions	(a,b)=(0,1) (c,d)=($\frac{1}{1000}$,1)	(a,b)=(0,1) (c,d)=($\frac{1}{1000}$,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(0,15) (c,d)=($\frac{1}{8}$,15)	(a,b)=(0,15) (c,d)=(15,20)
(`MAXTERMS, MAXDEGREE`)	(20, 20)	(10, 10)	(7, 7)	(15,15)	(20,15)
Number of Subregions	1	22	5	44	23
Approximation Generation Time (s)	18.137	95.069	16.609	676.246	445.755
$\|\|f\|\|_\infty$	1.00	1.00	0.50	1.00	0.28
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	3.79e-15	4.19e-15	3.54e-15	4.35e-15	3.06e-15
Original Maple Time (s)	1.612	1.625	1.620	1.952	1.976
`evalf` Time (s)	30.438	13.916	8.876	26.894	22.690
`evalhf` Time (s)	3.064	1.276	0.840	2.280	1.996
Compiled Time (s)	0.244	0.132	0.104	0.188	0.180
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	9.935	10.91	10.57	11.79	11.37
Speedup Factor (`evalhf`/compiled)	12.56	9.667	8.077	12.13	11.09
Overall Speedup (origMaple/compiled)	6.607	12.31	15.58	10.38	10.98
Original Maple Plot Time (s)	0.216	0.232	0.228	0.232	0.256
Approximation Plot Time (s)	0.040	0.028	0.024	0.036	0.032
Plot Speedup Factor	5.400	8.286	9.500	6.444	8.000

Table 7.1: Timings and Errors for Approximating $J_x(y)$ Using Rational Approximation

We also have included in the table the timings for plotting the original function in Maple and plotting our approximation in compiled code. The speedup factor is around 8.

Table 7.2 shows the experimental results for using polynomial approximations for $J_x(y)$.

Comparing the first case in this table and the second case in Table 7.1, both on the same region near the singularity, we can see that the former took less time in generating the approximation while achieving larger speedup factor for evaluation time than the latter.

The second and third cases tested different values of `MAXTERMS` and `MAXDEGREE` on the same region. As we can see, using `(MAXTERMS, MAXDEGREE)` = (15, 15) is better than `(MAXTERMS, MAXDEGREE)` = (20, 30) in terms of evaluation timings, which is similar to our observations in the rational case. The last two testing cases are again two successive large regions which show our approximation using polynomials works properly on large regions as expected.

7.3 Results for $Y_x(y)$

Table 7.3 and Table 7.4 include the testing results for the Bessel function of the second kind, $Y_x(y)$ or `BesselY(x,y)`, using rational and polynomial approximations respectively. Again we tested on various regions, from small to large, and used different values of `MAXTERMS` and `MAXDEGREE`, which result in diverse evaluation speeds as the tables show.

7.4 Results for $\beta(x, y)$

The experimental results for the Beta function, $\beta(x, y)$ or `Beta(x,y)`, are shown in Table 7.5 and Table 7.6.

Comparing the two tables, we can see that the closest point to the singularity of $\beta(x, y)$ is $(\frac{1}{100}, \frac{1}{100})$ when using polynomial approximation, while with rational approximation the closest point is only $(\frac{1}{10}, \frac{1}{10})$. For other regions which are far away from the singularity, we see that polynomial approximation took much less time in generating the approximation than that of rational approximation. However, since rational approximation achieved a little faster speeds than polynomial approximation in most of these cases, we may still prefer to use rational approximation for $\beta(x, y)$.

Regions	(a,b)=(0,1) (c,d)=($\frac{1}{1000}$,1)	(a,b)=(1,10) (c,d)=(1,10)	(a,b)=(1,10) (c,d)=(1,10)	(a,b)=(0,15) (c,d)=($\frac{1}{8}$,15)	(a,b)=(0,15) (c,d)=(15,25)
(`MAXTERMS, MAXDEGREE`)	(10, 20)	(15, 15)	(20, 30)	(15,25)	(15,25)
Number of Subregions	22	48	4	39	22
Approximation Generation Time (s)	74.708	260.492	65.348	465.885	281.317
$\|\|f\|\|_\infty$	1.00	0.58	0.58	1.00	0.28
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	4.18e-15	3.92e-15	2.57e-15	4.56e-15	4.76e-15
Original Maple Time (s)	1.708	1.780	1.908	1.892	2.120
`evalf` Time (s)	13.957	17.737	38.966	25.109	27.358
`evalhf` Time (s)	1.160	1.176	3.084	1.840	2.012
Compiled Time (s)	0.128	0.124	0.252	0.176	0.180
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	12.03	15.09	12.64	13.65	13.60
Speedup Factor (`evalhf`/compiled)	9.062	9.484	12.24	10.45	11.18
Overall Speedup (origMaple/compiled)	13.34	14.35	7.571	10.75	11.78
Original Maple Plot Time (s)	0.216	0.216	0.252	0.264	0.292
Approximation Plot Time (s)	0.028	0.028	0.048	0.036	0.036
Plot Speedup Factor	7.714	7.714	5.250	7.333	8.111

Table 7.2: Timings and Errors for Approximating $J_x(y)$ Using Polynomial Approximation

Regions	(a,b)=(0,1) (c,d)=($\frac{1}{1000}$,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(4,7) (c,d)=(2,4)	(a,b)=(0,10) (c,d)=($\frac{1}{4}$,10)	(a,b)=(0,10) (c,d)=(10,30)
(`MAXTERMS, MAXDEGREE`)	(15,10)	(15,15)	(8,8)	(15,10)	(15,15)	(15,15)
Number of Subregions	20	1	4	1	46	10
Approximation Generation Time (s)	222.525	7.216	22.093	15.072	514.120	163.974
$\|f\|_\infty$	636.62	0.42	0.42	271.55	1.24e14	0.36
Max Error ($\|f - approx\|_\infty / \max(\|f\|_\infty, 1)$)	3.30e-16	3.16e-15	5.21e-15	3.01e-15	4.27e-15	2.05e-15
Original Maple Time (s)	3.929	4.536	4.544	4.693	4.996	6.652
`evalf` Time (s)	14.853	10.745	9.321	21.109	33.258	33.162
`evalhf` Time (s)	1.428	1.144	0.896	2.008	2.952	2.884
Compiled Time (s)	0.136	0.116	0.108	0.184	0.232	0.228
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	10.40	9.388	10.40	10.51	11.27	11.50
Speedup Factor (`evalhf`/compiled)	10.50	9.862	8.296	10.91	12.72	12.65
Overall Speedup (origMaple/compiled)	28.89	39.10	42.07	25.51	21.53	29.18
Original Maple Plot Time (s)	0.532	0.632	0.792	0.852	0.716	0.832
Approximation Plot Time (s)	0.028	0.024	0.024	0.036	0.036	0.040
Plot Speedup Factor	19.00	26.33	33.00	23.67	19.89	20.80

Table 7.3: Timings and Errors for Approximating $Y_x(y)$ Using Rational Approximation

Regions	(a,b)=(0,1) (c,d)=($\frac{1}{100}$,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(4,7) (c,d)=(2,4)	(a,b)=(1,6.5) (c,d)=(1,6.5)	(a,b)=(0,10) (c,d)=($\frac{1}{4}$,10)	(a,b)=(0,10) (c,d)=(10,30)
(`MAXTERMS, MAXDEGREE`)	(15,25)	(15,20)	(15,20)	(15,25)	(15,30)	(15,30)
Number of Subregions	7	1	2	7	47	10
Approximation Generation Time (s)	68.364	5.840	21.169	74.908	426.494	128.488
$\|\|f\|\|_\infty$	63.68	0.42	271.55	8.68e3	1.24e14	0.36
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	4.71e-16	3.53e-15	2.61e-15	4.47e-15	4.91e-15	2.20e-15
Original Maple Time (s)	4.104	4.588	4.824	4.696	4.896	6.948
`evalf` Time (s)	17.517	10.517	18.929	24.814	32.623	33.538
`evalhf` Time (s)	1.788	1.064	1.628	2.056	2.800	2.612
Compiled Time (s)	0.160	0.116	0.160	0.188	0.228	0.224
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	9.799	9.887	11.63	12.07	11.65	12.84
Speedup Factor (`evalhf`/compiled)	11.18	9.172	10.18	10.94	12.28	11.66
Overall Speedup (origMaple/compiled)	25.65	39.55	30.15	24.98	21.47	31.02
Original Maple Plot Time (s)	0.608	0.636	0.808	0.636	0.744	1.036
Approximation Plot Time (s)	0.032	0.024	0.144	0.032	0.208	0.040
Plot Speedup Factor	19.00	26.50	5.611	19.88	3.577	25.90

Table 7.4: Timings and Errors for Approximating $Y_x(y)$ Using Polynomial Approximation

Regions	(a,b)=($\frac{1}{10}$,1) (c,d)=($\frac{1}{10}$,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(4,7) (c,d)=(2,4)	(a,b)=(4,7) (c,d)=(2,4)	(a,b)=($\frac{1}{2}$,10) (c,d)=($\frac{1}{2}$,10)	(a,b)=($\frac{1}{2}$,10) (c,d)=(10,20)
(`MAXTERMS, MAXDEGREE`)	(15,15)	(15,15)	(10,10)	(15,15)	(15,15)	(15,15)
Number of Subregions	25	2	31	8	134	29
Approximation Generation Time (s)	518.216	12.836	187.195	63.439	1517.578	252.943
$\|\|f\|\|_\infty$	19.71	0.33	0.05	0.05	3.14	0.57
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	4.54e-15	1.40e-15	2.54e-16	2.66e-15	3.98e-15	4.93e-15
Original Maple Time (s)	4.988	4.993	5.033	5.565	6.057	4.741
`evalf` Time (s)	8.561	10.576	7.376	10.232	17.049	10.520
`evalhf` Time (s)	1.032	1.472	0.940	1.397	1.612	1.348
Compiled Time (s)	0.108	0.132	0.104	0.136	0.148	0.128
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	8.296	7.188	7.847	7.323	10.58	7.804
Speedup Factor (`evalhf`/compiled)	9.556	11.15	9.038	10.27	10.89	10.53
Overall Speedup (origMaple/compiled)	46.19	37.83	48.39	40.92	40.93	37.04
Original Maple Plot Time (s)	0.128	0.148	0.664	0.692	0.200	0.704
Approximation Plot Time (s)	0.024	0.028	0.024	0.024	0.032	0.024
Plot Speedup Factor	5.333	5.286	27.67	28.83	6.250	29.33

Table 7.5: Timings and Errors for Approximating $\beta(x, y)$ Using Rational Approximation

Regions	(a,b)=($\frac{1}{100}$,1) (c,d)=($\frac{1}{100}$,1)	(a,b)=($\frac{1}{10}$,1) (c,d)=($\frac{1}{10}$,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(4,7) (c,d)=(2,4)	(a,b)=($\frac{1}{2}$,10) (c,d)=($\frac{1}{2}$,10)	(a,b)=($\frac{1}{2}$,10) (c,d)=(10,25)
(`MAXTERMS, MAXDEGREE`)	(15,30)	(15,25)	(15,25)	(15,30)	(15,30)	(15,30)
Number of Subregions	73	16	1	1	31	18
Approximation Generation Time (s)	270.752	73.980	3.908	5.932	221.941	112.559
$\|\|f\|\|_\infty$	199.97	19.71	0.33	0.05	3.14	0.57
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	4.00e-16	1.61e-15	2.60e-16	5.06e-17	9.87e-16	2.63e-15
Original Maple Time (s)	5.281	5.324	5.140	4.920	4.961	5.112
`evalf` Time (s)	13.717	10.533	11.881	15.517	17.985	13.909
`evalhf` Time (s)	1.256	1.144	1.356	1.760	1.728	1.356
Compiled Time (s)	0.124	0.116	0.136	0.148	0.156	0.132
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	10.92	9.205	8.761	8.818	10.41	10.26
Speedup Factor (`evalhf`/compiled)	10.13	9.862	9.971	11.89	11.08	10.27
Overall Speedup (origMaple/compiled)	42.59	45.90	37.79	33.24	31.80	38.73
Original Maple Plot Time (s)	0.148	0.132	0.148	0.448	0.192	0.432
Approximation Plot Time (s)	0.024	0.024	0.024	0.028	0.032	0.028
Plot Speedup Factor	6.167	5.500	6.167	16.00	6.000	15.43

Table 7.6: Timings and Errors for Approximating $\beta(x, y)$ Using Polynomial Approximation

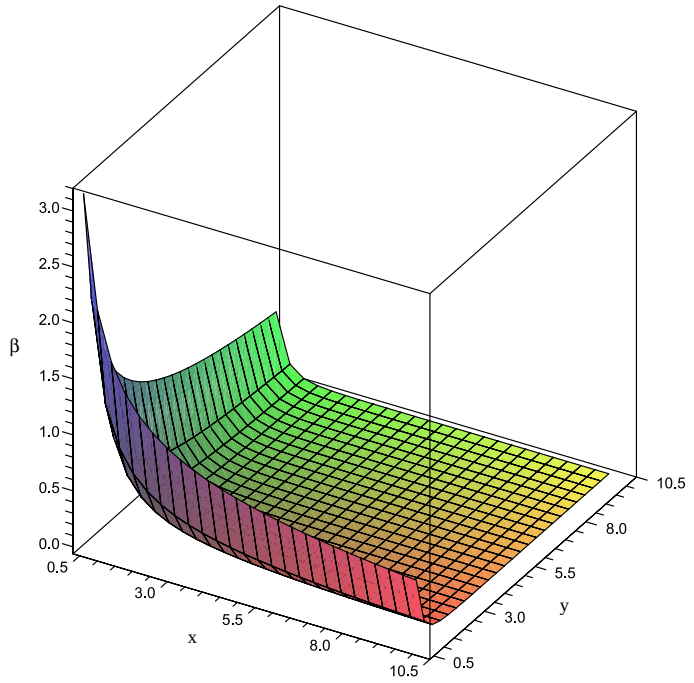

Figure 7.1: Plot for $\beta(x, y)$

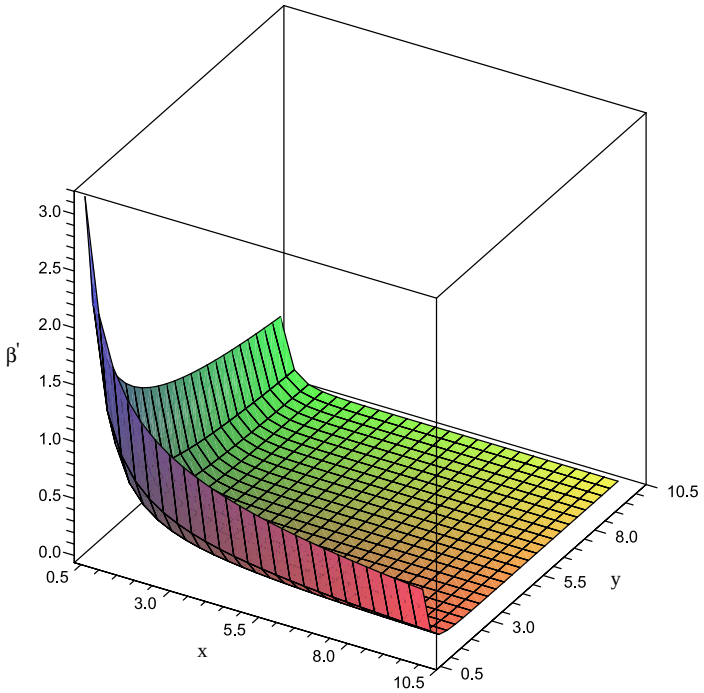

Figure 7.2: Plot for Our Approximation $\beta'(x, y)$

Figure 7.1 and Figure 7.2 are the graphs for plotting $\beta(x, y)$ and our approximation $\beta'(x, y)$ respectively on the region $[\frac{1}{2}, 10] \times [\frac{1}{2}, 10]$.

7.5 Results for `JacobiSN`(x, y)

We also tested the Jacobi elliptic function `JacobiSN`(x, y) in Maple. The results are shown in Table 7.7 and Table 7.8. Note that the regions are relatively small since `JacobiSN`(x, y) is very curvy and generating approximations takes much time on large regions for this function.

7.6 Results for `LegendreP`(x, y)

Table 7.9 and Table 7.10 show the results for the Legendre function of the first kind, `LegendreP`(x, y). Comparing the last two cases of large regions in the two tables, we see that the polynomial approximation is able to achieve the desired accuracy while the rational approximation fails. Additionally, the polynomial approximation took much less time for generating the approximation than the rational approximation (but still results in similar speedup factors) in these two cases. Therefore, for `LegendreP`(x, y) using polynomial approximation is advantageous over the rational approximation in the sense of being more time efficient for approximation generation and having better accuracy.

7.7 A Boundary Value Problem

Besides working on Maple's built-in functions as shown in the above sections, our toolkit can also generate approximations for user-defined functions. We show this by an example of a boundary value problem (BVP) here.

Consider the function $H(\alpha, x)$ defined by the following nonlinear BVP on $0 \leq x \leq 1$:

$$\begin{cases} y''(x) + \alpha\,|y(x)| = 0, \\ y'(0) = 1, \\ y(1) = -1. \end{cases} \tag{7.1}$$

Specifically, the value $H(\alpha, x)$ is defined by solving the BVP with parameter α and then returning the value $y(x)$. We can define the function $H(\alpha, x)$ by a procedure `H(alpha, tval)` which numerically solves the BVP to obtain the function value in Maple.

Regions	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(0,2.5) (c,d)=(0,2.5)	(a,b)=(0,2.5) (c,d)=(2.5,3.5)
(`MAXTERMS, MAXDEGREE`)	(20,15)	(9,8)	(15,15)	(20,15)	(20,20)
Number of Subregions	1	7	4	7	4
Approximation Generation Time (s)	14.968	71.540	189.127	464.509	279.341
$\|\|f\|\|_\infty$	0.84	0.84	0.33	1.00	0.40
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	6.50e-16	2.82e-15	1.24e-15	5.67e-15	3.83e-15
Original Maple Time (s)	20.642	23.397	46.695	42.822	54.199
`evalf` Time (s)	11.624	7.241	18.785	25.566	32.106
`evalhf` Time (s)	1.397	0.784	1.965	2.956	4.061
Compiled Time (s)	0.128	0.100	0.160	0.224	0.312
Speedup Factor (origMaple/`evalf`)	1.776	3.232	2.487	1.675	1.688
Speedup Factor (`evalf/evalhf`)	8.318	9.236	9.557	8.650	7.907
Speedup Factor (`evalhf`/compiled)	10.91	7.840	12.28	13.20	13.02
Overall Speedup (origMaple/compiled)	161.2	234.0	291.9	191.2	173.7
Original Maple Plot Time (s)	2.572	3.108	4.416	3.776	4.920
Approximation Plot Time (s)	0.028	0.024	0.028	0.036	0.048
Plot Speedup Factor	91.86	129.5	157.7	104.9	102.5

Table 7.7: Timings and Errors for Approximating `JacobiSN`(x, y) Using Rational Approximation

Regions	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(0,2.5) (c,d)=(0,2.5)	(a,b)=(0,2.5) (c,d)=(2.5,3.5)
(`MAXTERMS, MAXDEGREE`)	(20,30)	(20,35)	(10,20)	(20,35)	(20,35)
Number of Subregions	1	4	27	8	6
Approximation Generation Time (s)	13.656	163.050	711.220	496.903	315.723
$\|\|f\|\|_\infty$	0.84	0.33	0.33	1.00	0.40
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	5.80e-16	1.23e-15	4.08e-15	4.86e-15	3.94e-15
Original Maple Time (s)	22.466	44.166	47.367	48.119	61.828
`evalf` Time (s)	13.537	20.102	12.721	28.697	28.342
`evalhf` Time (s)	1.468	2.264	1.256	2.941	3.156
Compiled Time (s)	0.144	0.348	0.132	0.224	0.416
Speedup Factor (origMaple/`evalf`)	1.660	2.198	3.724	1.677	2.182
Speedup Factor (`evalf/evalhf`)	9.223	8.878	10.13	9.759	8.980
Speedup Factor (`evalhf`/compiled)	10.19	6.506	9.515	13.13	7.587
Overall Speedup (origMaple/compiled)	156.0	126.9	358.9	214.8	148.6
Original Maple Plot Time (s)	2.756	4.176	4.880	4.064	4.484
Approximation Plot Time (s)	0.032	0.032	0.024	0.032	0.040
Plot Speedup Factor	86.12	130.5	203.3	127.0	112.1

Table 7.8: Timings and Errors for Approximating `JacobiSN`(x, y) Using Polynomial Approximation

Regions	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(1,2) (c,d)=(3,4)	(a,b)=(-2,2) (c,d)=(0,3)	(a,b)=(-10,10) (c,d)=(0,15)	(a,b)=(10,20) (c,d)=(0,15)
(`MAXTERMS, MAXDEGREE`)	(20,20)	(15,15)	(10,10)	(15,15)	(15,15)
Number of Subregions	1	1	9	45	80
Approximation Generation Time (s)	5.932	7.748	56.723	772.560	960.540
$\|\|f\|\|_\infty$	1.00	23.50	13.00	1.03e14	4.28e28
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	1.40e-15	9.62e-16	3.13e-15	3.21e-09	1.65e-13
Original Maple Time (s)	1.792	4.792	3.172	4.756	5.344
`evalf` Time (s)	9.773	8.353	11.373	21.333	21.769
`evalhf` Time (s)	1.172	1.024	1.224	2.341	2.112
Compiled Time (s)	0.116	0.108	0.128	0.196	0.188
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	8.339	8.157	9.289	9.111	10.31
Speedup Factor (`evalhf`/compiled)	10.10	9.481	9.562	11.94	11.23
Overall Speedup (origMaple/compiled)	15.45	44.37	24.78	24.27	28.43
Original Maple Plot Time (s)	0.256	0.832	0.768	0.888	0.732
Approximation Plot Time (s)	0.024	0.020	0.028	0.036	0.032
Plot Speedup Factor	10.67	41.60	27.43	24.67	22.88

Table 7.9: Timings and Errors for Approximating `LegendreP`(x, y) Using Rational Approximation

Regions	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(-2,2) (c,d)=(0,3)	(a,b)=(-2,2) (c,d)=(0,3)	(a,b)=(-10,10) (c,d)=(0,15)	(a,b)=(10,20) (c,d)=(0,15)
(`MAXTERMS, MAXDEGREE`)	(15,30)	(10,15)	(15,25)	(20,30)	(20,30)
Number of Subregions	1	23	3	34	6
Approximation Generation Time (s)	4.028	102.274	27.101	394.052	85.669
$\|\|f\|\|_\infty$	1.00	13.00	13.00	1.03e14	4.28e28
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	1.08e-15	5.23e-16	3.18e-15	1.46e-15	3.06e-15
Original Maple Time (s)	1.844	3.104	3.156	4.740	5.028
`evalf` Time (s)	12.077	9.257	14.437	23.742	20.265
`evalhf` Time (s)	1.408	0.948	1.696	2.428	2.313
Compiled Time (s)	0.140	0.112	0.152	0.204	0.180
Speedup Factor (origMaple/`evalf`)	-	-	-	-	-
Speedup Factor (`evalf/evalhf`)	8.580	9.765	8.514	9.778	8.759
Speedup Factor (`evalhf`/compiled)	10.06	8.464	11.16	11.90	12.85
Overall Speedup (origMaple/compiled)	13.17	27.71	20.76	23.24	27.93
Original Maple Plot Time (s)	0.292	0.508	0.676	0.980	0.900
Approximation Plot Time (s)	0.032	0.024	0.028	0.036	0.032
Plot Speedup Factor	9.125	21.17	24.14	27.22	28.12

Table 7.10: Timings and Errors for Approximating `LegendreP`(x, y) Using Polynomial Approximation

```
> H := proc (alpha, tval)
>   local t, u, soln;
>   soln := dsolve({(D@@2)(u)(t)+alpha*abs(u(t))=0, D(u)(0)=1,
     u(1)=-1}, numeric);
>   eval(u(t), soln(tval));
> end proc; .
```

Table 7.11 and Table 7.12 show the experimental results of using our toolkit for $H(\alpha, x)$ with rational and polynomial approximations respectively. For this specific example, the largest x region is $[0, 1]$ while the α region can be flexible.

Note that the desired precision here is 5×10^{-6}, which is different from all the tests in the above sections, since this is the default value of error tolerance for the numerical solution of a BVP when using the `dsolve` command in Maple.

As seen in the tables, the computation of the original function $H(\alpha, x)$ is very slow, which is because each new evaluation point requires the numerical solution of a BVP. However, using the approximation generated by our toolkit, we are able to achieve almost 1900 times faster computation on average.

Figure 7.3 and Figure 7.4 are the graphs for plotting $H(\alpha, x)$ and our approximation $H'(\alpha, x)$ respectively on the region $[-2, 2] \times [0, 1]$.

7.8 Conclusions

Based on the results shown in the above tests, we are able to see that our toolkit works well for both Maple's built-in functions and user-defined functions. For all the functions in the experiments, our toolkit can achieve the specified accuracy, both hardware floating precision and user-defined precision, in all the three modes.

For Maple's built-in functions, although `evalf` mode and `evalhf` mode sometimes do not evaluate faster than the original functions in our tests, we are still able to have the compiled code always run faster than the original functions. The compiled code of our toolkit has gained from 11 to 200 times speedup compared with the original function evaluation.

For our BVP example, all three modes achieve faster calculation speeds than the original function: `evalf` mode is around 35 times faster than the original function on average, `evalhf` mode is around 9 times faster than `evalf` mode on average, and the compiled code is around 6 times faster than `evalhf` mode on average. This results in almost 1900 times speedup compared with the original function evaluation in Maple, which is a significant improvement. We also obtained more than 1000 times speedup for plotting with our approximation compared with the original function in Maple.

Regions	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(-2,2) (c,d)=(0,1)	(a,b)=(0,4) (c,d)=(0,1)	(a,b)=(4,10) (c,d)=(0,1)
(`MAXTERMS, MAXDEGREE`)	(15,20)	(10,10)	(15,20)	(20,20)	(20,20)
Number of Subregions	2	17	4	4	1
Approximation Generation Time (s)	147.997	1977.65	343.361	372.447	52.931
$\|\|f\|\|_\infty$	2.00	2.00	10.89	2.00	1.00
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	3.00e-07	2.80e-06	2.19e-07	1.25e-06	4.58e-06
Original Maple Time (s)	173.031	173.031	195.968	195.672	150.729
`evalf` Time (s)	5.556	2.644	6.029	6.296	6.461
`evalhf` Time (s)	0.732	0.372	0.844	0.905	0.836
Compiled Time (s)	0.108	0.092	0.108	0.108	0.096
Speedup Factor (origMaple/`evalf`)	31.14	65.4	32.51	31.08	23.32
Speedup Factor (`evalf/evalhf`)	7.590	7.108	7.143	6.957	7.728
Speedup Factor (`evalhf`/compiled)	6.778	4.043	7.815	8.380	8.708
Overall Speedup (origMaple/compiled)	1602.0	1880.8	1814.5	1811.8	1570.1
Original Maple Plot Time (s)	31.777	46.930	28.605	29.577	21.433
Approximation Plot Time (s)	0.032	0.028	0.028	0.028	0.024
Plot Speedup Factor	993.1	1676.1	1021.6	1056.3	893.04

Table 7.11: Timings and Errors for Approximating $H(\alpha, x)$ Using Rational Approximation

Regions	(a,b)=(0,1) (c,d)=(0,1)	(a,b)=(-2,2) (c,d)=(0,1)	(a,b)=(-2,2) (c,d)=(0,1)	(a,b)=(0,4) (c,d)=(0,1)	(a,b)=(4,13) (c,d)=(0,1)
(`MAXTERMS, MAXDEGREE`)	(20,35)	(20,35)	(10,15)	(20,35)	(20,35)
Number of Subregions	2	4	31	4	10
Approximation Generation Time (s)	139.892	350.829	3723.06	371.103	1929.62
$\|\|f\|\|_\infty$	2.00	10.89	10.89	2.00	1.00
Max Error ($\|\|f - approx\|\|_\infty / \max(\|\|f\|\|_\infty, 1)$)	3.50e-07	2.17e-07	1.21e-06	1.18e-06	3.75e-06
Original Maple Time (s)	165.843	198.613	198.613	194.268	194.614
`evalf` Time (s)	3.944	5.996	7.621	4.816	6.592
`evalhf` Time (s)	0.488	0.620	0.432	0.556	0.561
Compiled Time (s)	0.084	0.104	0.100	0.096	0.100
Speedup Factor (origMaple/`evalf`)	42.04	33.12	26.06	40.34	29.52
Speedup Factor (`evalf/evalhf`)	8.082	9.671	17.64	8.662	11.75
Speedup Factor (`evalhf`/compiled)	5.810	5.962	4.320	5.792	5.610
Overall Speedup (origMaple/compiled)	1974.3	1909.7	1986.1	2023.6	1946.1
Original Maple Plot Time (s)	24.653	34.206	54.731	34.298	39.142
Approximation Plot Time (s)	0.028	0.032	0.032	0.028	0.028
Plot Speedup Factor	880.46	1068.9	1710.3	1224.9	1397.9

Table 7.12: Timings and Errors for Approximating $H(\alpha, x)$ Using Polynomial Approximation

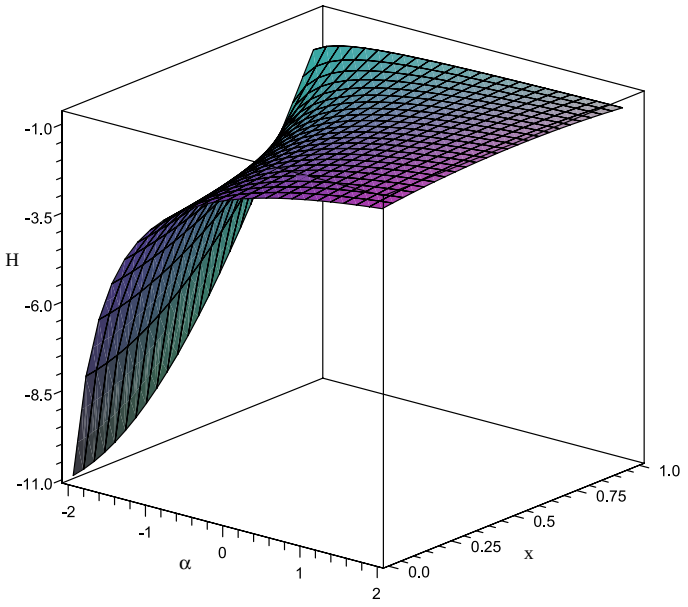

Figure 7.3: Plot for $H(\alpha, x)$ defined by a BVP

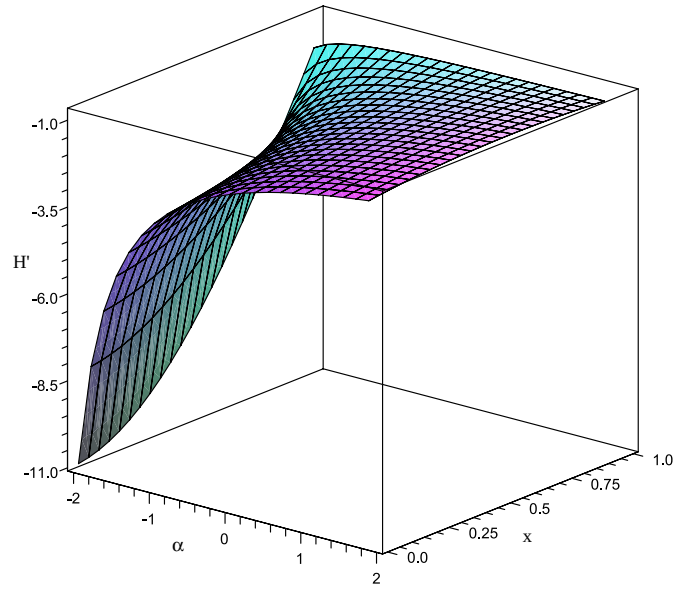

Figure 7.4: Plot for Our Approximation $H'(\alpha, x)$

Chapter 8

Future Work

In this chapter we mention several areas of investigation which may improve the performance of our toolkit by further development.

8.1 Singularities

As stated in the above two chapters, when using our toolkit we avoided approximating a function within a certain distance from its singularity. The rational and polynomial approximations that we mainly use in our package work well to meet the minimax criterion, but they cannot get results accurately enough when very close to function singularities.

Since a minimax approximation cannot be useful on a certain region around the singularity, a totally different type of method, such as a generalized series expansion, will need to be explored in order to approximate the value of the function for such special cases.

8.2 Maximum Number of Terms and Degree

As shown in Chapter 7, we choose diverse values for `MAXDEGREE` and `MAXTERMS`, the limit of maximum degree for univariate approximations and number of terms in tensor product series, in each testing case. These values are experiential results which we believe is a good choice; however, there may be some other values that can make the toolkit work faster with less subregions.

It will be optimal if the toolkit can automatically decide which values we should use individually for the specified function to be approximated. Further investigations will be needed on this aspect.

8.3 Region Subdivision

In this thesis we have already developed a region subdivision algorithm that works well for having a polynomial or rational approximation within a certain region. However, this algorithm may not be the optimal one since it is possible that the function has some sections where it changes relatively slowly. It would be better if our algorithm can estimate the appropriate size of region in such cases instead of always dividing the region by half.

We have two possible solutions to this problem but have not done experiments with them yet:

1. Detect the derivative which implies if the function changes slowly in the sub-region and if it does, we can expand that region. We will need to do more research on this method.

2. Always attempt a larger region when having a region with approximation of degree $\leq$ `MAXDEGREE` or number of terms $\leq$ `MAXTERMS` before finally deciding the region size. This method is pretty straightforward and easy to implement but may result in a fairly slow algorithm.

8.4 Complex Approximations

Currently our toolkit only works for functions on the real region; however, it is possible to extend it to approximation of complex functions, such as the Hankel functions $H_v^{(1)}(x) = J_v(x) + i\,Y_v(x)$ and $H_v^{(2)}(x) = J_v(x) - i\,Y_v(x)$.

It should be possible to achieve such approximation by calling our toolkit for the real part and imaginary part of the function separately. But plenty of research needs to be done to find an appropriate way to do this efficiently.

8.5 Multivariate Approximations

So far we have shown how to automatically generate numerical evaluation routines for bivariate functions via tensor product series in this thesis. We have also summarized the approximation for univariate functions in Thomas A. Robinson's thesis [9]. We have not yet investigated on approximation for functions of more than two variables, for which we must use a different method.

In Orlando A. Carvajal's Masters thesis [1] he attempted to solve the problem of integration in high dimensions by separating the variables in a recursive fashion

using a Deconstruction/Approximation/Reconstruction Technique (DART) until the multidimensional problem is reduced to the evaluation of one dimensional integrals. We may learn to use some of these techniques for multivariate function approximation, yet there is still a lot of research left to be done in this area.

Chapter 9

Conclusion

We have looked at the problem of approximating bivariate functions via tensor product series in this thesis.

We have shown that our toolkit can be used to easily and efficiently generate numerical evaluation routines for bivariate functions. From the experimental results we can see that the toolkit we have developed will be very useful in many Maple applications including evaluating given points as well as plotting, especially when combined with the Maple compiler.

In general, our toolkit can significantly speed up and optimize Maple's bivariate functions (both Maple's built-in functions and user-defined functions) and can also be utilized in many Maple applications to improve the computations.

APPENDICES

Appendix A

Geddes-Newton Series Convergence Theorem

Theorem (Chapman & Geddes, 2008): Assume $A \subset \mathbb{C}$ is compact and let f be a positive definite kernel on A. For all $n \geq 0$, let $r_n = f - s_n$, where s_n is the Geddes-Newton series expansion of f with n distinct diagonal splitting points $\{(a_i, a_i)\}_{i=0}^{n-1} \subset diag(A^2)$. If f is complex-analytic on a sufficiently large region containing A^2, then $s_n \to f$ absolutely and uniformly on A^2 at a linear rate or faster.

Proof Sketch: Let $R_n = f - S_n$, where S_n is the Boolean tensor product which interpolates f on the grid lines $x = a_i$ and $y = a_i$ for $i = 0, \ldots n-1$. For all $n \geq 0$ and some fixed $\gamma \in (0, 1)$,

$$\|r_n\|_\infty = \|\hat{r}_n\|_\infty \leq \left\|\hat{R}_n\right\|_\infty \leq \|R_n\|_\infty = O\left(\gamma^n\right) \text{ as } n \to \infty. \tag{A.1}$$

In addition, the Geddes-Newton series s_n converges absolutely by comparison with a geometric series with common ratio γ.

References

[1] Orlando A. Carvajal. *A Hybrid Symbolic-Numeric Method for Multiple Integration Based on Tensor-Product Series Approximations.* Masters thesis, University of Waterloo, Waterloo, ON, Canada, 2004.

[2] Orlando A. Carvajal, Frederick W. Chapman, and Keith O. Geddes. Hybrid symbolic-numeric integration in multiple dimensions via tensor-product series. *Proceedings of ISSAC*, pages 84–91, 2005.

[3] Frederick W. Chapman. *Generalized Orthogonal Series for Natural Tensor Product Interpolation.* PhD. thesis, University of Waterloo, Waterloo, ON, Canada, 2003.

[4] Jingchi Chen. *Generating Numerical Evaluation Routines for Bivariate Functions via Tensor Product Series.* Masters essay, University of Waterloo, Waterloo, ON, Canada, 2007.

[5] C. T. Fike. *Computer Evaluation of Mathematical Functions.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1968.

[6] K. Geddes. Block structure in the Chebyshev-Padé table. *SIAM J. Numer. Anal.*, 18(5):844-861, 1981.

[7] Keith Geddes, George Labahn, Michael Monagan, and K. M. *Maple 11 Programming Guide.* Maplesoft, a division of Waterloo Maple Inc., Waterloo, ON, Canada, 2007.

[8] Keith O. Geddes. Near-minimax polynomial approximation in an elliptical region. *SIAM J. Numer. Anal.*, 15(6):1225-1233, 1978.

[9] Thomas A. Robinson. *Automated Generation of Numerical Evaluation Routines.* Masters thesis, University of Waterloo, Waterloo, ON, Canada, 2005.

[10] I. Sarkas, D. Mavridis, M. Papamichail, and G. Papadopoulos. Volterra analysis using Chebyshev series. *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium*, pages 84–91, May 2007.

[11] Eric W. Weisstein. *MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/TaylorSeries.html.* 2008.