A Hybrid Symbolic-Numeric Method for Multiple Integration Based on Tensor-Product Series Approximations

by

Orlando Alberto Carvajal

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

Waterloo, Ontario, Canada, 2004

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

This work presents a new hybrid symbolic-numeric method for fast and accurate evaluation of multiple integrals, effective both in high dimensions and with high accuracy. In two dimensions, the thesis presents an adaptive two-phase algorithm for double integration of continuous functions over general regions using Frederick W. Chapman's recently developed Geddes series expansions to approximate the integrand. These results are extended to higher dimensions using a novel Deconstruction/Approximation/Reconstruction Technique (DART), which facilitates the dimensional reduction of families of integrands with special structure over hyperrectangular regions.

The thesis describes a Maple implementation of these new methods and presents empirical results and conclusions from extensive testing. Various alternatives for implementation are discussed, and the new methods are compared with existing numerical and symbolic methods for multiple integration. The thesis concludes that for some frequently encountered families of integrands, DART breaks the curse of dimensionality that afflicts numerical integration.

Acknowledgements

First of all, I wish to thank my two co-supervisors, Professor Keith O. Geddes and Professor Frederick W. Chapman. How could I have done this without them? They were providers of ground-breaking ideas and significant guidance that gave success to this work. I further thank Professor Geddes for accepting me as his student and giving me his support throughout these two years. I do appreciate it. I thank the readers of the thesis: Professor George Labahn and Professor Arne Storjohann.

Second of all, I thank those people from SCG (Symbolic Computation Group) who gave me not only technical assistance but also guidance and encouragement. I did learn a lot from everyone. Special thanks to Wei Zhou, Professor George Labahn, Reinhold Burger, Howard Cheng and Jennifer Keir. The help and support from Wei was invaluable.

Thanks go to David Linder at Maplesoft who, despite being so busy, took the time to make some NAG routines available to me. He always provided me with prompt answers to various technical questions.

I also owe gratitude to Professor Germán Hernández, my thesis supervisor during my Bachelor studies, for his long-lasting advice and guidance.

I wish to acknowledge the encouragement and emotional support of my dear friends: Ricardo and Consuelo Mojica, David Enns, Alejandro Morales and Luis Serrano. They were around during these years and made my life enjoyable. Last but not least, I have always said that I could have never made it this far without my longtime friends from Colombia. I consider myself to be lucky to have the best friends ever. Gracias amigos.

Dedication

Esta tesis va dedicada a mi familia. En especial a mi papá y a Doralba. Desde donde están siento que nos siguen protegiendo y siguen estando muy presente en mi mente.

Gracias papá, mamá por enseñarme buenos valores.

Quien sabe sonreir a cualquier hora, y a pesar de su propio ánimo puede hacer que surja y crezca la alegría oculta que una persona triste pueda tener.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Let us start with the formal specification of the problem of approximating multiple integrals and a discussion of its current importance in several sciences. This chapter also introduces the motivation for investigating a new method to calculate such integrals.

1.1 Approximation of Multiple Integrals

The problem of approximating definite integrals in one dimension is also known as **quadrature**. The term comes from the division of the area under the curve and within the integration interval into rectangles, such that the sum of their areas will be an approximation of the value of the integral. For definite integrals in multiple dimensions, some authors extend the terminology from one dimension, while others choose the term **cubature** instead.

Without loss of generality, we can assume that the integral has constant limits in every dimension. As we will see in Chapter 6, any iterated definite integral with variable limits of integration can be reduced to the following standard form via simple linear changes of variables:

$$\begin{aligned} I(f) &= \int_{[0,1]^d} f(\mathbf{x})\, d\mathbf{x} \\ &= \int_0^1 \cdots \int_0^1 f(x_1, x_2, \ldots, x_d)\, dx_d \cdots dx_2\, dx_1. \end{aligned} \tag{1.1}$$

The region of integration $[0,1]^d$ is known as the **unit hypercube** of dimension d or the **unit d-cube**.

1.2 Applications of Multiple Integration

The number of application areas where multiple integration arises is very large. Trying to list them all would take a lot of space and would certainly miss several areas. However, it is relevant to mention some areas where the method proposed here can be superior to other methods for certain important classes of multiple integrals.

As we will see in the next chapter, there are various methods to evaluate multiple definite integrals. In most of the cases the problems only have a small number of variables and existing methods are usually sufficient. Nonetheless, there are several problems that present a considerable challenge for current algorithms: for example, integrals in high dimensions (say more than 20) and integrands with singularities.

There are several important application areas where the multiple integrals to be calculated involve a large number of variables, sometimes associated to degrees of freedom. In these cases, the current methods do not fulfill the expectations. The list of applications of these problems includes among others: finance, atomic physics, quantum chemistry, statistical mechanics, and Bayesian statistics.

1.3 Motivation

During the last few decades, mathematicians and other scientists interested in numerical integration have been confronted with the large amounts of computer resources that could be needed to evaluate multiple integrals. The increase in the dimension causes an exponential growth in the number of integrand evaluations required to give an answer of specified precision. This growth is known as the **curse of dimensionality** for numerical integration.

In several areas of mathematics it is common to find that some methods are better suited to certain families of problems than others. This is certainly the case in numerical integration: there is no satisfactory solution that works well for all integrands. Various methods have been developed for this problem, but no single method is found to be the best for all cases. Here, we do not intend to develop a new method that will be the panacea for every multidimensional integral, but

rather propose a new method which we believe is superior to other methods for certain important classes of multiple integrals. Furthermore, we aspire to break the curse of dimensionality in multiple integration for some of these problems.

Recently during his graduate studies, Chapman developed the theory of multivariate approximation via natural tensor product series [5, 6, 7], and named such series expansions **Geddes series** in honour of his thesis supervisor. Continuing this work, we use the Geddes series to approximate integrands in multiple dimensions [8]. The motivation for this is that having a new way to approximate functions available, it is natural to try to use this approximation for calculating definite integrals.

There are two main advantages of the approximation by Geddes series. The first one is that the approximation is represented using basis functions with only half as many variables as the original function. This is quite different from other methods that rather choose to do a subdivision of the integration region in order to obtain the required accuracy. The second advantage is that because these basis functions belong to the same family as the original function (e.g., exponential, trigonometric, rational, or polynomial functions), the Geddes series achieves in several cases a faster convergence or gives an exact expansion after just a few terms.

In this way, we obtain a hybrid symbolic-numeric method to approximate an integral by evaluating a finite number of integrals of half the original dimension. Experiments show that when 10 digits of precision are required the number of new integrals is usually less than 15.

This method exploits the possibility of symmetrizing the integrand. In two dimensions, this symmetrization is very simple, but in higher dimensions it could become quite complex. Nonetheless, for several important classes of integrands we are able to use simple changes of variables to obtain the symmetry we require. In most cases, obtaining results with relatively high precision is possible and easy.

Before getting down to business, here is an outline of the rest of this document. Chapter 2 is a survey of the methods that are currently available for computing definite integrals in multiple dimensions. Chapter 3 includes the terminology and definitions that are relevant to natural tensor product series approximations. This provides enough background to continue to Chapter 4 and present our integration method in two dimensions for arbitrary continuous functions over very general regions of integration. The method goes along with some considerations required for a reliable and fast implementation: i.e. inherent properties of the series for symmetric functions allow a fast implementation of the algorithm using simple linear algebra. Chapter 5 presents the experimentation and results relevant to the

two-dimensional problem. In Chapter 6, we introduce DART, which is the extension of the method to integrals in more than two dimensions for certain families of integrands over hyperrectangular regions. Chapter 7 presents the corresponding experimentation and results for DART. Both Chapters 5 and 7 include sections with the details of possible future work for each case. Finally, the last chapter states the general conclusions of this work.

Chapter 2

Traditional Methods of Integration

In one dimension, there are several good methods for numerical integration. Among the most common methods are Clenshaw-Curtis, Newton-Cotes and Gaussian quadratures. Yet, even in one dimension we find that some methods are better suited for certain families of problems than others. For example, some methods are better at handling singularities than others. In general, it is not difficult to find a method that calculates the value of the integral with the required accuracy. In two dimensions, there are also a number of methods available which usually are direct extensions of the methods in one dimension.

There are two main factors that make integrals in multiple dimensions much harder to approximate than in one or two dimensions. The first one is that the geometry of the region over which the integration takes place can be very complex. The second factor is the occurrence of singularities. Handling singularities in one dimension is already hard to solve, let alone in higher dimensions.

This chapter will describe a number of specific methods for the calculation of multiple integrals. The techniques used in these methods are either numerical, symbolic, or a hybrid of the two.

2.1 Quadrature Formulas

The most basic technique to approximate definite integrals uses a quadrature formula that involves the values of the integrand at certain points. Product formulas

extend the quadrature formulas used for one-dimensional integrals to higher dimensions. When the dimension is not very high, these can be used to calculate integrals efficiently. However, the number of sampling points grows exponentially with the dimension. For some specific cases such as polynomial and trigonometric integrands, there are non-product formulas with less than an exponential number of evaluations. Stroud provides a comprehensive list of both product and non-product formulas for a wide variety of regions [27].

2.2 Globally Adaptive Methods

When the estimated error of the quadrature formula is greater than the accuracy required, a more comprehensive approach must be used.

Globally adaptive methods were originally proposed by van Dooren and de Ridder [10]. The most popular implementations are ADAPT [16] and DCUHRE [1, 2], the latter being an evolution of the former. The main idea of these algorithms is to iteratively divide the integration region into sub-regions until the desired accuracy is achieved. At each iteration, quadrature formulas are used to estimate the value of the integral over the subregions. If the estimated accuracy is not good enough, the subregion with highest estimated error is chosen to be divided next.

2.3 Monte Carlo Methods

Monte Carlo methods for numerical multiple integration have been around for many years [17, 22]. The idea behind it is very simple: we choose n uniformly distributed random points in the integration region R; the integral is then estimated by averaging the values at such points and multiplying by the volume of R. If the volume of R is not known, we can choose a simple region S (like a hyperrectangle) which contains the region R and generate random points directly inside of S, discarding those points which lie outside of R. The known volume of S times the fraction of random points lying inside of R estimates the volume of R.

The convergence rate of $O(n^{-1/2})$ for such a method is at the same time its great advantage and sad drawback. Because the convergence is independent of the dimension, the method works better with integrals in higher dimensions than deterministic methods. At the same time, this convergence is very slow, and thus the precision achieved within a reasonable amount of computation time is very low.

Nevertheless, these algorithms are particularly popular in statistical sciences where high precision is not necessary.

2.4 Quasi Monte Carlo Methods

These methods aim to improve the convergence rate of the Monte Carlo method by using quasi-random numbers with specific distributions. The first person to propose this method was Richtmyer in 1952 [24]. Later on, Korobov develop some methods of this type under the name of number-theoretic algorithms [20]. Eventually, a class of quasi Monte Carlo methods known as lattice rules became very popular. A very comprehensive study of lattice rules is given by Sloan [25]. These lattice rules are only a good choice for smooth one-periodic integrands over a hypercube.

Quasi Monte Carlo methods have been receiving some significant attention during the last decade [19, 21, 23]. Several authors investigate a single problem and try to determine the quasi-random sequence of sample points that is better suited for a specific problem or family of problems.

2.5 Dimensionality Reducing Expansions

He [18] presents analytical methods which reduce the dimension of the integral by one. This reduction is achieved by replacing the original region of integration with its boundary. The resulting dimensionality reducing expansions for multiple integration are derived using a multidimensional version of integration by parts, and thus require partial derivatives of the original integrand. This method is mostly used to create boundary type quadrature formulas (quadrature formulas which sample the integrand only on the boundary of the region) and in the evaluation of oscillatory integrals. However, the tensor product methods presented in upcoming chapters have a significant advantage over dimensionality reducing expansions: tensor product methods reduce the dimension geometrically by a factor of two, rather than arithmetically by only one dimension.

2.6 Comparison of Numerical Methods

Most of these numerical methods have been implemented in various numerical software packages such as the NAG library. In a survey of numerical integration methods, Smyth [26] recommends using globally adaptive methods for at most 10 to 15

dimensions. In some cases, quasi Monte Carlo methods start outperforming other methods at dimensions between 10 and 20. For a more specific set of cases, this superiority is maintained to higher dimensions. Beyond those dimensions, the conventional wisdom is that Monte Carlo methods are the only practical numerical methods. In the next section we consider alternatives to numerical methods for integration.

2.7 Symbolic Methods

The use of computer algebra systems such as Maple has permitted further improvements in numerical integration. The main advantages are found when dealing with singularities and unbounded regions in one dimension [11, 12]. These techniques include using a change of variables to make the limits of integration bounded, and approximating the integrand by a series near singularities. Maple uses both of these techniques.

Although the computation of a multiple integral is sometimes possible by iterated one-dimensional symbolic integration, in most cases it is necessary to revert to one-dimensional numerical quadrature rules for at least some of the levels of integration. Applying nested one-dimensional quadrature rules brings us back to the curse of dimensionality. On the other hand, the application of iterated one-dimensional integration can be faster than a purely numerical method when some levels of integration can be performed symbolically.

Chapter 3

Terminology and Notation

The first part of this chapter introduces the concepts required to construct the Geddes series: tensor products and the splitting operator. The second part describes the algorithm to create the Geddes series that are used to approximate our integrands. For a complete and more formal description of the material in this chapter, see Chapman [7].

3.1 Tensor Products

In two variables, a **tensor product** is a finite sum of terms, such that each term is the product of two univariate functions:

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

Example 1 *In elementary mathematics, we can find several bivariate functions and families of functions that can be expressed as tensor products:*

$$\begin{aligned} e^{(x+y)} &= e^x e^y \\ \cos(x+y) &= \cos(x)\cos(y) - \sin(x)\sin(y) \\ (x+y)^2 &= x^2 + 2\,x\,y + y^2. \end{aligned}$$

The minimum number of terms among all the equivalent representations of the tensor product is known as the **rank** of the tensor product. We also say that a tensor

product is **natural** when the factors of each term can be derived from the original function by a finite linear combination of linear functionals. The **Geddes series** is an approximation or representation of a function as a natural tensor product.

3.2 The Splitting Operator

The key tool for the generation of the Geddes series is the splitting operator: given a continuous bivariate function f and a limit point (a, b) of the domain of f, we define the **splitting operator** at the point (a, b) via

$$\Upsilon_{(a,b)} f(x, y) = \lim_{\hat{x} \to a} \left(\lim_{\hat{y} \to b} \left(\frac{f(x, \hat{y}) \cdot f(\hat{x}, y)}{f(\hat{x}, \hat{y})} \right) \right). \tag{3.1}$$

The point (a, b) is called a **splitting point**. When $f(a, b) \neq 0$, we can evaluate the limits in equation (3.1) using the continuity of f to express the splitting operator as

$$\Upsilon_{(a,b)} f(x, y) = \frac{f(x, b) \cdot f(a, y)}{f(a, b)}. \tag{3.2}$$

Two important properties of $\Upsilon_{(a,b)} f(x, y)$ when $f(a, b) \neq 0$ are:

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

The proof of these two properties is trivial from equation (3.2). The combination of these two properties is what allows us to generate a Geddes series approximation. This series is generated simply by iterating the splitting operator while varying the choice of splitting point.

3.3 Geddes Series Expansions

Not every bivariate function f can be expressed as a tensor product of finite rank as in Example 1; however, if the approximation is required only within a compact

rectangle R where f is continuous, we can approximate f by a natural tensor product $s_n(x, y)$ of rank n. We write

$$f(x, y) = s_n(x, y) + r_n(x, y)$$

$$s_n(x, y) = \sum_{i=1}^{n} c_i \, g_i(x) \, h_i(y) = \sum_{i=1}^{n} t_i(x, y) \tag{3.3}$$

where r_n is the **remainder** of the approximation, the t_i are the **terms** of the tensor product s_n, and the coefficients c_i are real constants and can be associated to either g_i or h_i. We call $|r_n|$ the **error function**.

Given a bivariate function f and a maximum allowable absolute error δ, the following algorithm generates a Geddes series expansion of f with maximum absolute error δ. (Assume that the norms $\|r_i\|_\infty$ are calculated only in the region R.)

1. Initialization:
   Let $r_0 = f$ and $s_0 = 0$.

2. For i from 1 to n, and while $\|r_{i-1}\|_\infty > \delta$, choose a splitting point $(a_i, b_i) \in R$ such that $|r_{i-1}(a_i, b_i)| = \|r_{i-1}\|_\infty$ and let

   $t_i = \Upsilon_{(a_i,b_i)} r_{i-1}$

   $s_i = s_{i-1} + t_i$

   $r_i = r_{i-1} - t_i$

3. Return s_n as an approximation of f within R with uniform error $\|r_n\|_\infty \leq \delta$.

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

The uniform convergence of the corresponding infinite series still needs to be formally proven in the general case. Based on much experimental evidence and some formal proofs for specific cases, we believe that for any continuous function f, the remainder r_n converges uniformly to zero when $n \to \infty$, and therefore s_n converges uniformly to f on R. A simple way to see this is to note that *the series s_n interpolates the function f* on the lines $x = a_i$ and $y = b_i$ of the two-dimensional grid generated by the splitting points. The remainder r_n therefore *vanishes on the boundary of each cell* in the resulting $n \times n$ grid. By adaptively choosing well-distributed splitting points we can cover the whole region R with small grid cells, and thus make r_n as close to zero as we want inside these grid cells by uniform

continuity of r_n. Figure 3.1 illustrates the lines, intersection points, and splitting points of a rectangular grid of order n; the splitting points are circled[1].

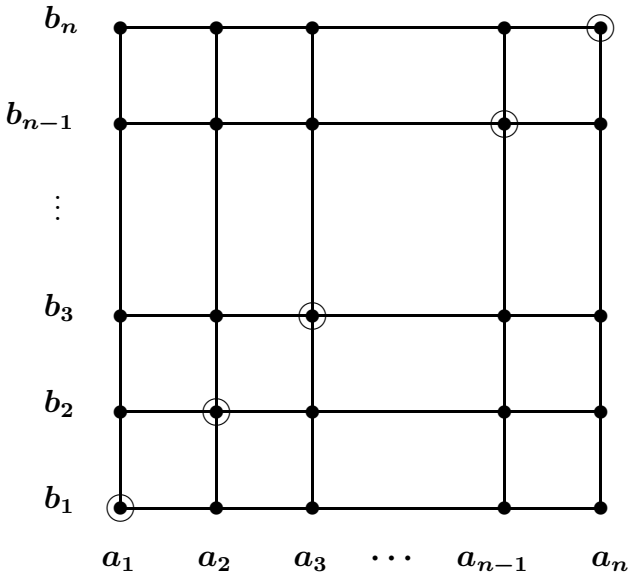

Figure 3.1: The Geometry of a Rectangular Grid of Order n

Notice that in step 2, we have

$$|r_{i-1}(a_i, b_i)| = \|r_{i-1}\|_\infty > \delta > 0,$$

and thus $r_{i-1}(a_i, b_i) \neq 0$. By the above interpolation properties, this implies that $a_i \neq a_j$, $b_i \neq b_j$, when $i \neq j$, and allows the use of (3.2).

The following example illustrates the particular form of the Geddes series expansion.

Example 2 *Consider the following function defined on* $[0,1] \times [0,1]$*:*

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

The Geddes series to three terms is as follows, where the splitting points are of the

[1]Figure courtesy of Frederick W. Chapman

form (a, a) *for* $a = 1,\ 0,\ 0.616336$*:*

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

where

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

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

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

The algorithm for generating Geddes series here presented is what we will use and further develop in the upcoming chapters to approximate the integrands.

Chapter 4

Integration in Two Dimensions

Now that we have a Geddes series to approximate a continuous function f within a compact rectangle $R = [a, b] \times [c, d]$, we can use it to calculate the definite integral

$$I = \int_a^b \int_c^d f(x, y)\, dy\, dx$$

using

$$I_n = \int_a^b \int_c^d s_n(x, y)\, dy\, dx.$$

The idea is that $I \approx I_n$ since $f \approx s_n$ on R.

Let us remember that the main goal we pursue here is the reduction of the dimension of the integrals. We go from calculating one integral in two dimensions to calculating $2n$ integrals in one dimension using equation (3.3):

$$I_n = \int_a^b \int_c^d s_n(x, y)\, dy\, dx = \sum_{i=1}^{n} \left(c_i \int_a^b g_i(x)\, dx \int_c^d h_i(y)\, dy \right).$$

With the concepts from Chapter 3 in mind, we can see how the algorithm will work. The first thing that requires our attention is how to meet the conditions that guarantee the success of the approximation. Let us assume that a double integral has been transformed to be over the unit square $[0, 1]^2$. The following conditions will guarantee that our algorithm will work:

- f is continuous in the closed region $[0, 1]^2$, which implies that the infinity norm of the integrand in $[0, 1]^2$ is finite: $||f||_\infty < \infty$.

- The one-dimensional integrals

$$\int_0^1 f(x,a)\,dx \text{ and } \int_0^1 f(a,y)\,dy$$

can be computed (numerically or symbolically) for any value $a \in [0,1]$.

A simplistic implementation of the algorithm can lead to excessive running times. In this chapter, we will see how a smart implementation of the algorithm dramatically improves the performance. We start by showing how to prepare the integral to meet the requirements of the main algorithm. We continue with a discussion of the evolution of the algorithm, justifying the decisions made in order to get to the final two-phase algorithm, which is fully explained in the remainder of the chapter.

Although some discussion of the performance of the algorithm is mentioned in this chapter, the final discussion is left to Chapter 5 since an important part of it is based on the empirical observations presented there.

4.1 Preparation of the Integral

Before the integral is passed to the approximation algorithm two important conditions must be met: the integrand must be symmetric and the integration limits in both variables must be the constants 0 and 1. Let us begin with the latter condition.

We can use a simple change of variables to convert any iterated integral with non-constant limits to a new one with the desired constant limits. The change of variables

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

will transform the integral as follows:

$$\int_a^b \int_{c(x)}^{d(x)} f(x,y)\,dy\,dx = \int_0^1 \int_0^1 \hat{f}(s,t)\,dt\,ds.$$

Once we have an integral over the unit square $[0,1]^2$, we need to guarantee that the integrand is symmetric. We say that a function is **symmetric** if $f(x,y) \equiv$

$f(y,x)$ and **anti-symmetric** if $f(x,y) \equiv -f(y,x)$. We can express any bivariate function as a sum $f = f_S + f_A$ of a symmetric part f_S and an anti-symmetric part f_A given by

$$f_S(x,y) = \frac{f(x,y) + f(y,x)}{2} \text{ and } f_A(x,y) = \frac{f(x,y) - f(y,x)}{2}.$$

From Fubini's theorem we know that we can directly interchange the order of integration in an iterated integral with constant limits of integration. Applied to our particular case, we have

$$\int_0^1 \int_0^1 \hat{f}(x,y)\,dy\,dx = \int_0^1 \int_0^1 \hat{f}(x,y)\,dx\,dy = \int_0^1 \int_0^1 \hat{f}(y,x)\,dy\,dx.$$

This allows us to conclude that

$$\int_0^1 \int_0^1 \hat{f}(x,y)\,dy\,dx = \int_0^1 \int_0^1 \hat{f}_S(x,y)\,dy\,dx.$$

Example 3 *Consider the double integral*

$$\int_0^1 \int_0^{1-x} e^{x^2y^2} \cos(x+y)\,dy\,dx.$$

Applying the change of variables to transform into an integral over the unit square, we obtain

$$\int_0^1 \int_0^1 (1-s)\, e^{s^2(1-s)^2t^2} \cos(s+t-s\,t)\; dt\,ds;$$

and then applying symmetrization, yields the new problem:

$$\int_0^1 \int_0^1 \frac{1}{2} \cos(s+t-s\,t)\, \left(e^{s^2(1-s)^2t^2}(1-s) + e^{s^2(1-t)^2t^2}(1-t)\right)\, dt\,ds.$$

The Geddes series for the integrand was computed to 3 *terms, where the splitting points were of the form* (a,a) *for* $a = 0,\ 1,\ 0.434450$. *Then the series was integrated (applying one-dimensional quadratures) yielding the following estimate for the original double integral:* 0.385433. *This result agrees with the correct value of the integral to 5 significant digits. By using more than* 3 *terms in the series approximation, more accuracy can be obtained.*

With these two transformations available, we can assume that the double integrals are always calculated over the unit square, with symmetric integrands.

4.2 Evolution of Prototypes

The main characteristic of our integration algorithm is the division of the process into two phases. This strategy came as a result from analysis performed with a series of implemented prototypes. They allowed us to understand the behaviour of the Geddes series for a heterogeneous set of functions.

Early versions of the prototypes were developed by Chapman [6]. He had identified two different approaches to construct the Geddes series expansion of the function distinguished by the strategy used to choose the splitting points. One option is to build the approximation by always choosing the same splitting point, thus requiring the use of equation (3.1). Another option is to build the approximation by choosing distinct splitting points, thus allowing the use of equation (3.2). Both types of expansion with their characteristics are explained in Chapman's PhD thesis [7]. It is our belief that for the purpose of integration the excessive use of limits in the calculation indicates that the first is not a practical choice. So, let us continue explaining the second option which is the one that evolved into what is presented in this work.

The second option is based on the algorithm presented in Section 3.3. The splitting points will always be different because once the point (a, b) is chosen, the remainder vanishes on its vertical and horizontal lines, $x = a$ and $y = b$, preventing subsequent splitting points from being chosen on these lines. The first and obvious advantage is that we can use the simplified form, equation (3.2), to compute the terms of the series. However, a bigger advantage derives from the fact that if the integrand is symmetric, we can find a set of splitting points that generates a series expansion that is also symmetric. As we will soon see, the benefits of this symmetry are more than what one would initially expect.

Although the algorithm for generating the series may seem very simple, some issues arise for its adequate implementation. The number of splitting points required to obtain an approximation of the function at a given precision is what drives the performance of the algorithm. This number depends specifically on the integrand and should make an attempt to minimize the number of splitting points. This is exactly the intent of using the infinity norm for choosing the splitting points.

Finding the splitting point (a_i, b_i) where the error $|r_{i-1}|$ attains its infinity norm, $\|r_{i-1}\|_\infty$, leads to some complications. The estimation of the norm in two dimensions requires an expensive two-dimensional sampling of the remainder r_n. This estimation becomes even more expensive after each iteration because the complexity of the remainder r_n grows cubically in the number of iterations n. What can we do about it?

If the function to approximate is smooth enough, a very practical solution comes from performing a discretization of the function. Having a discrete representation, makes it very simple to calculate and store the values of the remainder. Of course, a discrete representation of the series is not convenient for obtaining a value of the integral for which we can determine its precision. However, experimentation with various families of functions showed that the list of splitting points generated this way is nearly identical to the one that is obtained using a continuous representation, and at a significantly lower computational cost. More about this will be discussed in the last section of the chapter as this became the gist of the first phase in our new algorithm.

4.3 Symmetry Considerations

Chapman had observed that after some small number of iterations, the series starts behaving well, namely convergence became essentially monotonic. Once this point was reached, the maximum error throughout the unit square $[0,1]^2$ was almost always attained on the diagonal line $y = x$. This property is a consequence of the symmetry of the integrand and provided the first important benefit: after a certain number of iterations it is not necessary to sample on the whole unit square to estimate the norm of the remainder. This behaviour was quite consistent for various families of functions and led to the formulation of an important strategy for the algorithm: *try to choose the splitting points on the diagonal as much as possible.*

When the splitting operator is applied to a symmetric remainder r_{i-1} at a point (a_i, a_i), the result is a tensor product term that is also symmetric: $g_i = h_i$ in equation (3.3). Therefore, both the series s_n and the final remainder r_n will be symmetric if the original function f is symmetric and all the splitting points lie on the diagonal.

Unfortunately we cannot always choose splitting points on the diagonal for two main reasons. First, we could have an integrand that is zero on $y = x$, but non-zero in other areas of the unit square, thus, forcing the selection of off-diagonal splitting points. For instance, the function

$$f(x,y) = \frac{x-y}{2-x^2+y^2} + \frac{y-x}{2-y^2+x^2}$$

is zero on the diagonal, but not anywhere else within the unit square. Secondly, it could happen that at some point throughout the process the remainder becomes

very close to zero on the diagonal but it is still quite big off the diagonal. As it was previously mentioned, it is a safe assumption to say that this will not occur once the series starts converging monotonically.

The solution for keeping the symmetry is to *do the splitting at off-diagonal points in pairwise fashion.* Let us remember that $\|r_{i-1}\|_\infty \leq \delta$ is the termination criterion for the algorithm. If at iteration i, $|r_{i-1}(a,a)| \leq \delta$ for all $a \in [0,1]$, but there exists a point (a,b) such that $a \neq b$ and $|r_{i-1}(a,b)| > \delta$, then we do two consecutive splits at points (a,b) and (b,a). Assuming that $r_{i-1}(x,y)$ is symmetric, and $r_{i-1}(a,a) = r_{i-1}(b,b) = 0$, the sum of the two terms generated by the splits, $t_i + t_{i+1}$, is also symmetric:

$$\begin{aligned}
t_i(x,y) &= \frac{r_{i-1}(x,b)\cdot r_{i-1}(a,y)}{r_{i-1}(a,b)} \\
r_i &= r_{i-1} - t_i \\
t_{i+1}(x,y) &= \frac{r_i(x,a)\cdot r_i(b,y)}{r_i(b,a)} \\
&= \frac{\left[r_{i-1}(x,a) - \frac{r_{i-1}(x,b)\cdot r_{i-1}(a,a)}{r_{i-1}(a,b)}\right]\cdot\left[r_{i-1}(b,y) - \frac{r_{i-1}(b,b)\cdot r_{i-1}(a,y)}{r_{i-1}(a,b)}\right]}{\left[r_{i-1}(b,a) - \frac{r_{i-1}(b,b)\cdot r_{i-1}(a,a)}{r_{i-1}(a,b)}\right]} \\
&= \frac{r_{i-1}(x,a)\cdot r_{i-1}(b,y)}{r_{i-1}(b,a)} \text{ (because } r_{i-1}(a,a) = r_{i-1}(b,b) = 0) \\
t_i(x,y) + t_{i+1}(x,y) &= \frac{r_{i-1}(x,b)\cdot r_{i-1}(a,y)}{r_{i-1}(a,b)} + \frac{r_{i-1}(x,a)\cdot r_{i-1}(b,y)}{r_{i-1}(a,b)}.
\end{aligned}$$

The second benefit of symmetry is that although the series can become very large and complex, it can be represented in a very practical way. As we saw in Example 2, the series (3.3) resulting from our algorithm has the following algebraic form:

$$s_n(x,y) = \sum_{i=1}^{n}\left(c_i \cdot \left(\sum_{j=1}^{i} k_{i,j}\, f(x,b_j)\right)\cdot\left(\sum_{j=1}^{i} l_{i,j}\, f(a_j,y)\right)\right) \tag{4.1}$$

where $c_i, k_{i,j}, l_{i,j} \neq 0$ are real-valued coefficients, and (a_i, b_i) is the splitting point used to generate the tensor product term in the i-th iteration. The factors in the terms of the series are linear combinations of the cross-sections $\{f(x,b_i)\}_{i=1}^n$ and $\{f(a_i,y)\}_{i=1}^n$ of the original function $f(x,y)$. Although $a_i = b_i$ does not always hold for specific i, notice that $\{a_i\}_{i=1}^n = \{b_i\}_{i=1}^n$.

Equation (4.1) can be represented using matrices and vectors as

$$s_n(x,y) = \mathbf{V}^T(x) \cdot \mathbf{L}^T \cdot \mathbf{P} \cdot \mathbf{D} \cdot \mathbf{L} \cdot \mathbf{V}(y) \tag{4.2}$$

where

- $\mathbf{V}(x)$ is the column vector of dimension n whose elements are the univariate functions $f(x, a_i)$.
- $\mathbf{D}$ is an $n \times n$ diagonal matrix whose diagonal elements correspond to the coefficients $c_i = 1/r_{i-1}(a_i, b_i)$.
- $\mathbf{P}$ is an $n \times n$ permutation matrix that allows the coefficients $k_{i,j}$ to be obtained from the coefficients $l_{i,j}$ via $[k_{i,j}] = \mathbf{P} \cdot [l_{i,j}]$. The matrix $\mathbf{P}$ is symmetric and block-diagonal. Each on-diagonal splitting point (a, a) generates a diagonal block of the form [1], and each pair of off-diagonal splitting points (a, b) and (b, a) generates a diagonal block of the form $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. If there are no off-diagonal splitting points, then $\mathbf{P}$ is the identity matrix.
- $\mathbf{L} = [l_{i,j}]$ is an $n \times n$ unit lower triangular matrix.

Example 4 *Let us assume that during the tensor product series approximation of a function $f(x, y)$ in $[0, 1]^2$, the splitting points are: $(0, 1)$, $(1, 0)$, $(0.3, 0.3)$, $(0.6, 0.6)$, $(0.1, 0.1)$. After the 5-th iteration, the elements of equation (4.2) will have the following form:*

$$\mathbf{V}(x) = \begin{bmatrix} f(x,0) \\ f(x,1) \\ f(x,0.3) \\ f(x,0.6) \\ f(x,0.1) \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ l_{3,1} & l_{3,2} & 1 & 0 & 0 \\ l_{4,1} & l_{4,2} & l_{4,3} & 1 & 0 \\ l_{5,1} & l_{5,2} & l_{5,3} & l_{5,4} & 1 \end{bmatrix},$$

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} c_1 & 0 & 0 & 0 & 0 \\ 0 & c_2 & 0 & 0 & 0 \\ 0 & 0 & c_3 & 0 & 0 \\ 0 & 0 & 0 & c_4 & 0 \\ 0 & 0 & 0 & 0 & c_5 \end{bmatrix}.$$

This representation reduces the cost of handling what can become extremely complex expressions for r_n and s_n. The direct benefits are:

- The cost of evaluating s_n and r_n is reduced from $O(n^2)$ to $O(n)$ evaluations of the original function f.
- Having only matrix-vector multiplications makes the computation highly efficient, by avoiding unnecessary symbolic multiplication and sum of expressions.
- We only need to perform n one-dimensional integrations of cross-sections of the original function $f(x, a_i)$ for $i = 1, 2, \ldots, n$.

4.4 A Two-Phase Algorithm

As stated in the previous section, the series expansion exhibits a particular behaviour after a few splits. This is the reason why the process is divided into two phases. Let us start listing the differences between these two phases, so we can choose the best approach to use in each phase.

At the beginning of the process:

- The expression that represents the remainder is relatively simple. As a consequence, evaluating the remainder at a given point is relatively inexpensive.
- The qualitative behaviour of the remainder cannot be determined directly. Therefore, determining its norm may require a considerable amount of sampling.
- The remainder may or may not be wavy.

Once the uniform error $||r_{i-1}||_\infty$ starts decreasing monotonically we observe that the remainder exhibits a very particular numerical behaviour. Figures 4.1 and 4.2 show a typical case of what the remainder usually looks like after this point.

By the interpolation theory in Chapman [7], the remainder always vanishes along the lines of the grid generated by the splitting points. The following are additional characteristics of the remainder once monotonic convergence starts:

- The remainder in most cases has constant sign inside each grid cell.
- These signs usually alternate in adjacent grid cells resulting in a checkerboard pattern.

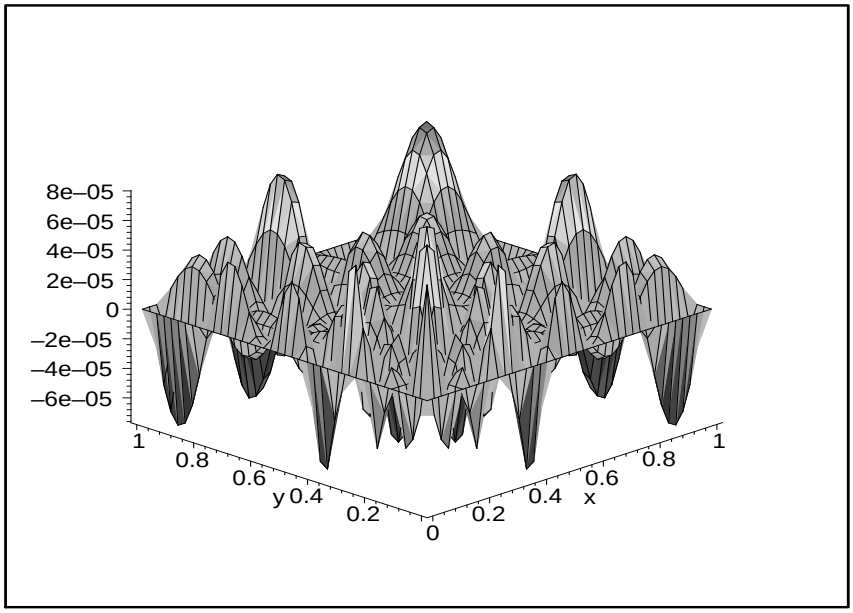

Figure 4.1: Remainder of $\exp\left(\sin\left(\frac{4\pi}{1+x}\right)\sin\left(\frac{4\pi}{1+y}\right)\right)$ after 5 Splits

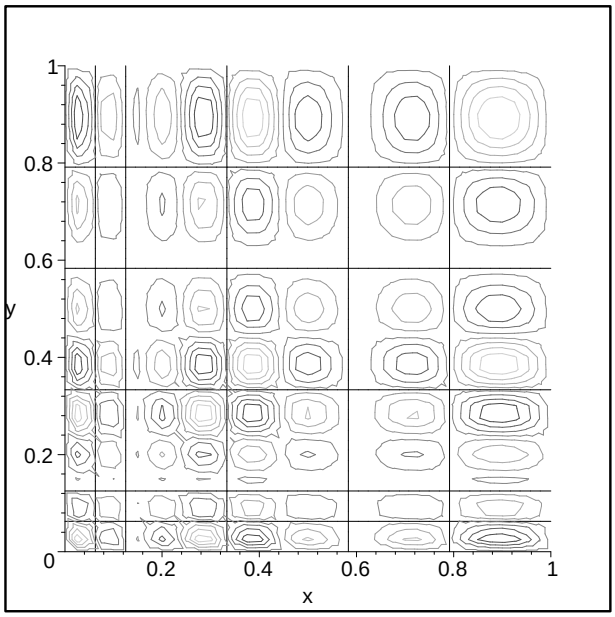

Figure 4.2: Contour Lines for the Remainder of $\exp\left(\sin\left(\frac{4\pi}{1+x}\right)\sin\left(\frac{4\pi}{1+y}\right)\right)$ after 5 Splits

- In general, the maximum error over the entire unit square occurs in one of the grid cells along the diagonal.
- The error on the diagonal, $|r_i(x, x)|$, looks like smooth hills with their crests near the *midpoint* between adjacent splitting points. See Figure 4.3.

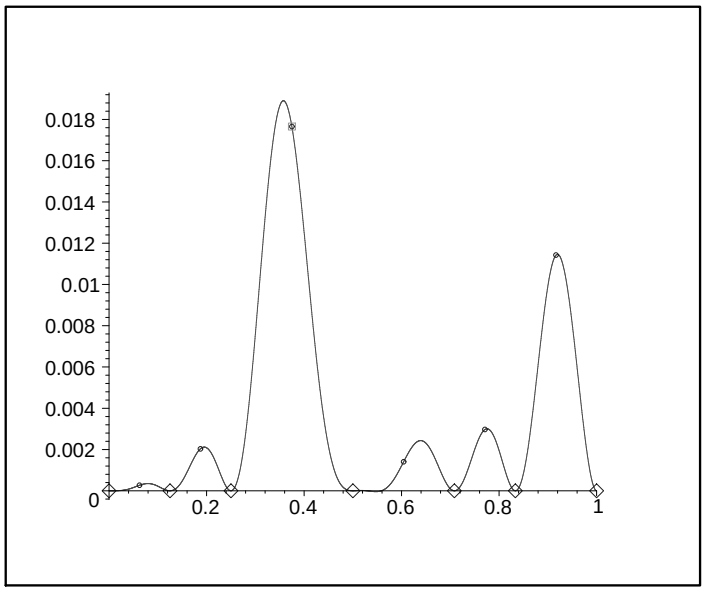

Figure 4.3: Error on the Diagonal for $\cos(2\,\pi\,x\sin(\pi\,y)) + \cos(2\,\pi\,y\sin(\pi\,x))$ after 7 Splits

Based on all these empirical observations and conclusions, we therefore split the process into two phases, which we call the **confinement phase** and the **convergence phase**. Each phase uses a method that we consider gives the best performance without affecting accuracy. Let us point out what happens in each phase.

4.4.1 The Confinement Phase

What we are actually doing in this phase is confining the location (a_i, b_i) of the maximum error $\|r_{i-1}\|_\infty$ to the diagonal $y = x$. After several experiments and a few prototypes, we arrived at the following conclusions:

- We should select splitting points (a_i, b_i) *on the diagonal* $y = x$, unless the function becomes numerically zero on the diagonal. Only in this case do we look at the rest of the unit square to select an *off-diagonal splitting point.*

- In order to preserve the symmetry of the remainder, off-diagonal splitting points must be chosen in symmetric pairs: selecting the point (a, b) as a splitting point implies that the next splitting point must be (b, a).

- The criterion for deciding when the first phase is over will be based on the norm of the integrand. Achieving $1/100th$ of the initial norm has proven to be a good threshold for switching to the convergence phase ($\|r_k\|_\infty \leq \|f\|_\infty /100$). The number of splitting points required to obtain such a threshold depends on the qualitative behaviour of the function f.

- Discrete sampling is the cheapest way to estimate $||f||_\infty$ and $\|r_k\|_\infty$ in the unit square. At the same time, a discrete representation of f in the form of a matrix of sample values at the intersection points of a grid can be iteratively transformed into a discrete representation of the subsequent remainders when the splitting points are selected from the grid.

The way in which the remainder is updated at every iteration deserves some detail. With the sample values of the initial function stored in a matrix, the operations to calculate a new term and update the remainder become simple linear algebra. The simple update operations that take place are known as rank-one and rank-two updates, and they are particular cases of **Schur complements**.

If we have sample values of f on a uniform grid stored in a $k \times k$ symmetric matrix $\mathbf{F} = [f_{i,j}]$, and a splitting point (a, b) whose function evaluation $f(a, b)$ is stored in the element $f_{i,j}$, the new matrix that represents the remainder after splitting is defined as

$$\mathbf{R}_1 = \mathbf{F} - \mathbf{u}_j \cdot \mathbf{v}_i / f_{i,j},$$

where $\mathbf{u}_j$ and $\mathbf{v}_i$ are respectively the j-th column vector and i-th row vector of $\mathbf{F}$. When (a, b) is located on the diagonal, the operation is a particular case of a **rank-one update** of $\mathbf{F}$; and when (a, b) is located off-diagonal, two consecutive operations on elements $[i, j]$ and $[j, i]$ correspond to a particular case of a **rank-two update** of $\mathbf{F}$. The rank-one update can be seen as the discrete version of the splitting operator: vectors $\mathbf{u}_j$ and $\mathbf{v}_i$ correspond to the univariate functions and $f_{ij} = f(a, b)$ to the denominator in equation (3.2). The new matrix R_1 is also symmetric and has zeros in row i and column j.

The discretization of f may potentially introduce some error in our process on two fronts: the number of candidates for splitting points is now finite, and the estimated norms can be smaller than the true norm. Therefore the density of the sampling plays an important role in the appropriate selection of the initial splitting points.

Density is also important in guaranteeing that there will be enough points to split while in the confinement phase: every new splitting point reduces the number of sampling points that are used to estimate $||r_n||_\infty$. On the other hand, the bigger the sampling grid, the more expensive the confinement phase becomes. Consequently, it is important to find some middle ground, that is a grid not too big that allows an acceptable number of splitting points to be chosen while the number of remaining points allows an adequate estimation of $||r_n||_\infty$.

In practice, an initial grid of 25×25 sample points is being used in the author's Maple implementation. If the number of non-zero rows/columns falls below 15 while the $||f||_\infty/100$ threshold has not been reached, the grid is expanded by a factor of 2 in order to allow more splitting points to be chosen. From analysis that takes place in Chapter 5, we postulate that more than one expansion would almost never be necessary for practical problems. If it happens, perhaps this method is not the best choice for the problem, or the problem may be exceptionally difficult.

The discretization of the function according to the parameters indicated above does not adversely affect the general performance of the algorithm. We have found that the number of terms generated without discretization is almost always the same and not necessarily less. Simultaneously, the matrix-vector representation as in equation (4.2) is accordingly updated for use in the next phase. The role of $||f||_\infty$ is to help in the estimation of the progress, and an approximation of it does not meaningfully affect the final result. Finally, the list of splitting points generated should be enough to lead the series towards the monotone convergence of the upcoming phase.

The result of the confinement phase is a list of splitting points (in appropriate order) that make the series start having essentially monotonic convergence. The matrix used in the discretization is not needed anymore and can be discarded.

4.4.2 The Convergence Phase

As it was mentioned before, an important assumption is made: *during the convergence phase, the maximum error will occur on the diagonal, very near the midpoint of adjacent splitting points.* These midpoints constitute the new set of sample points

used to estimate the norm of the remainder during the convergence phase. Again, extensive experimentation has supported the fairness of such an assumption and the norm estimation based on it results in a significant reduction in the running time.

There are two cases in which extra points should be added as sample points. It is very likely that the end points of the diagonal, $(0,0)$ and $(1,1)$, are chosen as splitting points during the confinement phase. In forthcoming discussions we will refer to a **zero exception** when the point $(0,0)$ is *not* chosen as a splitting point in the confinement phase, and a **one exception** when the point $(1,1)$ is *not* chosen as a splitting point in the confinement phase. If either of these points have not been chosen during the confinement phase, then we add them to the set of midpoints as candidates for the next splitting point. In total, at iteration n only $O(n)$ evaluations of the remainder are taken to estimate the norm of the remainder in the convergence phase.

It is during this phase that the benefits of the linear algebra representation of the series play a substantial role in the performance of the algorithm. From equation (4.1) one can see that evaluating the remainder at a single point could involve $O(n)$ integrand evaluations and $O(n^2)$ linear algebra operations; therefore, each iteration would have a computational cost of $O(n^3)$. Fortunately, the sets of sample points are almost equal for two consecutive iterations since at each iteration we remove one of the sample points to use it as the new splitting point, and add at most two new midpoints. When the new splitting point is $(0,0)$ or $(1,1)$ no new sample points are required. Moreover, each iteration inevitably incurs a cost of $O(n^2)$ floating-point operations when computing the new row of matrix $\mathbf{L}$, and this new row can be used to compute the values $t_n(m_i, m_i)$ for all the sample points $\{m_i\}$ that come from iteration $n-1$ in $O(n^2)$ time as well. With these values calculated, previously cached evaluations of the remainder, $r_{n-1}(m_i, m_i)$, are converted to evaluations of the new remainder $r_n(m_i, m_i)$. Evaluating the remainder at the two new sample points still requires $O(n)$ integrand evaluations and $O(n^2)$ floating-point operations.

The process of producing the new entries in matrices $\mathbf{L}$, $\mathbf{D}$, $\mathbf{P}$, and vector $\mathbf{V}(x)$ at iteration n can also benefit greatly from caching. For the new splitting point a_n, the values $f(a_i, a_n)$ for $1 \leq i \leq n-1$, that constitute $\mathbf{V}(a_n)$ in the computation of the new row in $\mathbf{L}$, have already been computed in previous iterations. Since a_n was a sample point until iteration $n-1$, it follows that $r_n(a_n, a_n)$ must have been calculated by means of evaluations of $f(a_i, a_n)$. Therefore, saving the values of $f(a_i, m_j)$ for every combination of splitting points $\{a_i\}$ and sample points $\{m_j\}$ minimizes the number of integrand evaluations required at every iteration.

These caching strategies increase the memory used by about a factor of 3, but they guarantee that no integrand evaluation happens more than once, and reduce the running time at each iteration from $O(n^3)$ to $O(n^2)$. In total, at iteration n the algorithm performs $O(n)$ integrand evaluations and $O(n^2)$ elementary floating-point operations. The space used to store the cached values is similar to the space used to store the matrix representation of s_n: $O(n^2)$.

Lastly, we should discuss the stopping criterion of the convergence phase. From previous chapters we know that the error criterion used to stop the process is of the form $||r_n||_\infty \leq \delta$, which corresponds to the absolute error of the function approximation. Therefore the required accuracy of the integral computation should be somehow expressed in similar terms. Let us see how this is done.

In the case of the integral, the value calculated using the Geddes series expansion has an absolute error that is also bounded by the norm of the remainder in the unit square:

$$\begin{aligned}
|I - I_n| &= \left| \int_0^1 \int_0^1 f(x,y)\,dy\,dx - \int_0^1 \int_0^1 s_n(x,y)\,dy\,dx \right| \\
&= \left| \int_0^1 \int_0^1 f(x,y) - s_n(x,y)\,dy\,dx \right| \\
&= \left| \int_0^1 \int_0^1 r_n(x,y)\,dy\,dx \right| \leq \int_0^1 \int_0^1 |r_n(x,y)|\,dy\,dx \\
&\leq \int_0^1 \int_0^1 ||r_n||_\infty\,dy\,dx = ||r_n||_\infty.
\end{aligned}$$

From the estimate of the norm $||r_n||_\infty$, estimates of the absolute and relative errors can be provided for the integral computation.

One important thing to notice is that the value $|I - I_n| = \left| \int_0^1 \int_0^1 r_n(x,y)\,dy\,dx \right|$ is expected to be much smaller than $||r_n||_\infty$ during the convergence phase. This happens because of the intrinsic form of the remainder: the hills and valleys seem to be evenly distributed above and below zero and thus the remainder's integral is considerably smaller in magnitude than its norm. Two consequences derive from this fact. The good thing is that an estimate based on the remainder's norm compensates for other approximations that take place in the algorithm. On the negative side, if the difference is quite large, there could be some extra terms generated after the required accuracy is achieved thus causing unnecesary work.

It is usually in terms of the relative error that ε, the requested accuracy of the

approximation, is specified. The relative error of I_n should satisfy

$$\frac{|I - I_n|}{|I|} \leq \varepsilon.$$

Assuming that in this phase I_n has at least one digit of accuracy, we can replace the denominator $|I|$ by $|I_n|$ and use the bound $|I - I_n| \leq ||r_n||_\infty$ to obtain the estimated relative error of I_n:

$$\text{Est}\left(\frac{|I - I_n|}{|I|}\right) = \frac{||r_n||_\infty}{|I_n|} \leq \varepsilon. \tag{4.3}$$

When a function has such a form that $|I_n| \ll ||f||_\infty$, e.g. the integral is 0 or very close to 0, the number of digits used in the computation may not be enough to properly handle cancellation problems when adding terms. In such case, one must abandon a pure relative error criterion and switch to an absolute error criterion. However, one can do better than aiming to achieve simply $||r_n||_\infty \leq \varepsilon$. By adding some number G of additional guard digits in the underlying floating-point environment, inequality (4.3) can be replaced by

$$\frac{||r_n||_\infty}{||f||_\infty \times \gamma} \leq \varepsilon, \tag{4.4}$$

where γ is a small constant usually related to the number of guard digits G. For example, $\gamma = 10^{-G}$.

Combining (4.3) and (4.4), we obtain the complete stopping criterion for the algorithm:

$$||r_n||_\infty \leq \varepsilon \ \max(|I_n|, ||f||_\infty \times \gamma). \tag{4.5}$$

Notice that this criterion prevents the algorithm from running forever when the value of the integral is 0. Notice also that if $|I_n|$ is comparable in magnitude with $||f||_\infty$ the stopping criterion can be safely replaced with $||r_n||_\infty \leq \varepsilon \ ||f||_\infty$.

With the full generation of the series completed, we can proceed to integrate the series.

4.4.3 Integration of the Series

Once we have the Geddes series expansion of the integrand, the value of the integral is computed by integrating the n functions in $\mathbf{V}(x)$, and doing the matrix/vector

multiplications in equation (4.2). These n one-dimensional integrals can be computed in two different ways: either numerically or symbolically.

Quadrature algorithms are in general very efficient and quite powerful. This is therefore the first option. As it was mentioned in Chapter 2, Maple takes advantage of the symbolic engine to handle various types of singularities before integrating numerically.

The second option is to take advantage of the symbolic integrator to integrate $f(x, y)$ with respect to one of the variables:

$$f^*(x) = \int_0^1 f(x, y)\, dy.$$

The new function f^* can be evaluated at the n coordinates $\{a_i\}_{i=1}^n = \{b_i\}_{i=1}^n$ of the splitting points to obtain the values of the one-dimensional integrals.

Although symbolic integration sounds very convenient, in practice, the function f^* may be difficult to compute and its algebraic representation could be quite more complicated than f. For instance, consider the functions:

$$f_1(x, y) = \exp\left(\frac{-1}{(x+1)(y+1)}\right)$$
$$f_2(x, y) = \exp\left(\frac{-1}{xy+1}\right).$$

Both functions are analytic in $[0, 1]^2$. However, the integral f_1^* yields a relatively large function that involves the exponential integral function, and the integral f_2^* cannot even be calculated in Maple 9.

In some cases it is difficult to predict which method is faster. Numeric integration is better for most practical cases. Symbolic integration is only worth trying when the number of terms of the series is very large. This is likely to happen for two reasons: the integrand itself is very difficult to approximate, or the precision required in the integration is very high (e.g. software floating-point precision).

This concludes the presentation of the two-dimensional integration algorithm. What lies ahead are the results of experimentation with the algorithm and an extension of the algorithm to higher dimensions.

Chapter 5

Results in Two Dimensions

The algorithm used to tabulate the data in this chapter has evolved through several prototypes. Most of the implementation decisions are empirical and extensive experimentation is used to judge its correctness.

The implementation of the algorithm was done in Maple 9 because of its suitability for performing symbolic mathematics as well as arbitrary precision numerical computation. Most of the linear algebra computation is done directly by NAG/BLAS routines, which are known to be highly optimized. Maple also allows an implementation of the algorithm that performs in software floating-point precision.

The following NAG routines were used in the implementation of the algorithms:

- `f06phf`. Multiplies a real triangular packed matrix by a vector in place.
- `f06eaf`. Computes the dot product of two real vectors.
- `f06pdf`. Multiplies a real symmetric band matrix by a vector in place.
- `f06yaf`. Multiplies two real rectangular matrices[1].

The tests were run on a computer with Intel Pentium 4 processor at 2.02 GHz, with 0.98 GB of RAM, and running Microsoft Windows XP with Service Pack 1. The maximum number of digits that Maple can use to represent numbers in hardware floating-point is given by the command `evalhf(Digits)` and its value is 14.

[1]At the time of implementation of these algorithms the routines `f06pqf` and `f06psf` were not available to Maple. They are a better choice to use instead of `f06yaf` since they directly compute rank updates.

5.1 The Suite of Test Integrands

The experimentation took place using 41 different test integrands that included exponential, trigonometric, logarithmic and polynomial functions. Most of them came from the suite of test integrands previously used by Chapman in his thesis proposal [6]. This is the complete list of functions:

$$
\begin{aligned}
f_{1-7}(x,y) &= \cos(k\,\pi\,x\,y) \text{ for } 1 \le k \le 7 \\
f_{8}(x,y) &= \sin(\pi\,x\,y) \\
f_{9}(x,y) &= \sin(8\,\pi\,x\,(1-x)\;y\;(1-y)) \\
f_{10}(x,y) &= \sin(8\,\pi\,x\,(1-x)\;y\;(1-y)\,(x-y)^2) \\
f_{11-13}(x,y) &= \cos(k\,\pi\,(x-y)^2) \text{ for } 0 \le k \le 2 \\
f_{14}(x,y) &= \exp\left(\sin\left(\frac{4\,\pi}{1+x}\right)\sin\left(\frac{4\,\pi}{1+y}\right)\right) \\
f_{15}(x,y) &= \ln(1+x\,y) \\
f_{16-17}(x,y) &= \cos(k\,\pi\,x\sin(\pi\,y)) + \cos(k\,\pi\,y\sin(\pi\,x)) \text{ for } 1 \le k \le 2 \\
f_{18}(x,y) &= \frac{1-x\,y}{1+x^2+y^2} \\
f_{19-21}(x,y) &= \cos(k\,\pi\,x\,y^2)\cos(k\,\pi\,y\,x^2) \text{ for } 1 \le k \le 3 \\
f_{22}(x,y) &= \frac{x-y}{2-x^2+y^2} + \frac{y-x}{2-y^2+x^2} \\
f_{23}(x,y) &= e^{-y\,x^2} + e^{-x\,y^2} \\
f_{24}(x,y) &= \exp\left(\frac{1-x^2}{1+y^2}\right) + \exp\left(\frac{1-y^2}{1+x^2}\right) \\
f_{25}(x,y) &= 10^{-x\,y} \\
f_{26}(x,y) &= 10^{-10x\,y} \\
f_{27}(x,y) &= \sqrt{\varepsilon}^{(x+y)^6} \text{where } \varepsilon \text{ is the machine epsilon.} \\
f_{28}(x,y) &= \exp\left(\sin\left(\frac{3\,\pi\,y}{1+x}\right)\sin\left(\frac{3\,\pi\,x}{1+y}\right)\right) \\
f_{29}(x,y) &= \exp\left(\sin\left(\frac{5\,\pi\,y}{1+x}\right)\sin\left(\frac{5\,\pi\,x}{1+y}\right)\right) \\
f_{30}(x,y) &= \sin(x+y) \\
f_{31}(x,y) &= e^{x+y}\cos(x+y)
\end{aligned}
$$

$$
\begin{aligned}
f_{32}(x,y) &= \cos(\pi/3 + x + y) \\
f_{33}(x,y) &= \cos(\pi/3 + 5\,x + 5\,y) \\
f_{34-36}(x,y) &= (x+y)^{4\,k} \text{ for } 1 \leq k \leq 3 \\
f_{37}(x,y) &= e^{x^2 y^2} \cos(x+y) \\
f_{38}(x,y) &= \sin(x\,y) \\
f_{39}(x,y) &= \sin(x\,y) + x + y \\
f_{40}(x,y) &= (1+x+y)^{-3} \\
f_{41}(x,y) &= (1+5\,x+5\,y)^{-3}
\end{aligned}
$$

Function f_{27} deserves extra attention as there is a justification for the use of the machine epsilon[2] in its definition. It was specifically designed by Chapman to test the algorithm to its limits and was expected to be extremely difficult to integrate [6]. Its norm is 1, located at $(0,0)$, but it decreases exceptionally fast to the numerical zero on the family of lines $x + y = c$ as c increases from 0 to 2. In fact, the function is also hard to integrate when using other numerical or symbolic methods. Since we can always add guard digits in a floating-point environment of arbitrary precision, the value ε appearing in f_{27} is calculated based on the target accuracy and will differ from the machine epsilon of the floating-point system in which extended-precision calculations take place; otherwise, one would not be able to obtain a correct result with standard numerical methods.

For all the tests here presented, the integral computed is

$$\int_0^1 \int_0^1 f(x,y)\,dy\,dx$$

where $f(x,y)$ is taken from the list above. Notice that seven functions, f_{30-36}, are already tensor products of finite rank.

5.2 Results and Analysis

The experimentation done during the development of this work was quite thorough and adds to the experimentation by Chapman [6]. Throughout the various iterations of the algorithm the data collected was analyzed and used as feedback to

[2]In a floating point computational system, the **machine epsilon** ε is the smallest positive number such that $1 + \varepsilon$ does not round to 1.

make further improvements. This section only presents the results, analysis and conclusions that have more significance to the project. What may look repetitive and does not contribute to a better understanding of the results is avoided for the sake of space. Instead, what reflects the typical behaviour is presented along with references and discussions of the odd cases that were observed.

Before we look at the results, let us make a couple of assumptions that will help in the understanding of the material. First, we shall use the initials **TPA** to refer to the **Tensor Product Algorithm** that approximates the two-dimensional integrals. This term will be used in this and subsequent chapters.

Additionally we will be referring to two types of errors: estimated and actual. When not specified, we shall assume the error is actual. Both terms **error estimate** and **estimated error** can be used interchangeably and refer to the estimation of the relative error in TPA which is an upper bound of the actual relative errors. Likewise, when not specified whether the error is absolute or relative, we shall assume it is relative.

We will also assume that the requested accuracy is always represented as 5×10^{-D} for different values of the number of significant digits D. In theory, the accuracy is not a discrete quantity but here we can assume so for practical reasons. Another reason is that the machine epsilon of the floating-point system is usually expressed in this form.

5.2.1 Integration of Various Functions

Let us begin with the analysis of the results obtained with various types of functions at a fixed accuracy. The results using TPA are compared to the results given by the adaptive routine DCUHRE. Maple has an implementation in C of DCUHRE. Table 5.1 shows the results of these experiments. Only 26 functions of the 41 in the test suite are listed in the table. The functions omitted have a behaviour very similar to at least one function that is included, e.g. functions that are tensor products of finite rank converge very quickly due to the implicit nature of the algorithm. In the table the column Est. RelErr corresponds to the estimated relative error, calculated as in equation (4.3). The column N. Pts indicates the number of splitting points (series terms) generated in the series expansion. The column Actual RelErr shows the relative error calculated with respect to DCUHRE.

The accuracy requested was $\varepsilon = 5 \times 10^{-10}$ for TPA and $\varepsilon = 5 \times 10^{-14}$ for DCUHRE. The difference allows a good calculation of the actual error of the result given by TPA.

Fcn	Time	Result	Est. RelErr	N. Pts	Actual RelErr
f_1	0.360	0.58948987223608	1.7×10^{-11}	7	4.3×10^{-15}
f_2	0.291	0.22570583339509	3.0×10^{-11}	9	9.0×10^{-14}
f_5	0.400	0.10402143283825	6.9×10^{-11}	14	1.1×10^{-13}
f_6	0.471	0.080534202916293	1.3×10^{-11}	16	3.5×10^{-14}
f_8	0.260	0.52466306757532	1.5×10^{-12}	7	1.1×10^{-15}
f_9	0.110	0.57355191766630	1.8×10^{-11}	5	7.9×10^{-13}
f_{10}	0.341	0.069551393138889	8.0×10^{-11}	15	2.6×10^{-13}
f_{12}	0.330	0.48825340607531	1.0×10^{-11}	17	6.3×10^{-14}
f_{13}	0.391	0.35045604941309	9.4×10^{-12}	23	1.7×10^{-14}
f_{14}	0.270	1.1714604745107	8.7×10^{-11}	9	5.3×10^{-14}
f_{15}	0.221	0.20876139454396	1.8×10^{-11}	8	2.1×10^{-13}
f_{17}	0.701	0.24165176672906	3.2×10^{-10}	15	1.0×10^{-13}
f_{18}	0.100	0.50869831592917	7.8×10^{-14}	10	7.9×10^{-15}
f_{19}	0.410	1.4959313336629	9.2×10^{-13}	9	1.9×10^{-14}
f_{20}	0.531	0.97650681215071	2.9×10^{-12}	11	3.0×10^{-14}
f_{22}	0.230	0.094224075145608	4.0×10^{-11}	11	4.1×10^{-14}
f_{23}	0.331	1.7230554135927	5.7×10^{-12}	9	6.2×10^{-14}
f_{24}	0.360	3.4920353042749	5.1×10^{-12}	7	5.2×10^{-14}
f_{26}	0.581	0.16128972668362	6.2×10^{-11}	16	4.5×10^{-14}
f_{27}	1.292	0.20256880553389	1.1×10^{-11}	34	1.9×10^{-14}
f_{28}	3.415	1.1176057514718	3.3×10^{-11}	33	2.9×10^{-14}
f_{29}	9.344	1.0712799262960	1.0×10^{-10}	48	9.7×10^{-15}
f_{31}	0.080	1.0720695615353	1.9×10^{-13}	2	1.6×10^{-14}
f_{34}	0.040	2.0666666666667	8.4×10^{-13}	5	1.6×10^{-14}
f_{36}	0.150	90.010989011673	1.1×10^{-09}	9	7.6×10^{-12}
f_{40}	0.090	0.16666666666659	2.0×10^{-11}	8	4.6×10^{-13}

Table 5.1: Results and Errors of Integrals in Two Dimensions

The current implementation of TPA uses $||r_n||_\infty \leq \frac{\varepsilon}{10} \ ||f||_\infty$ as the stopping criterion for the convergence phase. Two design-related reasons led us to use only $||f||_\infty$ and not consider $|I_n|$ as in (4.5). The first reason is modularity: the procedure that generates the series is separate from the one-dimensional integration in order to allow the former to be used to approximate a function for any other purpose. The second reason will become more evident in upcoming chapters: in higher dimensions, estimating $|I_n|$ during the convergence phase increases the space usage of the algorithm. The factor of $1/10$ that accompanies ε intends to compensate for not considering $|I_n|$ in the stopping criterion. Nonetheless, the estimated relative error is still computed at the end of the convergence phase for comparison.

While working in hardware floating-point precision, the TPA algorithm uses as many digits as the computer itself allows. This is a practical choice because there is no extra computational cost incurred by the algorithm. As a consequence, the farther the required precision is from the hardware floating-point threshold, the more guard digits that are effective. The value calculated by TPA is shown with all the digits used throughout the computation to show that in most cases the actual relative error is much smaller than the estimated error.

The following is a list of observations and conclusions from running these types of tests:

- We can see that all the approximations have actual errors that are over $1,000$ times smaller than the required accuracy $\varepsilon = 5 \times 10^{-10}$. This is an indication that we are generating more terms than necessary.

- There is no actual bound on the number of splitting points required to approximate any function. For example, replacing the constant 3 in function f_{28} by a higher value yields a new function that is harder to approximate, f_{29}. Basically, more oscillations are introduced in the integration region. As a consequence, the number of terms required to reach $\varepsilon = 5 \times 10^{-10}$ when integrating f_{29} increases from 33 to 48.

- In functions f_{36} and f_{41} the ratio between $||f||_\infty$ and I_n starts being significant and this causes the adjustment of ε (to $\varepsilon/10$) to also start falling short by giving an error estimate that does not meet the required accuracy. Despite this, the actual relative error is by far acceptable. When ε is not divided by 10 the results still meet the required precision. However, then about 50% of the functions report an estimated relative error greater than ε, which could be misleading.

- On average, running times get cut to under 30% by the introduction of the matrix-vector representation of the Geddes series as in equation (4.2). This is proof of how important efficient representations and linear algebra computations are in the performance of the algorithm.
- We expect the algorithm to be further improved and an implementation in a compiled language such as C will improve the performance. More on this topic is discussed towards the end of this chapter.

5.2.2 Inside the Series Generation Algorithm

Now that we have a general idea of what the algorithm generally produces, let us take a closer look at what happens during the generation of the series. Let us remember that the criterion to switch from the confinement phase to the convergence phase is estimating that $||r_n||_\infty \leq ||f||_\infty/100$.

Table 5.2 presents the progress of the algorithm for the function $f_4(x, y) = \cos(4\,\pi\,x\,y)$. The column Ph indicates the phase in which the splitting point was selected, 1 for the confinement phase and 2 for the convergence phase. The midpoint norm is defined as $\text{Est}(||r_n||_\infty) = \max_i(|r_n(m_i, m_i)|)$ where m_i is each one of the midpoints on the diagonal. The function's error estimate is given by $\text{Est}(||r_n||_\infty)/||f||_\infty$, the integral's error estimate by $\text{Est}(||r_n||_\infty)/||I_n||_\infty$, and the integral's actual error by $|I - I_n|/|I|$. The last column, midpoint-norm accuracy, corresponds to the ratio $\text{Est}(||r_n||_\infty)/||r_n(x, x)||_\infty$, where $||r_n(x, x)||_\infty$ is calculated via Maple's function `numapprox[infnorm]`. This number shows how well the value at the midpoints represent the norm of the remainder on the diagonal.

Function f_4 is one of the functions with more splitting points generated in the first phase, thus the beginning of the convergence phase was later than average. Because $||f_4||_\infty = 1$ the values for the function's error estimate also correspond to the midpoint estimate of $||r_n||_\infty$.

The numbers in the last column of Table 5.2 indicate that for f_4 the norm estimation via midpoint sampling is notably good. Other functions experience occasional low estimations, yet the average accuracy for the 41 problems is suitably high: 89%. Because the actual relative error is usually much smaller than the estimate (at least one digit more of accuracy), a poor estimate of $||r_n||_\infty$ can still be trusted to estimate the relative error of the integral.

The graphic in Figure 5.1 shows a comparison of the three columns with relative errors from Table 5.2. The values are converted into digits of accuracy via

	Splitting Coord.	Ph	Function's Error Est.	Integral's Error Est.	Integral's Actual Error	Midpoint-Norm Accuracy
1	0	1	2.0×10^{00}	2.0×10^{00}	7.4×10^{00}	-
2	0.500	1	1.9×10^{00}	3.9×10^{00}	3.2×10^{00}	-
3	0.875	1	2.0×10^{00}	5.0×10^{01}	6.7×10^{-01}	-
4	0.292	1	1.0×10^{00}	9.1×10^{00}	5.9×10^{-02}	-
5	0.708	1	2.6×10^{00}	2.1×10^{01}	6.1×10^{-02}	-
6	1	1	1.6×10^{-02}	1.4×10^{-01}	2.5×10^{-03}	-
7	0.958	1	5.8×10^{-03}	4.9×10^{-02}	6.2×10^{-04}	89.9 %
8	0.396	2	1.4×10^{-04}	1.2×10^{-03}	5.0×10^{-06}	99.8 %
9	0.792	2	4.4×10^{-06}	3.7×10^{-05}	1.7×10^{-07}	95.2 %
10	0.146	2	3.1×10^{-07}	2.6×10^{-06}	8.7×10^{-09}	84.3 %
11	0.979	2	1.4×10^{-08}	1.2×10^{-07}	1.1×10^{-09}	97.3 %
12	0.604	2	8.2×10^{-11}	6.9×10^{-10}	3.1×10^{-13}	99.9 %
13	0.917	2	5.2×10^{-13}	4.3×10^{-12}	3.8×10^{-14}	98.7 %

Table 5.2: Progress of the Approximation for f_4

$-\log_{10}(\mathrm{Err}/5)$. The reason for using f_4 as the first example is because this function shows a nice monotonic convergence after some struggle during the confinement phase. Functions f_{1-7} all have a similar algebraic form and their graphics look very similar as well. You could say that the constant k in the definition of these functions determines how long the confinement phase will be: the more oscillatations, the more iterations we need.

Before deriving more conclusions, let us look at the corresponding plots for two other functions, f_{27} and f_{36}, in Figures 5.2 and 5.3. Each of these two functions exhibits a somehow particular behaviour.

It had been mentioned that $f_{27}(x, y) = \sqrt{\varepsilon}^{(x+y)^6}$ is a function whose integral is, by nature, hard to approximate. Nonetheless a generally uniform convergence is noticeable right from the beginning. We can also see that if we had a better estimation criterion for $|I - I_n|$, we could have stopped the process after about 24 iterations, rather than 34. Even though the accuracy of the midpoint estimation is above average for this function, steps back cannot be avoided and they make the convergence rate relatively low.

Function $f_{36}(x, y) = (x + y)^{12}$ is a case where the error estimate of the integral does not meet the required accuracy while the actual relative error is still good. This happens because I is much smaller than $||f||_\infty$. As it was explained in Chapter 4, this becomes a problem when the number of guard digits used in

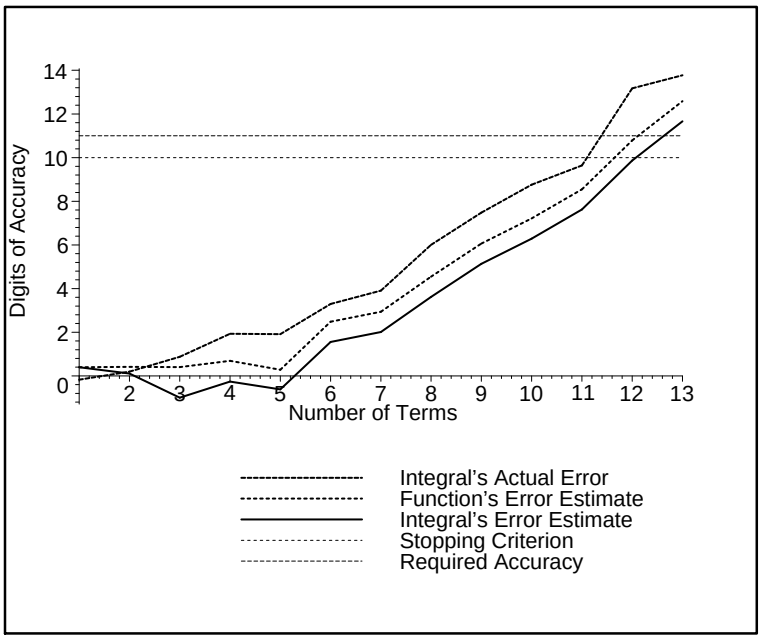

Figure 5.1: Comparison of Relative Error Estimates for f_4

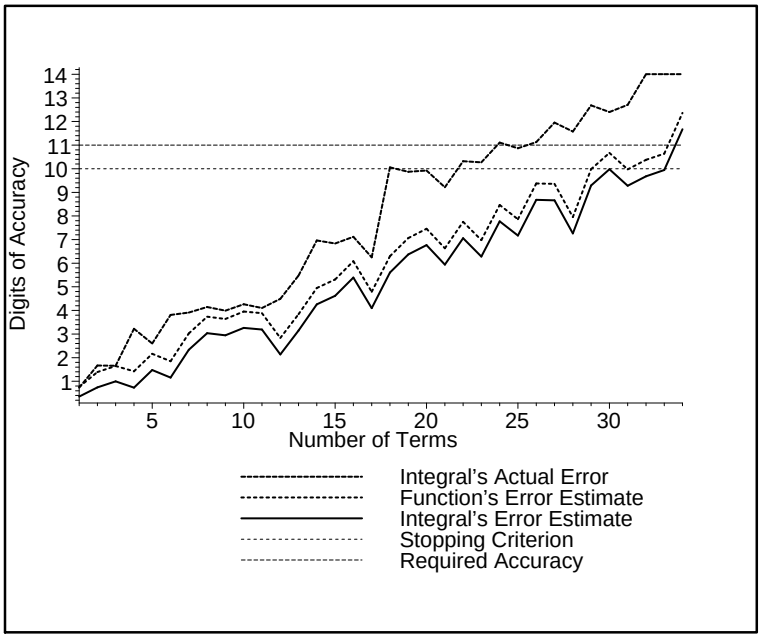

Figure 5.2: Comparison of Relative Error Estimates for f_{27}

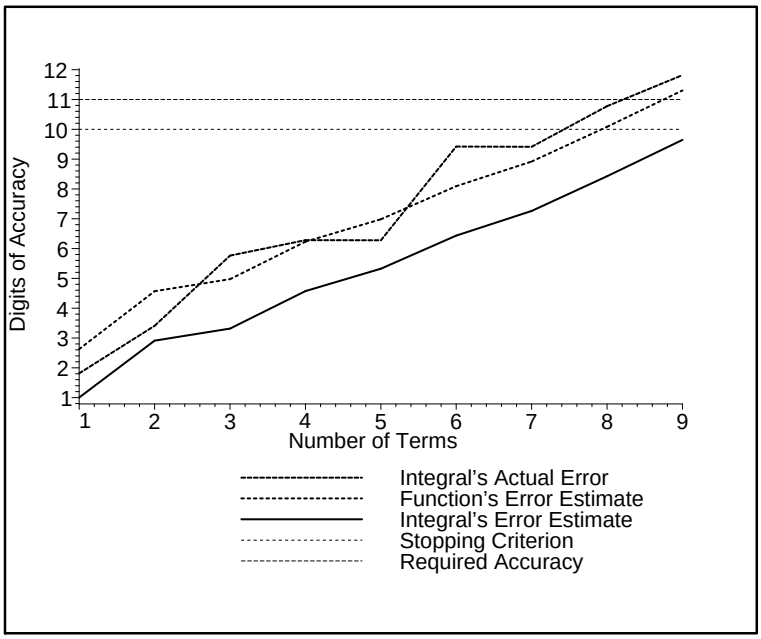

Figure 5.3: Comparison of Relative Error Estimates for f_{36}

the computation is small and does not compensate for such a difference. The consequence is the emergence of the well known cancellation problem that arises in floating-point computation whenever the value of a sum is much smaller than the magnitudes of the terms being added.

The graphics here presented are just a sample of the odd cases. In general, the graphics generated for all the test functions display error estimates that converge in a fairly monotonic way. On the other hand, the progress of the actual relative error is more uneven than the estimates. Based on the plots of the functions in the test suite we can draw the following important conclusions:

- Every function in our test suite has a point after which it starts converging. Usually the convergence starts early in the process.
- The convergence exhibits a linear relationship with respect to the number of terms generated. The only exception is seen in functions that are tensor products. These functions may have a sudden increase in accuracy when the last term is generated, since in that case the remainder is theoretically zero.
- The only functions that do not start converging immediately are oscillatory.

In general, we could say that each function has its own signature. However, they all behave according to what can be seen as a common pattern: no progress during some iterations followed directly by a nearly linear convergence. This is exactly the justification for separating the algorithm into two phases.

Let us finish the discussion of this segment of the tests with some statistics related to how the splitting points are selected:

- The average number of terms for all the 41 approximations is 12 terms.
- About 40% of the points are selected in the confinement phase.
- The nine functions that are tensor products required at most as many terms as their rank. The only one that used fewer terms than its rank was f_{36}: its rank is 13 and the algorithm gave an approximation accurate to 10 digits after only 9 terms. See Figure 5.3. This illustrates that TPA can be used to "economize" a tensor product by lowering its rank.
- There were only two functions that required off-diagonal splitting points: f_{10} and f_{22}. The reason is because the functions vanish on the diagonal, $f(x, x) \equiv 0$. After choosing an initial pair of off-diagonal points the rest of the points can be selected from the diagonal.
- The end points of the diagonal, $(0, 0)$ and $(1, 1)$, are usually chosen during the first phase. Excluding the functions that already vanish on the boundary, thus $f(x, 0) \equiv f(0, y) \equiv 0$ or $f(x, 1) \equiv f(1, y) \equiv 0$, there were nine functions that did not have $(0, 0)$ or $(1, 1)$ chosen as splitting points during the confinement phase. For three of those functions those points were never chosen, but the functions were tensor products of rank two so there was no time or need to choose them. In the other six cases the missing end points were always chosen early during the convergence phase.

5.2.3 Evaluating Integrals with High Precision

Now that we have presented the behaviour of the algorithm from an inside point of view, we will consider the results obtained from running the TPA algorithm at various accuracy tolerances.

To go further than hardware floating-point precision, we require a powerful and effective software floating-point environment. One important advantage that Maple

has is the availability of special linear algebra routines that allow software floating-point computations to be performed using the efficient NAG library routines. Another advantage is that Maple—as most mathematics software—has better tools to compute the approximation of an integral in one dimension than in multiple dimensions. These two facts combined make TPA a very powerful tool to compute two-dimensional definite integrals at software floating-point precisions.

The numerical methods that Maple has available to compute multiple definite integrals (DCUHRE and Monte Carlo) only work in hardware floating-point precision. For software floating-point precision, Maple may combine symbolic and numerical methods to try to give an answer, but the limitations are evident.

Once Maple is working in software floating-point precision, a few extra guard digits do not make a significant difference in the performance of the algorithm. Therefore, we decide to always use 4 guard digits of additional precision for the software floating-point computations.

Let us consider function f_{10} whose exact integral is

$$\int_0^1 \int_0^1 \sin(8\pi\, x(1-x)\, y\,(1-y)(x-y)^2)\, dy\, dx$$
$$= 0.069551393138907990172712979644869005563.$$

This value was symbolically computed using Maple with 34 digits of precision via a Maclaurin series expansion of $\sin(x)$. Table 5.3 presents the results of using the TPA algorithm at various accuracy tolerances.

The first thing we care to know is if the values of the approximations are correct and the series continues converging at higher precisions. The answer for all but one of the 41 functions in the test suite is yes. Again, the problematic function is f_{27}. The one dimensional integrations that took place while approximating the integral of f_{27} with TPA in software floating-point precision gave incorrect results, causing TPA to produce an incorrect result as well. In any other case, TPA provides a result that can be confirmed to be correct and the error estimate of the approximation is also within bounds.

Figures 5.4 and 5.5 illustrate the effect of the accuracy on the number of terms (splitting points) generated and on the time respectively. It was previously indicated that the relationship between the number of terms and the precision is linear during the convergence phase. In Figure 5.4 we can confirm that this assertion still holds in software floating-point precision.

The number of guard digits becomes the main issue when approaching the hardware floating-point threshold. Let us recall that experiments took place in a com-

Req. Acc.	Time	Result	Est. RelErr	N. Pts	Actual RelErr
5×10^{-10}	0.330	0.069551393138889	8.04×10^{-11}	15	2.73×10^{-13}
5×10^{-11}	0.291	0.069551393138909	8.34×10^{-12}	17	1.45×10^{-14}
5×10^{-12}	0.470	0.069551393138908	2.10×10^{-13}	18	1.41×10^{-16}
5×10^{-13}	0.421	0.069551393138908	1.45×10^{-13}	19	1.41×10^{-16}
5×10^{-14}	3.655	0.06955139313890799018	8.01×10^{-15}	19	1.05×10^{-19}
5×10^{-15}	3.725	0.069551393138907990186	8.05×10^{-16}	21	1.91×10^{-19}
5×10^{-16}	4.036	0.069551393138907990172 7	2.61×10^{-17}	23	1.87×10^{-22}
5×10^{-17}	3.866	0.069551393138907990172 73	5.15×10^{-18}	24	2.45×10^{-22}
5×10^{-18}	4.476	0.069551393138907990172 709	1.29×10^{-19}	25	5.72×10^{-23}
5×10^{-19}	4.637	0.069551393138907990172 7097	1.29×10^{-19}	25	4.72×10^{-23}
5×10^{-20}	5.297	0.069551393138907990172 71229	1.13×10^{-20}	26	9.92×10^{-24}
5×10^{-21}	6.359	0.069551393138907990172 712988	3.86×10^{-22}	29	1.20×10^{-25}
5×10^{-22}	6.079	0.069551393138907990172 7129792	7.63×10^{-25}	31	6.40×10^{-27}
5×10^{-23}	6.419	0.069551393138907990172 71297969	6.54×10^{-25}	31	6.49×10^{-28}
5×10^{-24}	7.361	0.069551393138907990172 712979651	6.56×10^{-25}	31	8.82×10^{-29}
5×10^{-25}	7.240	0.069551393138907990172 7129796449	6.66×10^{-26}	33	4.46×10^{-31}
5×10^{-26}	7.031	0.069551393138907990172 71297964491	3.31×10^{-28}	35	5.89×10^{-31}
5×10^{-27}	7.000	0.069551393138907990172 712979644869	3.36×10^{-28}	35	8.00×10^{-35}
5×10^{-28}	8.652	0.069551393138907990172 7129796448700	1.21×10^{-28}	36	1.43×10^{-32}
5×10^{-29}	11.367	0.069551393138907990172 71297964486901	6.28×10^{-32}	38	6.38×10^{-35}
5×10^{-30}	14.220	0.069551393138907990172 712979644869007	6.28×10^{-32}	38	2.07×10^{-35}

Table 5.3: Integral of f_{10} Calculated at Different Precisions

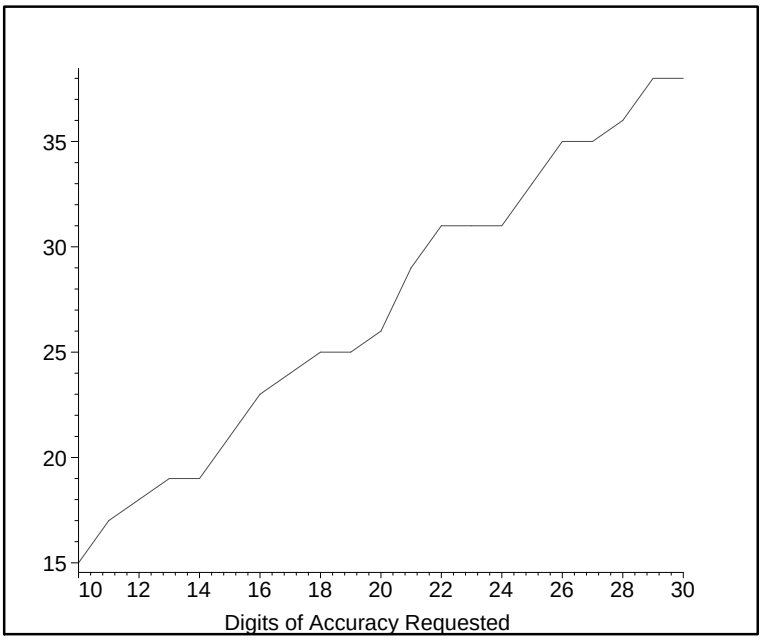

Figure 5.4: Number of Terms vs. Accuracy for f_{10}

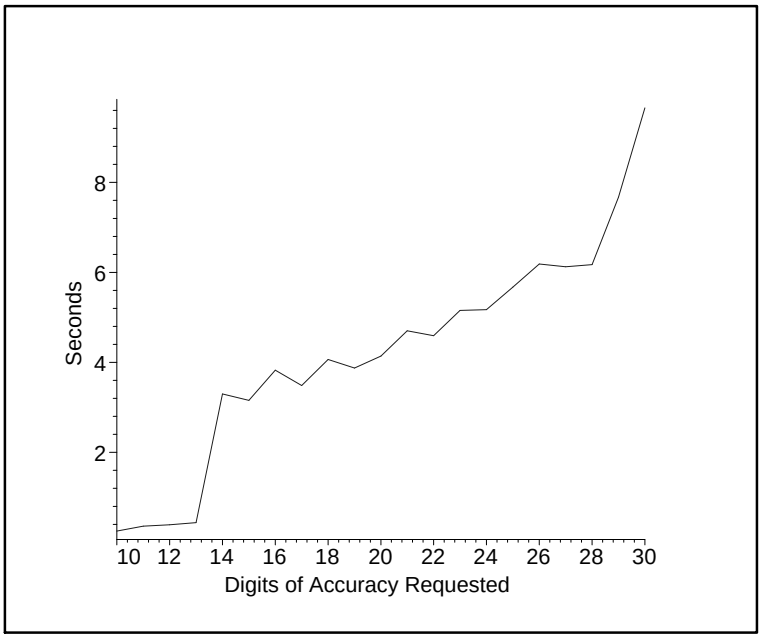

Figure 5.5: Time vs. Accuracy for f_{10}

puter that does hardware floating-point computation with approximately 14 digits. When the accuracy requested was $\varepsilon = 5 \times 10^{-13}$ there was one guard digit available to use without switching into Maple's software floating-point computation. The first adverse consequence is that although the actual error was below the required tolerance ε, the cases where the estimated error was greater than ε became less rare. This means that the estimation of the error was affected by the lack of guard digits.

The second consequence of using few guard digits is numerical cancellation problems. The process stops converging because the limits of the current working precision are reached, and after that, achieving the target absolute error becomes a matter of luck. Functions f_{3-6} exhibit this problem when computed at $\varepsilon = 5 \times 10^{-12}$. When $\varepsilon = 5 \times 10^{-13}$, the number of affected functions increases to 18 out of the 41 functions. The way that TPA handles these cases is by remembering the iteration when the lowest estimated error occurs throughout the convergence phase. If it is detected that the norm has not improved in the last five iterations, the process is stopped and the last five terms are ignored in the approximation. For now, a value of five for this threshold seems to be a fair choice since the longest sequence of no improvement was four and it happened with function f_{29}. No-improvement sequences of three only occurred with three functions: f_{10}, f_{27} and f_{29}.

In the cases where this numerical divergence happened, the approximation still achieved the requested accuracy. Nonetheless, it is recommended to use the algorithm with at least two guard digits in all cases. Requested accuracies of $\varepsilon = 5 \times 10^{-11}$ and larger, or $\varepsilon = 5 \times 10^{-14}$ and smaller did not exhibit any divergence problem with any of the test functions.

In general, the lack of enough guard digits affects the convergence of the algorithm and causes a poorer estimation of the error because of the round-off errors that are introduced. Yet, this is a problem that affects all numerical computations, and not only TPA. It is always dangerous to do computations with only one or no guard digits.

Being able to go to such high accuracy and still obtain correct results is a proof that the criterion of using splitting points on the diagonal as much as possible works remarkably well.

5.3 Performance of the Algorithm

With the experimental data tabulated we can now proceed with the asymptotic analysis of the running time of the algorithm. This analysis is different for each

stage of the algorithm: confinement phase, convergence phase and integration. The reason is because it is based on the generalization of two important conclusions from the previous section:

1. There is an upper bound (named k_1 below) on the number of terms required before monotonic convergence of the series begins. This is the expected number of terms generated in the confinement phase.
2. The convergence rate, which is a ratio between accuracy and the number of terms generated in the convergence phase, seems to be nearly constant.

In general, most of the computation in the algorithm falls into one of the following three categories: integrand evaluations, linear algebra operations on matrices and vectors, and one-dimensional integration. These classifications make the running time of the algorithm depend on several parameters. Let us define those parameters and other relevant variables that will help in the analysis of the computational cost of TPA:

- D is the number of digits of accuracy requested for the approximation.
- m is the size of the sample grid generated in the confinement phase. In our case m is 25.
- $m' \leq m$ is the minimum size of nonzero rows/columns of the grid that allow a good estimation of the norm of the remainder.
- n is the total number of terms required in order to achieve D digits of accuracy.
- For a given function we define the following parameters:
  - c is the cost of one function evaluation.
  - If the function is a tensor product, r is its rank.
  - k_1 is a constant representing the number of splitting points needed to take the Geddes series into convergence.
  - k_2 is a constant representing the convergence rate of the Geddes series when the function is not a tensor product. From the experimentation presented in this chapter, we can assume that such a constant exists and can be defined as

$$k_2 = \frac{n - k_1}{D}. \tag{5.1}$$

– $c_{Int}(D, c)$ is the average cost of the univariate integration of a cross-section of the function, which can significantly vary from one cross-section to another. For a given univariate function, the cost depends on the method used, the required accuracy D, and the integrand evaluation cost c.
For example, if we consider an implementation by Gentleman of Clenshaw-Curtis quadrature [13, 14], which is the default one-dimensional integration method in Maple, this cost is usually $c_{Int}(D, c) \approx k\, c\, 10^{D/3}$ for analytic integrands.

Constants k_1 and k_2 are associated with the maximum error criterion for selecting splitting points and their existence is just an empirical assumption. Based on the experimentation done, there is no evidence of a function that would not fit this generalization. Nevertheless, it is indeed possible that other selection criteria would yield different values for such parameters, but there is no evidence either that they would significantly improve the performance of the algorithm.

Let us recall what operations happen in the confinement phase. It begins with $O(m^2)$ integrand evaluations. Then at each iteration there is a rank one or rank two update that performs $O(m^2)$ elementary floating-point operations[3], and the norm estimation that is usually $O(m)$, or if zero is detected on the diagonal, $O(m^2)$.

There is however a relationship between k_1 and m: $m \geq k_1 + m'$. Let us assume we know k_1 and can therefore deduce the value of m. Then, the cost of the linear algebra operations can be expressed in terms of k_1: $O(k_1\, m^2) = O({k_1}^3)$. After adding the cost of integrand evaluations, we conclude that the cost of the computation during the confinement phase is

$$C_{CNF} = O(c\, {k_1}^2 + {k_1}^3).$$

Because the only structure needed in this phase is a matrix, we know the space required is

$$S_{CNF} = O({k_1}^2).$$

The time and space complexity of each iteration during the convergence phase was mostly analyzed in Chapter 4. The generation of the k-th term requires $O(k)$ integrand evaluations and $O(k^2)$ floating-point operations. Adding the costs for all

[3]Elementary floating-point operations usually refer to the basic arithmetic operations, and they are performed directly by the processor when working in hardware floating point precision.

n iterations, we determine the computational cost of the convergence phase to be:

$$C_{CNV} = O\left(n^3 + c\,n^2\right). \tag{5.2}$$

The space used in the convergence phase is

$$S_{CNV} = O\left(n^2\right).$$

The total cost of the univariate integration is

$$C_{INT} = O(n\,c_{Int}(D,c) + n^2).$$

Finally, the partial costs are combined, and simplified to yield the total computational cost of TPA:

$$\begin{aligned} C_{TPA} &= C_{CNF} + C_{CNV} + C_{INT} \\ &= O(n\,c_{Int}(D,c) + c\,n^2 + n^3). \end{aligned} \tag{5.3}$$

where $n = D\,k_2 + k_1$. If the integrand is a tensor product, $k_1 = 0$, n can be replaced by r, and (5.3) is simplified to

$$C_{TPA} = O(r\,c_{Int}(D,c) + c\,r^2 + r^3).$$

The constants k_1 and k_2 are usually small. For example, function f_{13} has the largest value of k_1, about 9; and the values of k_2 for the test functions that are not tensor products range from about 0.5 to 5. On the other hand, the parameters c and $c_{Int}(D,c)$ can be relatively large and have a more significant impact on the computational cost of the algorithm. After substituting n from equation (5.1) in (5.3), constants k_1 and k_2 are discarded and we obtain an expression in terms of the requested accuracy:

$$C_{TPA} = O(D\,c_{Int}(D,c) + c\,D^2 + D^3).$$

Some observations can be made based on the implementation done with this work which we consider to be quite efficient. After tracing the times for the three different types of operations that take place, we can see that nearly all the processing time is spent in evaluations of the integrand, which is what we would hope for. The computation that comes second in time consumption is the linear algebra operations, but its time is very small compared to integrand evaluations.

5.4 Future Modifications to the Algorithm

The aim of this final section is to give additional information to those who want to implement a new version of TPA. It contains some things that we tried and did not find helpful, and other things that, although promising, did not get to be implemented. Basically, better performance could be obtained by decreasing the number of terms needed to obtain a given accuracy, cutting the costs of the operations involved, or finding a faster way to estimate the norm of the remainder.

5.4.1 What We Tried with No Positive Results

The following modifications to the algorithm took place at some point throughout our experimentation. Unfortunately, they do not bring clear improvements to the performance, and rather confirm the postulates of the two phases identified in the Geddes series expansion.

- Off-diagonal points in the convergence phase. During the convergence phase, the norm of the remainder is occasionally smaller on the diagonal than over the unit square. Allowing the algorithm to sample the whole unit square and select off-diagonal splitting points during this phase rarely produced series with one term less and never with two terms less.
- Numerical approximation of infinity norm. The hills and valleys of the remainder did not always have the nice shape illustrated by the figures in Chapter 4. Negative and positive hills within the same cell and asymmetric hills are the causes of a poor norm estimation when using midpoints. This is the reason for comparing the performance between the midpoint approximation and the true norm via `numapprox[infnorm]`. As it was explained before, the number of terms did not consistently decrease whereas the excess in the cost was evident. The rare cases where using the numerical norm generated one term less happened because the stopping criteria had been missed by very little. Besides, this extra term provided more accuracy to the result.
- Tuning the parameters in the confinement phase. These parameters include the threshold for switching to the convergence phase, and the initial size and minimum size of the sample grid as described in Chapter 4.
  - Increasing the initial size of the sample grid to 40 also increases significantly the times (about 65%) and yields only occasional small differences in the number of terms.

- Although decreasing the initial size of the sample grid improves the performance of the algorithm (about 10%), the quality of the estimation of $||f||_\infty$ may not be as good as before.
- Decreasing the threshold forces the selection of more splitting points in the confinement phase, which implies either having a bigger initial grid or expanding it. Either choice increases the running time of the algorithm and the difference in the total number of terms—if any—is small.
- Increasing the threshold allows switching to the convergence phase sooner. However, it also brings about a remainder that does not exhibit the smoothly oscillating behaviour we require to properly use the midpoint estimation. On top of that, the improvement in performance is so subtle that it is not worth the risk.

5.4.2 What We Did Not Try and Could Bring Positive Results

There are still a significant number of things to be tried and experimented that could bring improvements to the algorithm here presented. In some cases we can predict that the results will be positive, whereas in other cases such an affirmation is uncertain.

The following are some possible improvements to the current TPA:

- Use better estimation of the errors. Being able to compute the estimated value of the remainder's integral may allow time savings in the generation of the series. We are using a bound on the absolute error as estimation, but as we have seen in this chapter, we are usually generating more terms than necessary for the approximation.
- Modify the criterion to detect cancellation problems. Add a condition to check that the divergence is detected when the norm is indeed close to the limits of floating-point zero. Then, the algorithm could automatically increase the number of guard digits in the computation, even if it has to pass from a hardware to a software floating-point precision environment.
- Improve the stopping criterion. Currently and due to reasons previously mentioned, the stopping criterion of the convergence phase is based on the function's norm. However, when it is known that the series will be integrated, this criterion should be reformulated to reflect (4.5). Both cases can be handled at the same time by allowing the specification of different stopping criteria.

- The sample points of the confinement phase can be quasi-randomly generated. That is, instead of having an evenly distributed grid, the points of the grid can be generated with a nearly even distribution of distances. When the sample points are known beforehand, a function can be constructed to make the algorithm believe the integrand is zero in the whole unit square, and therefore an incorrect result would be produced.

- The one-dimensional integration could be smarter in finding whether symbolic integration is faster than numerical, or the other way around. For example, let us assume the algorithm calculates the first integral numerically and finds that it takes such a significant amount of time that symbolic integration is worth trying. Then, we can determine a maximum amount of time that the symbolic integration can take if it is to calculate the values of the remaining integrals.
  Unfortunately, cases where this strategy would work could be rare, i.e. for integrals of the form $\cos(k\,\pi\,x\,y)$ when k is large, say $k > 20$, using symbolic integration is slightly faster than using numerical one-dimensional integration.

- Try to determine conditions under which the confinement phase is expected to be longer. So far, all we can say is that it occurs with *some* oscillatory integrands.

- Finally, it is yet to be seen how the algorithm performs when implemented in a compiled language such as C. The times for the algorithm when run in Maple are very reasonable, but they could be improved.

Once the performance of TPA is taken care of, we can proceed to find ways to enhance it to handle other types of problems. In the next two chapters, we present an algorithm that uses TPA to compute integrals in higher dimensions. This approach is by no means the only way to extend TPA to integrals in higher dimensions and other ways can be investigated. For example, another path to take is to explore the possibilities of handling singularities with Geddes series expansions.

Chapter 6

Integration in Higher Dimensions

In higher dimensions, a method based on dimensional reduction of a family of integrals is developed. In this chapter, we present the results of exploring the use of the tensor product integration in more than two dimensions. The reduction of the number of variables in the functions to be integrated is what motivates us to use this approximation technique in higher dimensions, with the goal of breaking of the curse of dimensionality in numerical integration.

Two different approaches are discussed. The first path we will follow is to extend Geddes series expansion to functions that exhibit some sort of symmetries. The discussion is limited as there are many issues to deal with, and during the development of the work presented in this dissertation an alternate, quite powerful technique was found. This second technique becomes the main focus of this chapter, and results of its implementation are summarized in next chapter.

For integrals in more than two dimensions, we still want to create an approximation of the integrand by a tensor product series such that the number of variables in the new functions is cut in half. The main concern is how to guarantee the symmetry that is so crucial to the method. As we saw in the previous two chapters, it is this symmetry of the integrand which makes the computation much faster.

We will be discussing a novel approach which allows us to exploit our *two-dimensional* approximation techniques in *high-dimensional* integration problems.

6.1 Pursuing Symmetry in Any Dimension

In Chapter 4, the preparation of the integral for integration in two dimensions involved two steps: applying a linear change of variables to obtain a new equivalent

integral over the unit square, and replacing the integrand with its symmetric part. The corresponding changes in higher dimensions are similar and are presented in this section.

We can convert an iterated integral with *variable* limits of integration

$$\int_{a_1}^{b_1} \int_{a_2(x_1)}^{b_2(x_1)} \cdots \int_{a_d(x_1,\ldots,x_{d-1})}^{b_d(x_1,\ldots,x_{d-1})} f(x_1, x_2, \ldots, x_d)\, dx_d \cdots dx_2\, dx_1$$

to an integral with *constant* limits of integration

$$\int_0^1 \int_0^1 \cdots \int_0^1 \hat{f}(s_1, s_2, \ldots, s_d)\, ds_d \cdots ds_2\, ds_1$$

by iteratively applying the one-dimensional linear change of variable

$$x_k = a_k(x_1, \ldots, x_{k-1})\,(1 - s_k) + b_k(x_1, \ldots, x_{k-1})\, s_k$$

from $k = d$ down to $k = 1$.[1] Since, the new region of integration $[0, 1]^2$ is symmetric, we can proceed to attempt the symmetrization of the new integrand $\hat{f}$.

The conversion of a general multivariate integrand to its symmetric form can be very simple in theory but it generates much larger expressions. We can obtain the symmetrization of a function $f(x_1, \ldots, x_d)$ by averaging the expressions generated from all the possible permutations of the variables. The problem is that in d dimensions, there are $d!$ possible permutations. The fully symmetrized form is

$$f_S(x_1, \ldots, x_d) = \frac{1}{d!} \sum_{\sigma \in S_d} f\left(x_{\sigma(1)}, \ldots, x_{\sigma(d)}\right),$$

where S_d represents the set of all $d!$ permutations of $1, \ldots, d$. The number of terms in $f_S(x_1, \ldots, x_d)$ will be very large for even modest values of d.

Having constant limits of integration in integrals with many variables is not rare, whereas obtaining a symmetric integrand in high dimensions is most unlikely. This is why it is wise to pursue alternate ways of symmetrization. In the next section we present a novel way of obtaining the symmetry we need for any dimension in certain families of functions.

[1] The order in which these transformations take place is important. Applying the transformations in increasing order of k will instead involve quadratic cost compared to the linear cost of the decreasing order.

6.2 General Description of DART

While trying to find fast and efficient ways to evaluate multiple integrals of functions with special structure in higher dimensions, Frederick W. Chapman came up with an idea that eventually evolved into the very powerful method that is presented here. We call this new method of multivariate approximation and multiple integration the **Deconstruction/Approximation/Reconstruction Technique (DART)**.

This method exploits the fact that the high-dimensional integrals arising in actual applications frequently fit certain patterns. Multivariate functions constructed from a sum or a product of univariate functions can be very common. We will see that we do not even need to have the original function be symmetric.

In few words, the goal is to find a change of variables that converts an integrand $f(x_1, x_2, \ldots, x_d)$ in $d > 2$ variables into a symmetric bivariate function $v(s,t)$, generate the corresponding Geddes series approximation $s_n(s,t)$, transform this to a series $S_n(x_1, x_2, \ldots, x_d)$ in d variables, and then integrate. First, we use an example to illustrate the steps of the method, and then we will formulate the more general method.

Example 5 *(Ridge Function) Let us assume that a function $f(x_1, x_2, \ldots, x_d)$ in $d = 2\,k$ dimensions, can be rewritten as*

$$f(x_1, x_2, \ldots, x_d) = u(a_1\,x_1 + a_2\,x_2 + \cdots + a_d\,x_d),$$

with $a_i > 0$. We need to calculate the integral of f over the unit d-cube $[0,1]^d$; thus, $x_i \in [0,1]$.

We start with a deconstruction *of the original function into a symmetric bivariate function by making a change of variables. We can split the sum into two groups, each containing k variables, and let*

$$\begin{aligned} s &= \frac{a_1\,x_1 + a_2\,x_2 + \cdots + a_k\,x_k}{c} \\ t &= \frac{a_{k+1}\,x_{k+1} + a_{k+2}\,x_{k+2} + \cdots + a_{2k}\,x_{2k}}{c}, \end{aligned} \tag{6.1}$$

where

$$c_1 = \sum_{i=1}^{k} a_i, \qquad c_2 = \sum_{i=k+1}^{2k} a_i, \qquad c = \max\left(c_1, c_2\right),$$

to obtain

$$\begin{aligned} f(x_1, x_2, \ldots, x_d) &= u(a_1\, x_1 + a_2\, x_2 + \cdots + a_{2k}\, x_{2k}) \\ &= u(c \cdot (s + t)) \\ &= v(s, t). \end{aligned}$$

Notice that the resulting function $v(s,t)$ *is* symmetric.

Since all x_i *are in* $[0,1]$*, in the best case we would have* $c_1 = c_2$ *and therefore both* s *and* t *will also be in* $[0,1]$*. Otherwise,* $c_1 \neq c_2$ *and only one of them will be in* $[0,1]$*, while the other will be in* $\left[0, \frac{\min(c_1,c_2)}{c}\right] \subset [0,1]$*. We now compute an* approximation *of the symmetric bivariate function* $v(s,t)$ *in* $[0,1]^2$ *with our Geddes series approximation algorithm.*

Once we have a Geddes series expansion of $v(s,t)$ *in* $[0,1]^2$*, we can substitute* s *and* t *using equations (6.1) to obtain a multivariate series expansion in the original variables* x_i *(*reconstruction*). The series approximation will be valid over the unit* d*-cube* $[0,1]^d$*. Finally, we just need to do the same process recursively for both factors in each term until we have only one variable in each factor. Then, we can use a standard quadrature method to evaluate the resulting one-dimensional integrals.*

This process looks indeed very simple, and the idea can be generalized to a larger family of problems that fit certain *function patterns*. The following are other types of functions where the same approach can be used:

1. The case where $a_i = 1$ for all i and d is even is already included in the example given. In this case the function f is symmetric and $c_1 = c_2 = c = k$.

2. If d is odd, we still can split the sum into two groups of $\lfloor d/2 \rfloor$ and $\lceil d/2 \rceil$ terms.

3. A slightly more complex case happens when $\{a_i\}$ can be negative. The solution in such case is to add offset values, $o_1, o_2 \geq 0$, to the numerators of the fractions that will make the numerators always positive

$$\begin{aligned} s &= \frac{o_1 + \sum_{i=1}^{k} a_i\, x_i}{c} \\ t &= \frac{o_2 + \sum_{i=k+1}^{2k} a_i\, x_i}{c}, \end{aligned}$$

and adjust c_1 and c_2 accordingly:

$$c_1 = \sum_{i=1}^{k} |a_i|, \qquad c_2 = \sum_{i=k+1}^{2k} |a_i|.$$

4. If in case 1, instead of a sum of variables we have a product of variables, $f(x_1, x_2, \ldots, x_d) = u(x_1 x_2 \cdots x_d)$, some simplification is possible: $c = 1$.

5. If in cases 1 and 4, instead of the single variables we have squares of variables, or any other power, the method would not change at all.

6. Finally, if instead of a power we have any univariate function whose range is known, the method can still be applied, and that is what we will see next.

Now that we know that there are more cases where DART can be used, we provide a general and complete description of the method. We start with the same function

$$f(x_1, x_2, \ldots, x_d)$$

of d variables to be approximated (integrated) in the hyperrectangle $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$; thus $x_i \in [a_i, b_i]$.

6.2.1 Pattern Recognition and Symmetrization

The first thing we do is detect a pattern in the integrand that allows the definition of a second analogue function, with the same number of variables, but symmetric. We should keep in mind that although this new function is symmetric, its domain may not necessarily be.

Let us assume that f can be expressed in terms of a univariate function u as

$$f(x_1, x_2, \ldots, x_d) = u(g_1(x_1) \,\&\, g_2(x_2) \,\&\, \cdots \,\&\, g_d(x_d)), \tag{6.2}$$

where & denotes either $+$ or $\times$.

Let us call each g_i a **template function** and define:

$$G = \{g_i\}_{i=1}^{d}$$

which is the set of all different univariate template functions identified in f. The size of G is clearly bounded by $1 \leq |G| \leq \dot{d}$. Let us further assume that for each template function $g \in G$, we can algorithmically find its range:

$$R_g(a,b) = \left[\min_{x \in [a,b]} (g(x)), \max_{x \in [a,b]} (g(x)) \right].$$

This allows us to compute the d ranges of $g_i(x_i)$.

The variables of the new symmetric function are defined as

$$y_i = g_i(x_i).$$

Thus, we can write

$$\begin{aligned} f(x_1, x_2, \ldots, x_d) &= u(y_1 \,\&\, y_2 \,\&\, \cdots \,\&\, y_d) \\ &= \hat{u}(y_1, y_2, \cdots, y_d). \end{aligned}$$

For each y_i, $1 \leq i \leq d$, we calculate its corresponding range

$$[a_{y_i}, b_{y_i}] = R_{g_i}(a_i, b_i)$$

The function $\hat{u}$ is symmetric and can be transformed into a bivariate symmetric function to be approximated by the Geddes series expansion. However, it is more practical to use $u(y_1 \,\&\, y_2 \,\&\, \cdots \,\&\, y_d)$ throughout the recursion. The following are the three steps that constitute the recursion and give the name to the method.

6.2.2 First Step: Deconstruction

The goal is to approximate $u(y_1 \,\&\, y_2 \,\&\, \cdots \,\&\, y_d)$, with $y_i \in [a_{y_i}, b_{y_i}]$, using a Geddes series expansion such that the variables y_i are separated into two groups. In this first step, we transform u into a function v:

$$u(y_1 \,\&\, y_2 \,\&\, \cdots \,\&\, y_d) = v(s,t), \tag{6.3}$$

where v is a bivariate symmetric function and $s, t \in [0,1]$.

We start by partitioning the d operands y_i from equation (6.3) into two groups such that

$$w_1 \,\&\, w_2 = y_1 \,\&\, y_2 \,\&\, \cdots \,\&\, y_d,$$

and the number of operands in w_1 and w_2 differs at most by one. Since we know the ranges of y_i we can use interval arithmetic to calculate the ranges of w_1 and w_2, which we represent as $[a_{w_1}, b_{w_1}]$ and $[a_{w_2}, b_{w_2}]$ respectively.

At this point we have a function that is symmetric, $u(w_1 \,\&\, w_2)$, but unfortunately the ranges of w_1 and w_2 are not necessarily the same. However, we can define a transformation of variables that maps the rectangle $[a_{w_1}, b_{w_1}] \times [a_{w_2}, b_{w_2}]$ to a region that is contained in $[0,1]^2$.

Consider the symmetric square defined by $[\min(a_{w_1}, a_{w_2}), \max(b_{w_1}, b_{w_2})]^2$, which contains $[a_{w_1}, b_{w_1}] \times [a_{w_2}, b_{w_2}]$. The following *symmetric* change of variables maps this square into $[0,1]^2$:

$$\begin{aligned} s &= \frac{w_1 - o}{c}. \\ t &= \frac{w_2 - o}{c}; \end{aligned} \tag{6.4}$$

where

$$\begin{aligned} o &= \min(a_{w_1}, a_{w_2}) \\ c &= \max(b_{w_1}, b_{w_2}) - o. \end{aligned}$$

The constant o shifts the bottom left corner of the square to $(0,0)$ and the constant c scales it to occupy the whole unit square. This transformation yields the symmetric function

$$u(w_1 \,\&\, w_2) = u((c\,s + o) \,\&\, (c\,t + o)) = v(s,t) \tag{6.5}$$

which is what we pass to the approximation algorithm.

Notice that this transformation is not as efficient as the one given in Example 5. This brings up the hidden cost of DART: *we could be approximating u in a larger region than is necessary.* We call this hidden cost **domain inflation**. It means that there is a waste of approximation space in a region outside of the integration limits. In the worst case, the function u may not even be defined outside of $[a_{w_1}, b_{w_1}] \,\&\, [a_{w_2}, b_{w_2}]$. This is why we should try to create a partition and a transformation that minimize the amount of unused approximation space. In the next section we will show that when & is the + operator, this optimization is possible.

6.2.3 Second Step: Approximation

The difficult part of the process is over. We can now compute the approximation $s_n(s,t)$ of $v(s,t)$ in $[0,1]^2$ using our Geddes series expansion algorithm for two dimensions.

6.2.4 Third Step: Reconstruction

Within the Geddes series expansion $s_n(s,t)$ of $v(s,t)$ on $[0,1]^2$, we substitute s and t back to obtain a series expansion in the variables y_i. If the series has n terms, we will have either n (if $w_1 \equiv w_2$) or $2\,n$ different univariate functions that will become the new u functions for the next level of the recursion. These new functions have either $\lfloor d/2 \rfloor$ or $\lceil d/2 \rceil$ variables.

The recursive steps continue until we obtain functions with only one variable y_i. Then, we can easily compute the values of the integrals over $[a_i, b_i]$ after substituting $y_i = g_i(x_i)$.

Notice that the role that each y_i plays is to avoid the substitutions of g_i and the calculation of $R_{g_i}(a_i, b_i)$ at every level of the recursion.

6.2.5 Integration

The values of the one-dimensional integrals can be computed either numerically or symbolically. The considerations discussed in Chapter 4 for the two-dimensional integration also apply here. However, the potential for savings by using symbolic integration in DART is more considerable. All the univariate functions that we need to integrate have the form:

$$h(x_i) = u(k \,\&\, g(x_i)), \tag{6.6}$$

where k is a real value introduced during the approximation step and g is a template function.

Because of the particular form of the function in (6.6), we can take advantage of Maple's symbolic integration again and obtain only $|G| \leq d$ integrated functions of the form

$$\overline{h}(z,a,b) = \int_a^b u(z \,\&\, g(x_i))\, dx_i. \tag{6.7}$$

We compute these integrals in closed form and then evaluate them for various values of $z = k$.

There are special cases that contribute to alleviate the cost of the symbolic one-dimensional integration. First, when the number of different template functions in the original integrand is small or even just one. On top of that, the integration limits, $[a_i, b_i]$, can be the same for some or all the variables associated with the same template function.

With the values of the one-dimensional integrals calculated, the process returns up the recursion tree. Just as in the two-dimensional algorithm, appropriate linear algebra operations take place to compute the approximated values of each integral until the result of the original integral is obtained.

6.3 Considerations for an Efficient Implementation

Even though DART is recursive, the number of terms generated in the approximations can be large enough to demand the algorithm to be as efficient as possible. In fact, thanks to the use of the variables y_i the manipulation of the original function throughout the recursion is simpler and more efficient. Additionally, having the option to choose between symbolic or numerical one-dimensional integration is an advantage that can be significant.

Besides these two aspects we just mentioned, there are two more important areas in DART that can be improved: the optimal partition of the variables, and the caching of approximations and partial results. Let us discuss them.

6.3.1 Optimal Partition of Variables

The optimal partition of variables is important for reasons previously mentioned: extra work on approximation and the risk of using undefined values for u. However, as we will see, a good partition has also a direct effect on the performance of DART as it contributes to the reuse of computations during the process.

When thinking of how to optimize the partition of variables, we need to consider the cases for each operand individually. Let us start with addition.

For the case of a sum of variables, the transformations (6.4) can be generalized as:

$$\begin{aligned} s &= \frac{w_1 - o_1}{c}. \\ t &= \frac{w_2 - o_2}{c}; \end{aligned} \tag{6.8}$$

where

$$\begin{aligned} o_1 &= a_{w_1} \\ o_2 &= a_{w_2} \\ c &= \max(b_{w_1} - a_{w_1}, b_{w_2} - a_{w_2}). \end{aligned}$$

This means that the shifting does not have to happen symmetrically, and we can directly map the point (a_{w_1}, a_{w_2}) to $(0, 0)$. After substitution in $u(w_1 + w_2)$ we obtain

$$u(w_1 + w_2) = u(c \cdot (s + t) - o_1 - o_2) = v(s, t),$$

which is still symmetric. The original region, $[a_{w_1}, b_{w_1}] \times [a_{w_2}, b_{w_2}]$, is mapped to either $[0, b] \times [0, 1]$ or $[0, 1] \times [0, b]$ with $b < 1$ and therefore the transformation is optimal for any rectangle.

These new formulas are consistent with the ones used in Example 5, where we assumed that $x_i \in [0, 1]$. The partition

$$\begin{aligned} w_1 &= a_1\, x_1 + a_2\, x_2 + \cdots + a_k\, x_k \\ w_2 &= a_{k+1}\, x_{k+1} + a_{k+2}\, x_{k+2} + \cdots + a_{2k}\, x_{2k} \end{aligned}$$

with $a_i > 0$ therefore yields

$$\begin{aligned} o_1 = o_2 = a_{w_1} = a_{w_2} &= 0 \\ b_{w_1} = \sum_{i=1}^{k} a_i &= c_1 \\ b_{w_2} = \sum_{i=k+1}^{2k} a_i &= c_2 \\ c = \max(b_{w_1} - a_{w_1}, b_{w_2} - a_{w_2}) &= \max(c_1, c_2), \end{aligned}$$

which is consistent as claimed.

The second part of the optimization is finding a partition that minimizes $|(b_{w_1} - a_{w_1}) - (b_{w_2} - a_{w_2})|$; this is, the difference between the lengths of the two intervals. In the best case $|b_{w_1} - a_{w_1}| = |b_{w_2} - a_{w_2}|$ because then there will be no waste in the approximation region. Unfortunately this may not always be possible. For example, when we have only two variables, say $u(y_1 + y_2)$, we have no other option but to use the region $[a_{y_1}, b_{y_1}] \times [a_{y_2}, b_{y_2}]$ whether it is a square or not.

When there are more than two variables, several different partitions of $y_1 + y_2 + \cdots + y_d$ are possible. In this case, the optimal partition is calculated with the following algorithm:

1. Sort the variables by decreasing order on the lengths of their ranges. Arrange variables with identical range such that those representing the same function g_i and original range of x_i are adjacent.
2. Make $w_1 = w_2 = 0$.
3. Select the partition with longest current range and add the next y variable to that partition.
4. Take the next y variable and add it to the other partition.
5. Update ranges of w_1 and w_2 appropriately by adding the ranges of the two y variables.
6. Continue with step 3 until there are zero or one variables left.
7. If there is one variable left (d is odd), add it to the partition with shortest current range and update its range.
8. Return partitions w_1 and w_2 along with their ranges.

The algorithm runs in $O(d \log d)$ time. This additional cost does not affect the general running time because the series expansion takes $O(n^3)$ time already. For an even better performance we could do Step 1 for the first partition only, and make sure that the order of the variables after each partition is preserved for the next level of recursion.

The importance of finding a symmetric partition—when possible—goes beyond the waste on the approximation region. One of the additional benefits is related to the next implementation topic: caching. The probability of reusing approximations and integrals is increased when the partition is symmetric either to the level of

variables y_i, or when considering the expression in terms of the original variables x_i.

There is one more benefit of symmetric partitions that although smaller, is worth mentioning. Recall that the function we are approximating can be different than the function we need to integrate because variables y_i have been used to represent $g_i(x_i)$. This means that in the linear algebra representation of the series, as in (4.2), the integration of $\mathbf{V}(x)$ can be different than the integration of $\mathbf{V}(y)$. When they are the same we only need to multiply the matrix $\mathbf{L}$ by one vector instead of two. The detection of symmetry for this purpose can easily be added to the loop of the partition algorithm.

This concludes the discussion of the variable partitioning for sums. We now consider what happens when we partition a product of the variables y_i.

For the case of a product, the transformations (6.4) can also be modified and have a more general form:

$$\begin{aligned} s &= \frac{w_1}{c_1} - o. \\ t &= \frac{w_2}{c_2} - o; \end{aligned} \tag{6.9}$$

thus

$$u(w_1 w_2) = u(c_1 \cdot (s + o) \cdot c_2 \cdot (t + o)) = v(s, t).$$

In this case the operation that does not have be done symmetrically is scaling. Shifting of the original region, however, must still be done symmetrically.

Unfortunately, at the time of writing this dissertation optimal formulas for c_1, c_2 and o had not been developed and the general form of the substitutions as in equations (6.4) was used. Initial work on this indicated several different cases need to be handled when the ranges of w_1 or w_2 include both positive and negative values.

The optimal partition of a product of variables is also more difficult to predict unless we know something else about them, e.g. the ranges of all y_i variables are non-negative. The reason is because computing the product of two intervals requires the checking of four different numeric multiplications, which translates into a large number of combinations to try when several intervals are multiplied in sequence. In addition, the optimal partition needs to consider not only the length of intervals $[a_{w_1}, b_{w_1}]$ and $[a_{w_2}, b_{w_2}]$, but also the distance of points (a_{w_1}, a_{w_2}) and (b_{w_1}, b_{w_2}) to the diagonal $y = x$.

Nonetheless, the partitioning algorithm for a sum can be extended to a product and it does help in finding symmetric partitions, which is also very important. This is, therefore, the current choice of implementation.

In conclusion, when the operation identified in the pattern is a product, an algorithm to determine the optimal partition may exist but is not yet available.

6.3.2 Caching vs. Space

A great advantage of DART is that when the integrand has symmetries associated with the sum or product, the possibilities of reusing computations can be very high. From a multi-dimensional point of view, the two areas where computation time is spent are approximations and integrations.

The bivariate functions deconstructed in DART always have the following form

$$v(s,t) = u(k \,\&\, (c_1\, s + o_1) \,\&\, (c_2\, t + o_2)),$$

where k is a constant (possibly zero) introduced at higher levels of the recursion, and c_1, c_2, o_1, and o_2 come from the (w_1, w_2) to (s,t) transformations. Caching the result of an approximation implies storing the list of n splitting points and matrices $\mathbf{L}$, $\mathbf{D}$, and $\mathbf{P}$. Matrices $\mathbf{D}$ and $\mathbf{P}$ can be stored in one or two vectors, but matrix $\mathbf{L}$ has at least $(n^2 - n)/2$ values to be stored. Therefore the cost of caching approximations is very high, $O(n^2)$ per approximation, which makes it recommendable only when strictly necessary.

On the other hand, caching values of integrals is much cheaper in terms of space. In this case, it is the search key that takes space, as we need to include in it the list of template functions g_i, and corresponding ranges $[a_i, b_i]$ adequately sorted. Therefore the cost of caching integral results is $O(d)$ per integral.

6.4 How DART can be Extended

The power of DART relies on how many different template functions can be identified. At the time of writing this thesis, the following template functions were supported by the author's Maple implementation:

- $g(x) = c\, x^k$, where k is a non-negative integer, and c is any nonzero real number.

- $g(x) = c\,|x-d|$, where c is any nonzero real number and d is any real number.
- $g(x) = c\,(x-d)^2$, where c is any nonzero real number and d is any real number.
- $g(x) = (c + (x-d)^2)^{-1}$, where c is any nonzero real number and d is any real number.
- $g(x) = \cos(\psi(x))$ and $g(x) = \sin(\psi(x))$. We can assume that $g(x) \in [-1, 1]$ for any range of $\psi(x)$.

The first and more important way how DART can be extended is by adding support for more template functions. This, however, depends on the power of the symbolic engine.

The current implementation of DART supports the addition of template functions by the user at any time. Support for a new template function is done by providing a pattern detection routine. The arguments to the routine are a univariate expression representing $g(x)$ and an interval $[a, b]$ representing the range of x. The routine will do a pattern matching to determine if it can compute the range of $g(x)$, which is then returned. As can be seen, a routine like this is not very complicated to code and multiplies the power of DART as every new pattern detection routine is considered together with existing ones.

Other possible extensions to DART include:

- Add a mechanism to detect at which level a traditional numerical multiple integration method is less expensive than recursive tensor product based approximations and use it then.
- Extend the template functions to accept multivariate functions. This is possible without very much difficulty as long as each variable in f is part of only one template function g.
- Investigate if there are cases where another operator (or function) besides addition and multiplication is necessary.

Let us finish the section and the chapter with a short remark about **ridge functions**, which we encountered earlier in Example 5. Univariate functions of a linear combination of the variables have the form

$$f(x_1, x_2, \ldots, x_d) = u\left(\sum_{i=1}^{d} a_i\, x_i\right),\ a_i \in \mathbb{R}$$

and are known as ridge functions. It has been proved that every continuous function on a compact subset of Euclidean space can be uniformly approximated by a finite sum of ridge functions [9]. Therefore, this special class of integrands can be used to develop a very general method for multiple integration that would combine approximation by ridge functions and approximation by Geddes series expansions.

Chapter 7

Results in Higher Dimensions

Consistent with the implementation of TPA in two dimensions, the implementation of DART was done in Maple 9. The computing environment used for the implementation and testing presented in this chapter was the same as described in Chapter 5.

For the tests in this chapter, we do not consider the behaviour of the algorithm at various precisions. We rather focus on testing various dimensions for the integration and fix the accuracy requested at a value of $\varepsilon = 5 \times 10^{-10}$.

From a general point of view, DART is quite different than TPA as DART involves recursion and pattern matching routines. Nonetheless, since both use approximations by Geddes series a common procedure is used to generate them.

7.1 The Suite of Test Integrands

Some authors have compiled suites of multidimensional test problems from various application areas. Our test integrands are based on a very comprehensive list given by Burkardt [3]. It includes a set of functions originally given by Genz [15] in 1984 and widely used to test multidimensional integrals since. Continuous functions that fit DART's patterns were selected, and some other integrals of particular interest for testing DART were added. Table 7.1 shows the various families of integrands that were ultimately selected for the tests.

We will refer to a family of functions with the uppercase letter F, and to a specific function that is an instance of a family with the lowercase letter f.

$F_1 = \left(\sum_{i=1}^{d} x_i\right)^2$	$F_{13} = \exp\left(\sum_{i=1}^{d} \lvert x_i - 0.5\rvert\right)$
$F_2 = \left(\sum_{i=1}^{d} x_i\right)^5$	$F_{14} = \exp\left(\sum_{i=1}^{d} x_i\right)$
$F_3 = \left(\sum_{i=1}^{d} (2x_i - 1)\right)^4$	$F_{15} = \exp\left(\sum_{i=1}^{d} b_i x_i\right)$
$F_4 = \left(\sum_{i=1}^{d} (2x_i - 1)\right)^6$	$F_{16} = \exp\left(\prod_{i=1}^{d} x_i\right)$
$F_5 = \left(\sum_{i=1}^{d} b_i x_i\right)^2$	$F_{17} = \ln\left(1 + \sum_{i=1}^{d} x_i\right)$
$F_6 = \left(\sum_{i=1}^{d} b_i x_i\right)^3$	$F_{18} = \left(1 + \sum_{i=1}^{d} 2x_i\right)^{-1}$
$F_7 = \cos\left(\sum_{i=1}^{d} x_i\right)$	$F_{19} = \sin^2\left(\frac{\pi}{4} \sum_{i=1}^{d} x_i\right)$
$F_8 = \cos\left(\sum_{i=1}^{d} a_i x_i\right)$	$F_{20} = \prod_{i=1}^{d} \sin(\pi\, x_i)$
$F_9 = \prod_{i=1}^{d} \left(a_i^{-1} + (x_i - 0.5)^2\right)^{-1}$	$F_{21} = \prod_{i=1}^{d} \cos(i\, x_i)$
$F_{10} = \left(1 + \sum_{i=1}^{d} a_i x_i\right)^{-(d+1)}$	$F_{22} = \left(\sum_{i=1}^{d} x_i\right)^2 \cos\left(\sum_{i=1}^{d} x_i\right)$
$F_{11} = \exp\left(-\sum_{i=1}^{d} a_i (x_i - 0.5)^2\right)$	$F_{23} = (\pi/2)^d \sin\left(\prod_{i=1}^{d} x_i\right)$
$F_{12} = \exp\left(-\sum_{i=1}^{d} a_i \lvert x_i - 0.5\rvert\right)$	$F_{24} = (\pi/2)^d \sin\left(\prod_{i=1}^{d} x_i^{c_i}\right)$

Table 7.1: Families of Integrands Tested with DART

The first five of the six families given by Genz [15] are represented by F_8 to F_{12}, and they respectively correspond to: oscillatory, product peak, corner peak, Gaussian, and continuous integrands. The sixth family is discontinuous integrands, which is the reason why we cannot consider it for DART. In these families, the parameters a_i are meant to drive the hardness of the problem: the larger the value $\sum |a_i|$, the more difficult to approximate.

At a given dimension d, random values were generated for the constants a_i, b_i, and c_i. These values were uniformly chosen in the following manner: for a_i from $\{1/5, 2/5, 3/5, 4/5, 1\}$, for b_i from $\{-5, -4, -3, -2, -1, 1, 2, 3, 4, 5\}$, and for c_i from $\{1, 2, 3, 4, 5\}$. Although there is no reason to believe that such a distribution of values is common in actual applications, there is no reason to assume that they will rarely happen either. On the other hand, having so few possible values for these constants promotes the use of series expansions and integrals that have been cached. Therefore, let us interpret the integrands as mere examples and not a fully representative sample of the family.

7.2 Results and Analysis

The following characteristics apply to the computations with DART here presented:

- The requested accuracy was $\varepsilon = 5 \times 10^{-10}$. However, DART uses 14 digits to do the computations, because that is the limit of the hardware floating-point computation in the machine used to run the tests.
- The dimensions considered were chosen from a representative set of 4, 6, 8, 12, 15, 16, 32, 64, 128, 256, and 512.
- The integration region was always the unit d-cube: $x_i \in [0, 1]$ for all $1 \leq i \leq d$.
- Two sets of 512 random numbers were generated to be used as coefficients a_i, b_i, and c_i in the integrands. The lists of such coefficients are included in Appendix A.

The current DART implementation was written to accept three additional boolean arguments that indicate whether: symbolic or numerical one-dimensional integration should be used, caching of approximations (Geddes series expansions) takes place or not, and caching of integral values takes place or not. On average, the best results were obtained by using numerical one-dimensional integration and caching of both approximations and integrals. This is, therefore, our base configuration as some variations are analyzed with respect to it.

Tests were run independently for each algorithm in increasing order of dimension while reasonable computation times were obtained (usually up to a few minutes). This criterion provided 212 integrands from our 24 test families listed in Table 7.1, on which the analysis takes place. Table 7.2 lists the results for each family at the highest dimension tested. For each family of integrands we can observe how many dimensions DART can handle in reasonable time, and what the results are when we bring DART to its limits.

First, let us analyze the accuracy of the results obtained. Then we will proceed to consider what happens when DART is run with no caching of approximations or integrals, and when symbolic one-dimensional integration is used. We will see how approximation caching and one-dimensional symbolic integration do not seem to bring the advantages that in theory they may promise.

Family	Dimension	Time	Result	Estimated RelErr	Actual RelErr	$\lvert I_n\rvert/\lvert\lvert f\rvert\rvert_\infty$
1	128	24.376	$4.1066666666652 \times 10^{03}$	5.0×10^{-13}	3.6×10^{-13}	2.5×10^{-01}
2	64	71.453	$3.5315626666783 \times 10^{07}$	1.8×10^{-10}	3.3×10^{-12}	3.3×10^{-02}
3	32	31.202	$3.3706666669082 \times 10^{02}$	1.1×10^{-10}	7.2×10^{-11}	3.2×10^{-04}
4	16	19.920	$2.1089523806122 \times 10^{03}$	1.1×10^{-09}	1.6×10^{-10}	1.3×10^{-04}
5	32	49.703	$3.1000000000006 \times 10^{01}$	5.9×10^{-13}	1.9×10^{-13}	1.2×10^{-02}
6	16	21.266	$-1.9250000000003 \times 10^{02}$	5.5×10^{-12}	1.6×10^{-13}	9.8×10^{-03}
7	512	37.921	$-1.8045810938095 \times 10^{-11}$	2.6×10^{-07}	3.1×10^{-10}	1.8×10^{-11}
8	64	39.063	$1.4835075898465 \times 10^{-01}$	7.3×10^{-12}	3.1×10^{-12}	1.5×10^{-01}
9	512	23.566	$1.6420395247860 \times 10^{-155}$	3.7×10^{-11}	1.1×10^{-11}	3.7×10^{-11}
10	16	23.204	$6.5121417560075 \times 10^{-14}$	2.1×10^{-04}	9.9×10^{-01}	6.5×10^{-14}
11	512	5.968	$1.1705480620246 \times 10^{-11}$	1.2×10^{-11}	3.6×10^{-12}	1.2×10^{-11}
12	512	3.531	$2.6447104633969 \times 10^{-33}$	5.0×10^{-13}	7.4×10^{-12}	2.6×10^{-33}
13	512	1.078	$7.9638759332326 \times 10^{57}$	5.0×10^{-13}	8.1×10^{-13}	5.3×10^{-54}
14	512	1.937	$2.3352393763894 \times 10^{120}$	5.0×10^{-13}	7.2×10^{-12}	1.0×10^{-102}
15	256	4.391	$9.9191983800595 \times 10^{43}$	5.0×10^{-13}	1.1×10^{-11}	1.2×10^{-122}
16	256	165.248	$9.9999999999998 \times 10^{-01}$	7.8×10^{-11}	2.0×10^{-14}	3.7×10^{-01}
17	64	147.046	$3.4940406596282 \times 10^{00}$	6.9×10^{-11}	2.8×10^{-12}	8.4×10^{-01}
18	16	30.373	$5.9973550251138 \times 10^{-02}$	3.5×10^{-10}	1.5×10^{-11}	6.0×10^{-02}
19	128	99.455	$4.9999927298117 \times 10^{-01}$	5.9×10^{-12}	6.4×10^{-13}	6.7×10^{-01}
20	512	0.656	$3.8603169217883 \times 10^{-101}$	5.0×10^{-13}	1.1×10^{-12}	3.9×10^{-101}
21	128	1.094	$-1.3386244933718 \times 10^{-253}$	3.7×10^{-10}	1.4×10^{-12}	1.3×10^{-253}
22	32	92.611	$-5.6249710525974 \times 10^{01}$	3.6×10^{-08}	9.8×10^{-12}	5.7×10^{-02}
23	256	51.202	$1.3943511931729 \times 10^{-27}$	3.2×10^{-06}	3.1×10^{-03}	1.0×10^{-77}
24	128	80.763	$5.1733847587873 \times 10^{-46}$	9.6×10^{-02}	4.4×10^{-05}	4.8×10^{-71}

Table 7.2: Highest Dimensions Achieved for the Suite of Integrands

7.2.1 Accuracy

From Table 7.2 we can observe that the results obtained are prominent. In most cases the integrals were correctly approximated at relatively high dimensions. However some results cause a couple of concerns with respect to accuracy.

The following is the description of how the values in the table are calculated:

- Estimated RelErr: The error estimate provided by DART. It is calculated as the minimum of all the relative errors for every integration throughout the whole process. The relative error estimate in each integration is calculated the same way as TPA: based on the estimated absolute error on the diagonal.
- Actual RelErr: The actual relative error calculated by comparison with other results symbolically computed in Maple.
  In order to obtain a more accurate value of the integral, various symbolic methods were used. There was no single way that could provide results for all integrands at accuracies of at least 5×10^{-14} and with a reasonable running time. Symbolic integration, expansion by Maclaurin series, and (in the worst cases) specific optimized formulas for the integrand were the techniques used for the computation of the integrals. In most cases, values were computed in software floating-point with a precision of 32 digits. One exception is f_{17}, which needed over 50 digits to give a result with 15 digits of accuracy with $d = 64$ dimensions!
- $|I_n|/||f||_\infty$: The ratio in percentage between the value of the integral produced by DART and the estimated infinity norm of the integrand.

The main issue we care about is the error of the result. We would like to obtain a result that satisfies the requested accuracy; or if it is not satisfied, a correct estimate that can tell us the guaranteed accuracy. In TPA the accuracy is mainly affected by the one-dimensional integration, the lack of guard digits, and the approximations while generating the series. In DART the effect of such factors gets multiplied by the effect of the dimension, and makes accuracy more difficult to estimate.

Let us start our analysis by observing that the actual relative error provided by DART was correct for all functions except f_{10}, f_{23} and f_{24}. These three functions have a very small $|I_n|/||f||_\infty$ ratio, thus the integral is much smaller than the values used to compute it and the number of guard digits does not compensate such a difference. We shall, however, notice that although other functions have a similar or smaller ratio, they did not get affected because they are tensor products,

e.g. f_{11-15} and f_{20-21}. In general, we can affirm that the accuracy of the results given by DART is quite good.

The second concern is the quality of the relative error estimate. Two cases should be brought to our attention: the case where the estimated error is greater than the requested maximum error but the actual error is not (e.g. f_4, f_7, and f_{22}), and the case where the error estimate reported is below the actual error (e.g. f_{10}, f_{12}, f_{13}, f_{14}, and f_{20}).

In the case of f_4 and f_7, the value of the integral is just on the limit where the number of guard digits is not enough to obtain the requested accuracy because of a small $|I_n|/||f||_\infty$ ratio. Luckily, in these cases the actual accuracy was better than the requested. The problem with function f_{22} is similar but it occurs at a lower level in the recursion with integrals that do not contribute very much to the final result. This case calls for a better method to estimate absolute and relative errors.

Another cause for giving an error estimate that is too large is doing the approximation in a larger region when we generate the bivariate function to be approximated in the deconstruction step. For example, let us assume that the region where the approximation is required is $[0,1]\times[0,0.25]$, and the function to approximate is $(s+t)^2$. The infinity norm of the function restricted to the approximation region is $(1+0.25)^2 = 1.5625$; however, the whole unit square is considered for the approximation and the infinity norm is 4 instead. It is this latter value which is used in the stopping criterion and to estimate the relative error when we could have used a more adequate smaller value. In this case the difference is not very big, but in other cases such as f_5 and f_6 it can easily be.

The case where the error estimate reported is below the actual error is perhaps more complicated to study. A precise explanation is not currently available, but it should not come as a surprise that errors can be more common in DART than in TPA. In some cases, such incorrectness may be due to the significant number of sums, which may cause the occurrence of cancellation problems and their propagation.

7.2.2 Caching

Usually caching involves a trade-off between time and space. In our case, we can assume that the first priority is perhaps time. It is totally predictable that with integral caching the cost in space is low compared to the profit on speed up, and there is no need to focus very much on this aspect. Caching of approximations, however, is more a concern and will receive more attention in this discussion.

Some statistics were recorded while running the algorithm in order to allow a better understanding of the internal behaviour of DART. Table 7.3 shows those metrics that are relevant to caching for the same list of integrands from Table 7.2.

The following is the description of the columns in the table:

- Memory: The number of megabytes allocated by Maple to compute the integral. This is obtained via the Maple command `kernelopts(bytesalloc)`.
- Approximations, Cached and Usage: The number of approximations with Geddes series expansions that were cached and its usage ratio (the number of times an approximation was found in the cache / total number of approximations cached).
- Integrals, Cached and Usage: The number of multidimensional integrals whose values were cached and its usage ratio (the number of times an integral was found in the cache / total number of integrals cached).
- Integrals, 1-Dimension: The total number of one-dimensional numerical integrations performed.
- Number of Terms: The number of terms generated in the Geddes series.
  - Total: Total number of terms in all approximations.
  - Average: Average number of terms per approximation
  - Min: Minimum number of terms among all approximations.
  - Max: Maximum number of terms among all approximations.

The current implementation of DART takes advantage of the very particular pattern of the integrands, and stores the expression in a way that is independent of the variable names. This is what allows the use of previously computed approximations and integrals as much as possible. For example, if the original integrand is the function with four variables $\cos(x\,y\,z\,w)$ and the integration limits are identical for all four variables, say for the region $[0,1]^4$, at some point we need to integrate $\cos(x\,y)$ and $\cos(z\,w)$. We can see that computing only one of the two-dimensional integrals is enough because both integrals give the same result. Therefore, for caching matters it is important to refer to each variable as the range of its integration limits rather than its own name.

When using both approximation and integral caching, the overall usage ratio of approximation and integral caching was 0.576 and 1.123 respectively. This means

			Approximations		Integrals			Number of Terms			
Family	Dimension	Memory	Cached	Usage	Cached	Usage	1-Dimension	Total	Mean	Min	Max
1	128	5.64	535	0	1114	0.442	579	1605	3.00	3	3
2	64	6.55	1230	0	2418	1.065	1188	4993	4.06	3	6
3	32	5.77	512	0	1499	0.609	987	2411	4.71	4	5
4	16	5.64	289	0	1208	0.489	919	1798	6.22	5	7
5	32	7.73	759	0.668	5214	0.398	3948	2277	3.00	3	3
6	16	6.22	309	0.437	2486	0.429	2042	1236	4.00	4	4
7	512	5.83	508	0	993	0.024	485	1016	2.00	2	2
8	64	5.90	474	0.549	2035	0.388	1301	948	2.00	2	2
9	512	5.70	378	0	721	0.001	343	378	1.00	1	1
10	16	5.90	238	0.181	2091	0.888	1810	1767	7.45	1	8
11	512	5.24	40	0.650	100	0.270	34	40	1.00	1	1
12	512	6.42	38	0.737	99	0.283	33	38	1.00	1	1
13	512	6.36	9	0	10	0	1	9	1.00	1	1
14	512	5.05	9	0	10	0	1	9	1.00	1	1
15	256	5.24	49	2.224	211	0.469	53	49	1.00	1	1
16	256	7.73	1292	1.121	4792	1.228	2052	6083	4.90	3	7
17	64	7.93	2400	0	3717	2.061	1317	11377	4.74	3	14
18	16	5.64	478	0	1250	1.305	772	2880	6.03	4	13
19	128	6.49	938	0	2364	0.191	1426	2814	3.00	3	3
20	512	6.81	3	2.000	10	0	1	3	1.00	1	1
21	128	5.18	3	41.333	255	0	128	3	1.00	1	1
22	32	6.75	823	0	2568	0.695	1745	4351	5.29	4	6
23	256	5.77	383	1.431	1474	0.963	543	1126	3.11	2	5
24	128	12.45	254	32.563	10435	3.920	1910	780	3.19	2	5

Table 7.3: Internal Statistics for the Integrands from Table 7.2

that among the 212 integrands, the number of times when integrals were found in caching exceeded the total number of integrals cached by a factor of 1.123. This second ratio clearly shows the significance of integral caching.

Deactivation of approximation caching translated into an average increase of 51% in the running time. As expected, time increases in those functions that used caching approximations the most. However, the use of memory did not always decrease; in fact, there was an average increase of 0.6%.

The reason why space does not decrease as theory would suggest can be found in the way Maple manages memory. Memory is not deallocated by the user, and some objects may be unnecessarily kept in memory until garbage collection takes place. On top of this, more memory gets used when computing the cachable approximations which is reflected in an increase of the space used. Although the current implementation tries to minimize of the amount of memory that is left unreferenced in Maple so these unwanted effects do not occur, an optimal point is not always possible and the consequences become evident.

Basically, the ratio of approximation caching usage of a function directly relates to the ratio of series terms that need to be re-computed, and the number of these additional terms, in turn, is what drives the increase in the running time. On the other hand, there are some integrals for which the algorithm caches a large number of approximations that are never used, e.g. f_2, f_7, and f_{17}. These integrals are the best candidates for not caching because both the running time and space can be less than when using caching. This leads us to find some criteria that would help us decide whether approximation caching should be done or not.

Basically, caching of approximations is useful when the integrand has a symmetric partition (including ranges) in the deconstruction step, but it is not symmetric when the original x_i variables are considered. Let us consider, for instance, family $F_{24} = (\pi/2)^d \sin\left(\prod_{i=1}^{d} x_i^{c_i}\right)$. The partition in the deconstruction step is done with respect to a product of positive powers. What is convenient is that if $x_i \in [0,1]$ for all $1 \leq i \leq d$, the range of $x_i^{c_i}$ is always $[0,1]$ as well. When the exponents c_i are very different, the possibility of reusing integrals in DART is low (see also f_{21}). However, once the variables y_i come to replace $x_i^{c_i}$, the list of variables to partition is fully symmetric, even if d is odd. One can easily guess that the probability of this happening will be higher when the deconstruction is product-based rather than sum-based. This is why among our test integrands, f_{21} and f_{24} have the highest approximation caching usage.

On the other hand, cases where the partition takes place over a sum of univariate functions can also have good use of approximation caching, again, provided the

integrals are not symmetric. This is more likely to happen when the number of template functions is relatively small compared to the dimension of the integral.

The following rule can be used to know whether a symmetric integral in d dimensions will have approximations repeated or not. If

$$f(x_1, x_2, \ldots, x_d) = u\left(\sum_{i=1}^{d} g(x_i)\right),$$

$x_i \in [a, b]$, and $d \in \{2^k, 3 \times 2^k, 5 \times 2^k\}$ for $k \geq 0$, then the usage of approximation caching is 0 and it should not be used. This happens because the deconstruction step will generate symmetric partitions at all the levels (except perhaps the last two), such that integral caching always happens. This is the case for functions f_{1-4}, f_7, f_{13-14}, f_{17-19}, and f_{22}.

These guidelines just mentioned should be considered for adding a smart way of deciding when to use approximation caching to DART.

Now let us mention some observations from integral caching. First of all, in most cases this type of caching is very worthwhile. The increase in the space and running time can be extremely high when no integral caching occurs for integrands that are symmetric and have series expansions with a larger than average number of terms. Although we can observe some functions, such as f_{13}, f_{14}, f_{20}, and f_{21}, that do not use caching at all, they are tensor products of rank one that only need to compute one integral per level. Other cases where integral caching will not help include integrands that are very far from product-based or sum-based symmetry.

At any level, the partition of the operands into a product or a sum can generate either k or $2\,k$ integration subproblems, depending on whether the partition is fully symmetric or not. Here k is the number of terms generated in the approximation. This saving in integral computation is not interpreted as integral caching.

It should be noticed that the number of dimensions affects the caching usage of both series expansions and integrals because it may introduce asymmetries in the partitions.

We have seen how symmetry in the partition allows a much better performance by the use of caching techniques. As a conclusion, caching is very much worthwhile, especially with an implementation of DART in Maple. If space is a concern, some heuristics can be developed to automatically determine when not doing each type of caching could be better.

7.2.3 Symbolic One-Dimensional Integration

When the numerical one-dimensional integration was substituted by symbolic integration, the performance was unexpectedly slower. The average increase in running times for the 212 test integrands was 27%. On top of this, the memory used increased by an average of 7%. The worst cases occurred with f_{12} and f_{13} which respectively reported an increase of 23% and 21% in space, and 89% and 68% in time.

The cause of such a slowdown is mostly the complexity of the new functions resulting from the symbolic integration. The integral result can be significantly larger than the original integrand, and its evaluation may require the use of limits which makes it much more expensive. Therefore, we can conclude that—at least for now—numerical one-dimensional integration should be the preferred method to be used in DART.

7.3 Estimating the Performance of DART

At this point we have a good understanding of the performance of DART. Some of the families of integrands clearly run in linear, or even sub-linear time with respect to the dimension. For other families the running time seems to be rather polynomial. This mostly depends on the amount of caching usage for each case. The question now is whether we can, in general, conclude that DART runs in polynomial time, thus breaking of the curse of dimensionality for the families of integrands that DART supports. We will see that under a safe assumption DART does break the curse.

From Chapter 5 we know that there is a well established relationship between the accuracy and the number of terms needed to achieve that accuracy. This is not possible to affirm for DART because the number of terms in each series expansion varies with the current dimension unless the integrand is a tensor product.

Let us assume that for a given class of integrands that DART can solve, the number of terms of the Geddes series generated in the expansions has an upper bound n. We will see that in this case the running time of DART is polynomial with respect to the dimension or with respect to the number of digits of accuracy of the result, but not both.

Every problem throughout the recursion consists in the approximation of an integral in d dimensions. When $d = 1$ the integration is done numerically and its

cost is independent of d. When $d > 1$ the solution requires an approximation by Geddes series expansion and subsequent calls to subproblems of dimension $\lfloor d/2 \rfloor$ or $\lceil d/2 \rceil$. Let us assume that the number of terms of the series is bounded by n. From Chapter 5, we know the cost of the series expansion as given by (5.2). This allows us to define the cost of the integration in DART with the recursive formula:

$$C(d) \leq 2\,n\,C(d/2) + c_1\,n^3 + c_2\,n^2 + c_3\,d\,\log(d).$$

Using the Master Method[1] to solve for $C(d)$ requires us to compare

$$d^{\log_2(2\,n)} \text{ and } c_1\,n^3 + c_2\,n^2 + c_3\,d\,\log(d).$$

Notice that for any value of $n \geq 4$ and $\epsilon = 1$, then

$$c_1\,n^3 + c_2\,n^2 + c_3\,d\,\log(d) = O(d^{\log_2(2\,n)-\epsilon}).$$

With this relation, we can finally conclude that:

$$C(d) \leq \Theta(d^{\log_2(2\,n)}) \text{ or}$$
$$C(d) \leq \Theta((2\,n)^{\log_2(d)}),$$

which indicates that the running time of DART is polynomial with respect to the dimension (for fixed accuracy) and with respect to the number of digits of accuracy (for fixed dimension). (Recall that we assumed $n = k_1 + D\,k_2$).

Adding the cost of the one-dimensional integration to the inequality gives:

$$C(d) \leq \Theta(d^{\log_2(2\,(k_1+D\,k_2))}\,c_{Int}(D,c)),$$

where $c_{Int}(D,c)$ is the same one-dimensional integration cost defined for TPA in Chapter 5. This shows that under the assumptions used in our analysis DART does break the curse of dimensionality.

In the tests here presented, we saw that the largest value of n among all integrands was 14 for a requested accuracy of 5×10^{-10}. This would lead us to say that since for all the integrands $n \leq 16$, their computational cost would be $O(d^5)$.

[1]The Master Method can be found in any textbook on algorithm analysis.

7.4 General Observations

In general, DART works very fast when the series approximations have very few terms, or when the symmetry allows significant reuse of approximations and integral calculations.

Comparing the results from DART with the two numerical algorithms available in Maple, DCUHRE and Monte Carlo (from NAG), shows a big difference both at low and high dimensions. In low dimensions, usually up to $d = 6$, DART is easily beaten by both numerical methods (but recall that we have not implemented a compiled version of DART). However, in high dimensions neither DCUHRE nor Monte Carlo can achieve the results of DART.

On the one hand, DCUHRE does not accept problems with more than 15 variables. With only a few dimensions, about $d \leq 4$, DCUHRE gives great speed at hardware floating-point precision. As the dimension increases, about $d \geq 8$, the running time raises to a point that most integrals fail to be computed (even for polynomials such as those from F_2 and F_3).

On the other hand, the Monte Carlo algorithm allows only up to about 5 digits of accuracy, and often it achieves only two or three digits. DART also outperforms Monte Carlo in several cases, e.g. Monte Carlo only attempts the integration of $\sin\left(\prod_{i=1}^{16} x_i\right)$ with two digits of accuracy, and yet the result is wrong.

In some cases such as ridge functions, we found that the running time increases faster than with other types of problems, but this growth is not as bad as with other methods, including DCUHRE and Monte Carlo, or even symbolic methods. The resources required to perform the calculation are still manageable, and an accurate result is still possible. We have seen that even for several integrands that are not tensor products the time spent in DART is considerably less, allowing a higher limit for the dimension of the integrals that can be evaluated in reasonable time.

Now let us consider the negative factors of DART's current implementation. Most of these issues are actually related to the accuracy rather than the speed of the algorithm. Having a better method to estimate the expected accuracy is the only way to know what to do in order to guarantee a result with the requested accuracy from DART. We have seen that there are various factors that affect such an estimation. Let us summarize them here:

- The first factor is not using a criterion based on the size of I_n as the stopping criterion for the series expansions. The reasons for this, as explained before, were code modularity and saving in space. The analysis of caching in this

chapter has, however, proved the latter reason to be not very relevant in a Maple implementation.

- Secondly, the optimal partition of the y_i variables is available when it is a sum-based partition, but not for a product-based partition, in which case the domain inflation can be significantly large. Some domain inflation cannot be eliminated, even when an optimal partition of variables is available.

- Additionally, the insufficiency of guard digits when the ratio $|I_n|/||f||_\infty$ is too small also affects DART more significantly than TPA.

- Finally, the accuracy can be negatively affected by the number of levels of recursion, $O(\log(d))$, and the number of series terms, even in cases where the ratio $|I_n|/||f||_\infty$ is not very small.

7.5 Future Enhancements to the Algorithm

As we have seen, DART is an algorithm whose performance depends mostly on the generation of the Geddes series. Therefore, any improvement in the performance of the series generation will not only help TPA but it will most likely have a multiplicative effect in the performance of DART.

The last section of Chapter 6 had already listed some possible extensions to DART:

- Explore new patterns where some DART-like partition techniques can be used. Multivariate template functions are one possibility.

- Implement a criterion to intelligently determine when caching should take place or not based on observations such those from above where caching was discussed.

- Add support for more template functions.

However, the improvements that perhaps have the highest priority at the current stage of DART are accuracy-related. Several things can be done in order to give DART a better control of error estimates:

- Stopping criterion for the convergence phase. First of all, the criterion to stop the series generation must change and be based on $|I_n|$ instead of $||f||_\infty$. This modification implies the computation of $|I_n|$ after a new term is generated in the convergence phase. In DART this also implies carrying out series expansions simultaneously in all levels of recursion. The price to pay is extra space, but based on the tests presented in this chapter we can believe that it will not really affect very much.

- Approximation in a subregion. It was explained in Section 7.2.1 that one cause for giving an error estimate that is too large is the approximation taking place over a larger region than required. The minimum thing that can be done about it is to make sure that the stopping criterion of the approximation is based on the norm of the function in the subregion instead of the whole unit square. This way the expansion goes as far as it is really needed to obtain the requested accuracy in the subregion.
  Nonetheless, we can go further and design a new alternate criterion to choose splitting points such that they always intersect the target subregion. For example, if the subregion to approximate is $[0.8, 1] \times [0, 0.2] \in [0, 1]^2$, the initial estimation of $||f||_\infty$ is restricted to that region only, and the splitting points would be chosen from $[0, 0.2] \cup [0.8, 1]$. Choosing $(0.5, 0.5)$ as a splitting point will not help with the approximation in $[0.8, 1] \times [0, 0.2]$.

- Optimal partition of variables. As discussed in Chapter 6 and related to the approximation in a subregion, this optimal partition is very important. We should avoid going outside of the actual approximation region for as much as possible, and this optimization is still to be done for product-based partitions.

- Adjust the requested accuracy of the integration subproblems. This can be done in two directions. The first way is to increment the requested accuracy to compensate for possible cancellation problems propagated to higher levels of the recursion. The second option is to reduce the requested accuracy of those subproblems where it is known that they will not significantly contribute to the final result. Let us use an example to illustrate this.
  Consider the approximation by Geddes series expansion of an integral to a relative accuracy of 1×10^{-5} such that after iteration n, $I_n = 3.141592$, and the absolute error estimate of $||r_n||_\infty$ is 10^{-4}. Since the target accuracy has not yet been reached, we need to generate a new term and use it to do one or two additional recursive integrations. The new integral(s) do not really need to achieve a relative accuracy of 10^{-5}, because we know that the result will be less than 10^{-4} and therefore a relative error of 10^{-2} would be sufficient

to add to 3.141592. In this example we are saving just a couple of digits of accuracy, but in other cases this saving can be significant.

On top of these accuracy-related enhancements, DART could also benefit from adding a mechanism to detect at which level another method, say DCUHRE, is less expensive than the recursive Geddes series based approximations and use it then for added efficiency at hardware floating-point precision.

Applying symbolic one-dimensional integration has proved to be not helpful with most of the test integrands used here. However, we should not rule out this option because it could be possible to find families of integrands for which this option is advantageous.

Singularities are another topic to be investigated in the future. One issue is to find a feasible way to have the Geddes series generation handle functions with singularities in the integration region. For example, consider

$$\pi^{d/2} \int_{[0,1]^d} \cos\left(\sqrt{\frac{\sum(\phi^{-1})^2(t_j)}{2}}\right) dt,$$

where ϕ is the cumulative normal distribution function with mean 0 and variance 1:

$$\phi(u) = \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{u} e^{-s^2/2}\, ds,\ u \in [-\infty, \infty].$$

The inverse function ϕ^{-1} has singularities at both 0 and 1, which prevents DART from deconstructing the sum in a way that can be handled by the Geddes series generation routine. Integrals like this, using the cumulative normal distribution function, are common in various fields in physics [4].

Finally, we have developed a theoretical running time for DART under a few assumptions. We can, however, try to make this analysis more generic and determine the real effect of the dimension on the number of terms. This would give us a true asymptotic bound for the performance of DART. Additionally, it should be possible to state the conditions on the integrand for which DART runs in linear or sub-linear time.

Chapter 8

Conclusions

It is now the end of this dissertation. However, it is by no means the end of the work on approximation of integrals via Geddes series. The last sections of Chapters 5 and 7 have listed some conclusions derived from the analysis of the results obtained. Let us discuss the highlights.

From the work in two dimensions, we have seen how the intrinsic characteristics of the Geddes series on symmetric bivariate functions allow an efficient implementation of the series expansions after exploiting one assumption: the maximum error can be estimated from the values of the error on the diagonal. Extensive experimentation using TPA, the current implementation of the method, supports the acceptability of such an assumption. The result is an adaptive algorithm that proves to be able to handle very well various families of continuous integrands, including those with steep slopes and oscillations. Empirical results tell us that the convergence of the series for analytic functions is essentially monotonic at a linear rate. Furthermore, they also let us conceive formulae to describe the behaviour of the integration algorithm.

The extension of the method to higher dimensions via DART reveals the great potential of the Geddes series. The reduction of the dimension of an integration problem by a factor of two gives important and clear benefits for the families of functions that DART currently supports. We have seen that under a simple assumption the running time of DART is polynomial with respect to the dimension (for fixed accuracy) and with respect to the number of digits of accuracy (for fixed dimension)

The future work that can be done on TPA and DART points towards improvements coming from applying theory and more heuristics to the algorithms. On the

one hand, experimentation lets us choose heuristics and other techniques that are adequate for an optimal implementation. On the other hand, deeper study of the theory around tensor product approximation may shed some light on what other possible improvements can be added.

Alternative implementation environments can clearly improve the performance of the algorithms. First of all, implementation in a compiled language will lead to better running times, where a fair timing comparison with other methods is possible. Additionally, the type of reduction on the integration problems in TPA and DART makes them well-suited for a parallel implementation of the algorithm, as the sub-problems generated require almost no interprocess communication.

The fact that the conclusions derived from this work are mostly empirical does not imply that the theory to support or contradict these conclusions cannot be developed. This is indeed one of the main open problems for future work on the theoretical side. Proof of the assumptions made for estimating the performance of DART is needed to fully confirm the breaking of the curse of dimensionality in numerical multiple integration.

The families of functions supported by DART are based on problems dependant on the dimension. The development of alternate patterns to be used in DART or a DART-like approach can widen the capabilities of methods based on Geddes series in multidimensional integration.

Another source for future studies and development is singularity handling. It is quite possible that the reduction in the dimension can lead to better methods for handling isolated singularities in multiple integrals. Pursuing this path may also prove to be productive.

As pointed out by Chapman [7], what we present here is only one of several paths that can be taken to exploit Geddes series. For example, Geddes series can be used to make substantial improvements to Bateman's method for solving Fredholm linear integral equations.

Finally, the author thanks the reader's interest in this work, and appreciates comments and insights. The author and his supervisors hope that these results will encourage the study of more applications of Geddes series in the near future.

Appendix A

Coefficients for Testing DART

The following are the lists of coefficients used with the families of functions when testing DART in Chapter 7:

b_i = 5, -5, 5, -1, 2, -1, -2, -2, -2, -3, 4, 4, -2, -1, -4, -4, 2, 5, 3, -3, -2, -4, 1, 4, 1, 5, -3, -3, -4, -4, 2, 5, -1, 2, -4, -4, 4, -1, 4, 5, -4, 5, -5, -4, -1, 2, 2, -4, 2, -3, 2, -5, -2, -1, 5, -2, 1, -4, 3, 1, -2, -4, 5, 3, -5, -4, -5, 5, 3, -4, 3, -2, -4, 1, 2, 5, 3, 5, -5, -3, 1, -2, -1, -3, -1, -3, -5, -5, 5, -1, 4, -4, 5, 1, 3, -3, -4, 1, 4, 3, -2, 2, 2, -5, 2, 2, 2, 5, 1, -3, 3, -2, 1, -1, -3, -2, 3, -4, 3, -1, 4, -3, -5, -5, -4, 3, -1, 3, -2, 1, 2, 5, 1, -1, -2, 3, -1, 1, -1, 3, -5, 4, -1, -3, 1, -5, -3, -1, -4, 3, 5, 4, 1, -2, 4, -2, 1, -3, -2, -2, 4, -1, -1, 3, -5, 3, -3, 2, -5, -1, -1, -5, -5, 3, 4, 3, 5, -1, 5, -1, 1, 5, -1, 3, 1, -1, 4, 3, -1, -5, 3, -5, -2, 5, -3, 5, -5, -3, 4, -3, -5, -3, -2, 5, -1, -3, 2, -2, -5, 4, -2, -5, -5, 4, -4, -3, 1, 3, 4, 5, -2, -5, 2, -1, 5, 4, -5, -2, -2, -5, 3, -1, -2, -5, 4, 1, 5, 5, -4, -5, 4, -3, 2, 3, 5, 1, 4, 5, 1, 5, 2, -4, -1, -1, 1, 3, 2, -5, 3, 4, -3, 4, -2, -3, 3, -3, 4, -2, 1, 4, -4, -4, -3, -1, -4, 4, -4, 1, 5, -1, -3, 3, -5, 5, 2, -3, -1, -3, -5, -1, 4, 2, -2, 4, 1, -5, 4, -4, 4, -4, -2, -4, 1, 2, 1, 4, -4, -2, -3, -5, -5, 1, 4, 5, 2, 3, -3, -1, -1, 3, 3, -5, 1, 2, 5, 5, 5, -1, 3, 3, -4, 4, -2, -3, -1, -2, -3, 1, 5, -1, -1, 5, 1, 5, 3, -5, 4, 4, -1, 1, 3, -2, -2, -5, -3, 4, -5, -5, -1, 5, -2, -5, 4, 4, 5, 5, 5, 2, 2, 1, 1, -4, -5, 1, 4, -2, -4, 4, -3, 2, 1, 3, 2, 4, 1, 2, -1, -4, 2, 1, 1, -2, 3, 5, 3, -3, -1, 3, 4, 3, -1, 3, -3, -4, -3, 2, 4, -3, -5, -1, 1, -3, -1, 4, 3, 5, -3, -3, 1, 3, -2, -1, 2, 3, 4, 4, 3, 1, 3, 4, -5, 3, -2, 3, 2, -1, -4, 1, 3, -1, -5, 2, -1, 1, 4, 1, 2, -1, -1, -1, 1, -5, -4, -3, 3, -4, -1, 3, -3, 1, -5, -5, -3, 4, -1, -4, -5, 5, -2, 1, -1, -1, 2, 1, -5, 1, 4, -2, 4, -3, -3, -1, 2, -5, 2, -2, 3, -2, -2, 2, -3, 4, -4, -5, -4, 4, -3, -4, 4, 5, -1, 5, 5, 5, -2, 1, 1, -1, -5, -5, 1, 1.

c_i = 3, 4, 3, 1, 1, 3, 4, 1, 5, 3, 2, 5, 5, 3, 2, 2, 2, 3, 5, 4, 5, 3, 1, 4, 5, 1, 3, 5, 1, 4, 4, 2, 2, 3, 1, 3, 3, 2, 1, 2, 1, 5, 5, 2, 1, 2, 2, 1, 1, 1, 2, 1, 3, 2, 4, 4, 4, 2, 2, 2, 4, 3, 3, 4, 1, 1, 1, 5, 4, 5, 5, 1, 3, 3, 2, 1, 2, 2, 1, 5, 4, 1, 1, 4, 1, 3, 2, 4, 1, 4, 3, 2, 2, 1, 1, 2, 2, 4, 4, 4, 1, 5, 4, 4, 2, 4, 4, 5, 3, 5, 4, 1, 1, 5, 4, 4, 1, 5, 4, 2, 3, 4, 5, 1, 4, 1, 5, 2,

3, 1, 2, 4, 2, 3, 5, 3, 2, 2, 5, 1, 2, 1, 5, 1, 4, 3, 1, 2, 1, 4, 3, 4, 2, 3, 1, 4, 3, 4, 3, 3, 2,
1, 3, 5, 5, 5, 1, 5, 4, 5, 5, 5, 1, 5, 2, 3, 1, 4, 3, 4, 3, 5, 1, 5, 5, 2, 4, 3, 5, 4, 3, 4, 1, 5,
5, 2, 1, 5, 4, 1, 4, 1, 3, 4, 3, 5, 4, 1, 4, 5, 1, 2, 4, 4, 4, 2, 2, 5, 3, 1, 2, 3, 3, 3, 3, 4, 4,
2, 1, 3, 4, 1, 4, 4, 4, 1, 5, 4, 4, 2, 4, 5, 5, 1, 5, 5, 5, 1, 4, 4, 2, 3, 1, 4, 5, 1, 1, 4, 4, 5,
1, 2, 5, 1, 4, 1, 4, 5, 5, 2, 4, 2, 2, 2, 3, 1, 5, 2, 4, 3, 1, 3, 5, 3, 3, 3, 3, 3, 5, 4, 4, 3, 2,
2, 2, 3, 3, 1, 3, 2, 5, 2, 3, 2, 2, 3, 2, 5, 3, 1, 2, 1, 4, 2, 3, 4, 1, 4, 5, 3, 5, 2, 1, 2, 1, 4,
4, 5, 1, 2, 2, 3, 3, 4, 1, 5, 5, 5, 1, 3, 1, 3, 2, 1, 2, 2, 1, 1, 4, 5, 4, 4, 2, 1, 1, 5, 5, 2, 3,
5, 1, 3, 1, 5, 2, 2, 2, 1, 1, 4, 3, 4, 4, 4, 1, 2, 5, 2, 1, 3, 5, 5, 1, 5, 1, 4, 3, 3, 5, 5, 1, 4,
3, 4, 4, 1, 5, 4, 4, 1, 4, 5, 2, 4, 3, 5, 2, 5, 2, 2, 3, 5, 4, 5, 5, 5, 4, 5, 5, 4, 3, 4, 4, 2, 5,
2, 1, 2, 2, 5, 3, 5, 4, 4, 4, 3, 4, 2, 5, 1, 3, 2, 5, 4, 4, 2, 5, 1, 4, 1, 1, 5, 1, 1, 3, 1, 2, 2,
2, 3, 4, 1, 2, 5, 5, 3, 4, 4, 5, 2, 3, 3, 2, 1, 3, 3, 4, 5, 1, 3, 1, 1, 5, 1, 1, 3, 2, 3, 4, 5, 4,
5, 1, 5, 3, 4, 3, 5, 5, 3, 5, 4, 5, 5, 4, 1, 1, 3, 3, 5, 2, 4.

$a_i = c_i/5$ for all $1 \leq i \leq 512$.

Bibliography

[1] J. Bernsten, T. O. Espelid, and A. C. Genz. An Adaptive Algorithm for the Approximate Calculation of Multiple Integrals. *ACM Transactions on Mathematical Software*, 17:437–451, 1991.

[2] J. Bernsten, T. O. Espelid, and A. C. Genz. Algorithm 698: DCUHRE - An Adaptive Multidimensional Integration Routine for a Vector of Integrals. *ACM Transactions on Mathematical Software*, 17:452–456, 1991.

[3] J. Burkardt. Test Problems for Approximate Integration (Quadrature) in M Dimensions, August 2003. www.csit.fsu.edu/~burkardt/f_src/testnint/testnint.html.

[4] S. Capstick and B. D. Keister. Multidimensional Quadrature Algorithms at Higher Degree and/or Dimension. *Journal of Computational Physics*, 123:267–273, 1996.

[5] F. W. Chapman. Theory and Applications of Dual Asymptotic Expansions. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 1998.

[6] F. W. Chapman. An Adaptive Algorithm for Uniform Approximation by Tensor Products: With Applications to Multiple Integrals and Integral Equations. University of Waterloo, 2000. PhD thesis proposal.

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

[8] F. W. Chapman and K. O. Geddes. A New Symbolic-Numerical Method for Calculating Multiple Integrals. Poster presented at ISAAC '01. University of Western Ontario. London, Ontario, Canada, 2001.

[9] W. Cheney and W. Light. *A Course in Approximation Theory.* The Brooks/Cole Series in Advanced Mathematics. Brooks/Cole Publishing Co., Pacific Grove, California, 2000. Chapters 22-24, pages 165-168.

[10] P. Van Dooren and L. de Ridder. An Adaptive Algorithm for Numerical Integration over an N-Dimensinal Cube. *Journal of Computational and Applied Mathematics*, 2(3):207–217, 1976.

[11] K.O. Geddes. Numerical Integration in a Symbolic Context. In B. W. Char, editor, *Proceedings of SYMSAC'86*, pages 185–191, New York, 1986. ACM Press.

[12] K.O. Geddes and G. J. Fee. Hybrid Symbolic-Numeric Integration in Maple. In P. Wang, editor, *Proceedings of ISAAC'92*, pages 36–41, New York, 1992. ACM Press.

[13] W. M. Gentleman. Implementing Clenshaw-Curtis Quadrature, I Methodology and Experience. *Communications of the ACM*, 15(5):337–342, May 1972.

[14] W. M. Gentleman. Implementing Clenshaw-Curtis Quadrature, II Computing the Cosine Transformation. *Communications of the ACM*, 15(5):343–346, May 1972.

[15] A. C. Genz. Testing Multidimensional Integration Routines. In B. Ford, J. C. Rault, and F. Thomasset, editors, *Tools, Methods, and Languages for Scientific and Engineering Computation*, pages 81–94, Amsterdam, 1984. Elsevier North-Holland.

[16] A.C. Genz and A. A. Malik. An Adaptive Algorithm for Numerical Integration over an N-Dimensional Rectangular Region. *Journal of Computational and Applied Mathematics*, 6:295–302, 1980.

[17] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods.* Methuen, 1964.

[18] T. X. He. *Dimensionality Reducing Expansion of Multivariate Integration.* Birkhaűser, 2001.

[19] F. J. Hickernell. What Affects Accuracy of Quasi-Monte Carlo Quadrature? In H. Niederreiter and J. Spanier, editors, *Monte Carlo and Quasi-Monte Carlo Methods*, pages 16–55. Springer-Verlag, Berlin, 2000.

[20] N. M. Korobov. *Number-Theoretic Methods in Approximate Analysis*. Fizmatgiz, Moscow, 1963. in Russian.

[21] C. Lemieux and P. L'Ecuyer. On Selection Criteria for Lattice Rules and Other Quasi-Monte Carlo Point Sets. *Mathematics and Computers in Simulation*, 55(1-3):139–148, 2001.

[22] N. C. Metropolis and S. M. Ulam. Monte Carlo Method. *Journal of the American Statistical Association*, 44:335–341, 1949.

[23] A. Papageorgiou and J.F. Traub. Faster Evaluation of Multidimensional Integrals. *Computational Physics*, 11(6):574–578, June 1997.

[24] R. D. Richtmyer. On the Evaluation of Definite Integrals and a Quasi-Monte Carlo Method Based on Properties of Algebraic Numbers. Report LA-1342, Los Alamos Scientific Laboratories, Los Alamos, NM, 1952.

[25] I. H. Sloan and S. Joe. *Lattice Methods for Multiple Integration*. Oxford University Press, 1994.

[26] G. K. Smyth. Numerical Integration. In *Encyclopedia of Biostatistics*, pages 3088–3095, Wiley, London, 1998. P. Armitage and T. Colton.

[27] A. H. Stroud. *Approximate Calculation of Multiple Integrals*. Prentice-Hall, 1971.