"Multidimensional Quadrature"

by

Peter John King

A thesis submitted to the University of Aston in Birmingham for the degree of Doctor of Philosophy.

Department of Computer Science

October 1983

The University of Aston in Birmingham

Multidimensional Quadrature

by

Peter John King

Summary

In this thesis original software is presented for the approximate evaluation of multiple integrals over two basic regions of integration; the simplex and the hypercube. The majority of the work is based upon an adaptive approach. One exception is a program which generates a sequence of product rules from the one dimensional Patterson family of formulae and applies these rules iteratively over a hypercube type region of integration. A basic adaptive algorithm for multidimensional quadrature is described and programs for both the simplex and the hypercube based on this are presented. The problems of testing and comparison of quadrature routines are discussed. Two alternative approaches to storing integrand evaluations are presented; the first using a linked list type data structure while the second uses scatter storage techniques. Programs for the hypercube which use the basic adaptive strategy but store the integrand evaluations illustrate the storage techniques. It is suggested that the storing of integrand evaluations is only feasible for particularly "expensive" integrands. A case is presented for the adoption of a global subdivision strategy as opposed to a local subdivision strategy in the construction of multidimensional quadrature algorithms. A global strategy reduces the total number of integrand evaluations used by producing a result which is closer to the required tolerance. One approach to extending the methods to other regions of integration is described and the problems associated with this are considered. Finally the possibilities of using a multiple processor for the evaluation of multiple integrals are discussed.

Keywords: adaptive multidimensional quadrature, numerical software

A thesis submitted to the University of Aston in October 1983 for the degree of Doctor of Philosophy.

Acknowledgement

I would like to thank Dr. L.J.Hazlewood for his help and assistance throughout the entire project and to acknowledge the S.R.C. for funding the first two years of this research.

Contents

Summary

Contents

	Page
Chapter 1 - Introduction	1
1.1 Description of the thesis	1
1.2 Background to the study	1
1.3 A brief history of multidimensional quadrature	2
1.4 Decisions taken as a result of the literature survey	6
1.5 Objectives	8
1.6 Contents of the chapters	13
Chapter 2 - Product Formulae	16
2.1 Introduction	16
2.2 The construction of a product formula for the hypercube	17
2.3 General cartesian product formulae	19
2.4 The construction of product formulae for the simplex	20
2.5 The degree of the formulae used in the construction of a product formula	24
2.6 A set of product rules based upon Patterson's formulae	24
2.7 The construction of a product type multidimensional quadrature routine	26
2.8 Testing the routine	39
2.8 Conclusions	39
Chapter 3 - The testing of quadrature routines	41
3.1 Introduction	41
3.2 "Battery" type testing	42

3.3 The performance profile approach to testing 43

3.4 Comparison between these two approaches to testing 50

3.5 A possible approach to testing multidimensional quadrature routines 51

3.6 the method of testing adopted in this research 52

3.7 The set of test problems used 53

Chapter 4 - A basic adaptive multidimensional quadrature procedure 55

4.1 Introduction 55

4.2 The brief description of the basic approach 55

4.3 The structure of the basic algorithm 58

4.4 The subdivision strategy used with the hypercube 60

4.5 The basic rules used for the hypercube 62

4.6 Defining the integrand 63

4.7 Segmentation of the program 64

4.8 The simplex as a basic region 65

4.9 The hypervolume of a simplex 66

4.10 The centroid of a simplex 67

4.11 Area coordinates 68

4.12 The subdivision strategy used with the simplex 69

4.13 The structure of the algorithm as applied to the simplex 71

4.14 The basic rules used for the simplex 71

4.15 The program for the simplex 75

4.16 The data required by the program 75

4.17 Testing the two programs 77

4.18 Conclusions 78

Chapter 5 - Storing the integrand evaluations 82

5.1 The need for storing integrand evaluations 82

5.2 The basic rules and the subdivision strategy for the hypercube 83

5.3 Storing the integrand evaluations in a linked list 83

5.4 Ordering the list 85

5.5 The generation and use of linked lists 85

5.6 A procedure to search a linked list 86

5.7 A procedure to insert an item in a linked list 89

5.8 Enumerating the keys 91

5.9 The basic program 98

5.10 Scatter storage techniques 99

5.11 The development of an algorithm based upon scatter storage techniques 100

5.12 Key generation 101

5.13 Hash codes 103

5.14 Finding the integrand evaluation at a particular node 104

5.15 Computing an estimate to the interal over the subregions 110

5.16 The program using scatter storage techniques 113

5.17 Testing the two approaches to storing the integrand evaluations 113

5.18 Comparison between the two approaches 113

5.19 Conclusions 114

Chapter 6 - Global subdivision strategies 116

6.1 Introduction 116

6.2 A modification of the basic algorithm for the hypercube

to use a global subdivision strategy 119

6.2.1 Adding a node to the list of subregions 121

6.2.2 Adopting a doubly linked list to store the details of the subregions 124

6.3 The complete program using this strategy 127

6.4 Testing the program 127

6.5 Conclusions 127

Chapter 7 - Extension of the methods to other regions 130

7.1 Introduction 130

7.2 Extension of the region of integration 130

7.3 Subdivision of a region into a region of simplexes and hypercubes 132

7.4 The basic structure of an algorithm 136

7.5 Defining the original region as a linked list 138

7.6 Storing the linked list 141

7.7 Processing the list of subregions 147

7.8 The complete program 149

7.9 A simple test problem 149

7.10 Conclusions 150

Chapter 8 - Multiprocessor techniques 153

8.1 Introduction 153

8.2 The architecture of a multiprocessor 153

8.3 Multidimensional quadrature as a suitable task for solution on a multiprocessor 155

8.4 An algorithm for use on a multiprocessor 156

8.5 Conclusions 161

Chapter 9 - Conclusions 162

Appendices 167

Appendix 1 - The set of test problems 168

Appendix 2 - The product Patterson formulae program 171

Appendix 3 - The basic adaptive programs

3.1 The basic adaptive program for the hypercube 178

3.2 The basic adaptive program for the simplex 193

Appendix 4 - Timings for basic operations on the I.C.L.1904s 202

Appendix 5 - The programs using stored integrand techniques

5.1 The program using linked list techniques 203

5.2 The program using scatter storage techniques 217

Appendix 6 - The program using a global subdivision strategy 226

Appendix 7 - The program for an extended region 231

Appendix 8 - The test results for the hypercube programs 240

Appendix 9 - The test results for the simplex program 274

References 279

Chapter 1 Introduction

1.1 Description of the thesis

This thesis is concerned with the development of algorithms for the approximate evaluation of multiple integrals on a mainframe computer. That is integrals of the form :

$$\int_{Rn} \cdots \int f(x_1, x_2 \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

where the region of integration, Rn, is a given region in n dimensional Euclidean space En and the function $f(x_1, x_2 \ldots x_n)$ is Riemann integrable over this region. The work is limited to the evaluation of multiple integrals over two main regions : the hypercube and the simplex. Classical type methods (as opposed to Monte Carlo type) are used exclusively. The majority of the algorithms are based upon an adaptive approach. However, one exception to this is the algorithm based upon a product type rule. The emphasis of the work is on the development of efficient, reliable numerical software based on the existing mathematical theory, not on the extension of the theory of numerical integration.

1.2 Background to the study

Many eminent mathematicians including A.H.Stroud and J.N.Lyness have devoted a large amount of time and effort to the research and development of the theory of the approximate evaluation of multiple integrals. However, the application of this theory has lagged far behind its growth because of the lack of effort in the field of

numerical software written to apply the theory efficiently and reliably. Many mathematicians consider programming a trivial task and disregard it; while most computer scientists have neither the mathematical knowledge nor the motivation to approach the task of writing rigorous numerical software. Most important of all the potential users of the theory, engineers and scientists, have both insufficient time and insufficient knowledge of either the mathematical theory or computer science to produce adequate software. Thus the author has attempted to find the parts of the theory which are most suitable for use in computer programs and to bridge the gap between this theory and the potential user by the development of efficient and reliable numerical software.

1.3 A brief history of multidimensional quadrature

Archimedes, Heron and Pappus were among the first people to study areas and volumes. These studies can be considered as the start of the history of integration. From these early beginnings the impetus was given for the evolution of the continuous calculus, which spread rapidly to a wide variety of applications. Inevitably, problems arose in which the integrals formed could not be solved analytically and hence, the study of numerical integration began early in the history of the calculus.

The majority of the work on quadrature was concerned with one dimensional problems. However, as early as 1877 a paper was published by James Clerk Maxwell [42] which gave two formulae for numerical integration over the cube. The theory of the solution of multiple integrals advanced very slowly from this point possibly due

to the large amount of computation required for anything but the simplest of problems. In fact only about fifteen papers are recorded as being written on the subject prior to 1945.

With the advent of the digital computer a greater impetus was given to the advance of the theory of multidimensional quadrature since the amount of calculation that could be considered feasible was increased dramatically. Two quite distinct approaches to multidimensional quadrature were adopted. These are firstly, the use of Monte Carlo type methods, and secondly, the use of classical or systematic type methods. The Monte Carlo type methods are based on statistical random number sampling techniques whereas the classical type methods are based on weighted sums of integrand evaluations at predefined nodes. That is formulae of the type:

$$\int_{Rn}\cdots\int f(x_1 x_2 \ldots x_n) dx_1 dx_2 \ldots dx_n \approx \sum_{i=1}^{n} Bi\ f(v_{i,1} v_{i,2} \ldots v_{i,n})$$

where the Bi are the weights, or coefficients, of the formula and the $v_{i,1} v_{i,2} \ldots v_{i,n}$ are the nodes.

A discussion of Monte Carlo and other number theoretic methods is given in Stroud[57] chapter 6 and Zakrzewska[59]. Tables of classical type formulae are given in Stroud[57] chapter 8. Specific examples of formulae for the hypercube are given in Lyness and McHugh[40] and Piessens and Haegmans[50], and for the simplex in Cowper[6], Hammer, Marlowe and Stroud[20], Hillion[22], Lyness[36] and Silvester[54].

A variety of attributes of a multidimensional quadrature problem are

relevant in deciding whether to use classical or Monte Carlo type methods. Typically, the classical type methods are more suitable for low dimensional problems where the region of integration is familiar, for example the hypercube or the simplex, a high degree of accuracy is required and for which the integrand is analytic and smooth. Whereas Monte Carlo type methods are more suited to high dimensional problems over unfamiliar or erratic regions with a low accuracy requirement and possibly with a highly discontinuous integrand. The areas of application are obviously overlapping and somewhat blurred.

During the period 1945 to 1960 an increasing amount of effort went into the theory of multiple integrals and between 1960 and 1970 considerable progress was made with respect to classical methods. The state of the art of the subject in 1970 formed the prime subject of a paper by Haber [18] in 1970. Essentially, quadrature rules for many standard regions of integration were developed and the form and limitation of such rules were investigated. The major results of this period were collected by A.H.Stroud[57] and published in the form of a book in 1971. This book has now become a standard reference on the subject of the approximate evaluation of multiple integrals. Since this period the subject has advanced more slowly and the theory has tended to be rounded out and various gaps filled in so that now there exists quite an extensive body of theory available for a wide range of regions of integration.

Considerably less attention, however, has been paid to the software or applied numerical quadrature side of the problem. Only a few important pieces of software have been published (although many more

may have been written). During the early 1970's both A.C.Genz and I.Robinson were actively working (independently) on the problem of producing quality software for numerical quadrature over rectangular regions. In September 1973 Robinson[52] completed a Ph.D. thesis on methods of numerical integration. The last chapter of which describes a general adaptive algorithm for integration over an n dimensional rectangular region. The algorithm was a simple generalization of the trapezium rule and is extremely time consuming but was one of the first algorithms of its type to appear in the literature.

In 1972 A.C.Genz[16] published a paper which described an adaptive multidimensional quadrature procedure for the hypercube. His procedure used two classical type rules and an extrapolation technique to improve the accuracy of the results. A few years later Genz prepared a modified version of this algorithm, which adopted Monte Carlo type methods, for the NAG Fortran library[17].

In 1976 Kahaner and Wells[25] were developing an algorithm for n dimensional adaptive quadrature using the simplex as a basic region of integration. This algorithm was written in a high level language and took advantage of advanced programming techniques. The method involved the derivation of variable order interpolatory quadrature formulae during the execution of the program. A paper relating to the algorithm was not published until 1979, and it is still in the form of an experimental test bed rather than a polished piece of library software.

Some major contributions to the area have been made by J.N.Lyness

[31,32,..40] particularly with respect to quadrature over the simplex. With reference to one dimensional integration Lyness[34] has suggested that too many automatic routines deny the user of the opportunity to think. He believes that the user should be encouraged to take a more active part in the choice of a suitable algorithm for the evaluation of his particular integral and that software should be written so as to be able to take advantage of any prior knowledge of the integrand. With the greater complexities of higher dimensional problems there is an even greater scope for savings from this kind of information and so these suggestions are even more important.

To summarise, the theory has reached a stage where any further research can only bring diminishing rewards but the application of the theory is still at an early stage of development.

1.4 Decisions taken as a result of the literature survey

The survey of the literature revealed the diversity of the topic and the author made some decisions as to the direction of the research from the very outset. Particularly with reference to the region of integration and the type of formulae adopted.

First consider the region of integration. In one dimension there are only three basic types of region of integration; a closed interval, a semi-closed interval and an open interval. Whereas in more than one dimension there are a potentially infinite number of different types of region to contend with and only in particular cases is it possible to transform formulae from one type of region onto a second

type of region. Hence, the theory of multidimensional quadrature has tended to be developed for specific regions of integration.
Obviously, this makes it very difficult to attempt to construct algorithms that will be applicable to completely general regions of integration and still be efficient and reliable. The author therefore chose to limit his work to two specific basic regions of integration; the hypercube and the simplex. These are the natural n dimensional extensions of the square and the triangle respectively. This was not an entirely arbitrary choice. These two regions are quite common in practical problems and a large body of theory has been based upon them. Further because of the properties of linearity for multiple integrals**, it is possible to form an approximation over a given region that can be defined as a combination of subregions by summing estimates over the subregions and both the hypercube and the simplex lend themselves to forming the subregions of other regions.
Now consider the type of formulae adopted. The previous section described the two paths taken as regards the derivation of formulae; the classical type methods and the Monte Carlo type methods. The author considered it impractical to follow both approaches and since the research was chosen to be restricted to specific regions this lead to the consideration of classical type methods only. Monte Carlo type methods have the advantage of being adaptable to various regions but offer no reasonable means of avoiding returns to the same neighbourhood sample which suggests that the methods should not be as effective as an efficient classical type method.

**{The properties of linearity for multiple integrals

Multiple integrals satisfy the following (which correspond to the

properties of linearity for one dimensional definite integrals) :

a) if k is a constant

$$\int \cdots \int_{Rn} kf(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

$$= k \int \cdots \int_{Rn} f(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

b) $$\int \cdots \int_{Rn} f(x_1, x_2, \ldots x_n) + g(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

$$= \int \cdots \int_{Rn} f(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

$$+ \int \cdots \int_{Rn} g(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

c) $$\int \cdots \int_{Rn} f(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

$$= \int \cdots \int_{Rn1} f(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

$$+ \int \cdots \int_{Rn2} f(x_1, x_2, \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

where Rn can be defined as a union of the regions Rn1 and Rn2 which have at most only boundary points in common.

These properties have natural extensions, for example in b) to a sum of more than two functions and in c) to a union of more than two regions.}

1.5 Objectives

The overall objective of this work has been to produce efficient,

reliable software for the approximate evaluation of multiple integrals over the two specific regions of the hypercube and the simplex. The work is limited to these two basic regions since this poses a sufficiently difficult task to be examined realistically in the time available. The software is not intended to provide the final solution to the problem but rather to form a basic grounding for the available theory which can be enhanced as additional theory is developed. A secondary aim is to demonstrate the usefulness of a high level language and advanced programming techniques in the construction of such numerical software.

A survey of techniques for evaluating one dimensional integrals revealed that the majority of one dimensional problems are tackled by adaptive quadrature algorithms. An adaptive algorithm attempts to take advantage of the "shape" of an integrand by applying less integrand evaluations where the integrand is "well behaved" and comparatively more integrand evaluations where it is "badly behaved". This can reduce the overall "cost", usually measured in terms of integrand evaluations, of achieving a given accuracy for some problems. That is problems which require comparatively more integrand evaluations in certain parts of the region to obtain a particular accuracy in those parts of the region as compared to the number required to obtain the same accuracy in the rest of the region. There is a potential saving for integrands which are "badly behaved" over relatively small areas of the region and which are "well behaved" over the rest. However, if the integrand is uniformly "well behaved" or uniformly "badly behaved" over the entire region then an adaptive method will perform less favourably than a non adaptive method. This is because an even distribution of integrand

evaluations will be required and the adaptive method involves various overheads in determining this whereas a non adaptive method always uses this type of distribution. In fact any "potential reduction" expected with an adaptive method has to be off set against the additional work required to achieve the desired distribution of integrand evaluations. The potential savings in "cost" by adopting this technique for multidimensional problems are far greater due to the size and complexity of such problems. Thus, one of the first objectives was to develop a basic adaptive multidimensional quadrature procedure. There are a wide variety of formulae that lend themselves to this type of algorithm and in order to be able to compare their merits a further aim was to make it a feature of the procedure that it was able to utilise alternative formulae. This allows any new formulae to be tested as they become available

A major feature of multidimensional integrals is the large number of integrand evaluations required to obtain a meaningful approximation. Even for a simple problem the number of integrand evaluations required can be very high and the "cost" increases exponentially as the number of dimensions increases. Further if the integrand is a complicated expression and hence, expensive to evaluate, then the time taken to compute the integrand evaluations becomes the dominant proportion of the time required to compute an estimate to the integral. If an adaptive scheme is used it seems feasible that for "expensive" integrands it would be economical to store and reuse these evaluations rather than recompute their values. Obviously the method of storing the integrand evaluations needs to be both fast and efficient in the use of store. The adoption of such methods

assumes that the quadrature formulae being used have their nodes at sufficiently convenient positions to allow a suitable subdivision strategy to take advantage of the positions in terms of reusing the integrand evaluations. Thus, the subdivision strategy and the chosen formulae are interrelated. These three topics, choice of formulae, subdivision strategy and methods of storing integrand evaluations, have been investigated by the author.

A straight forward approach to the design of an algorithm for the approximate evaluation of multiple integrals is one based on the use of product type formulae. However, at first sight such algorithms appear to be far too "expensive" in terms of numbers of integrand evaluations even though they can produce very accurate approximations in certain cases. With the advent of more and more powerful computers they could offer a feasible technique. Consequently the author has developed an algorithm based on product type formulae in order to quantitatively measure their effectiveness in terms of cost and accuracy and in order to provide a bench mark for comparing other techniques. This algorithm is non adaptive and uses an even distribution of nodes throughout the region of integration. Hence it is well suited to integrands that are uniformly behaved, either "well" or "badly", throughout the region.

Experience with one dimensional adaptive quadrature procedures (Malcolm and Bruce-Simpson [41]) suggest a global subinterval selection stategy as opposed to a local one. With a local strategy the local error criterion decreases linearly with the interval length and hence is most stringent as a tolerance in regions where the adaptive process is performing the most subdivisions. With a global

strategy the aim is to select subintervals so that the local errors are roughly equal in magnitude rather than scaled by the length of the subintervals. Malcolm and Bruce-Simpson suggest that a global subdivision strategy has the potential both for reducing the number of subintervals, and the corresponding integrand evaluations, and for generating a result with an error closer to a specified tolerance, rather than more accurate with the corresponding "cost" overheads. Since the number of subregions used in a multidimensional quadrature procedure is likely to be higher than that in a one dimensional problem the potential savings would seem to be far greater. Such considerations have been examined by the author to determine if the potential savings can be achieved or if they are lost in the overheads incurred in implementing such a strategy.

Once successful procedures have been written to approximate multiple integrals over the two basic regions the problem of extending these methods to cover other regions which are unions of the basic regions must be considered. The scope of this thesis cannot cover the topic of subdivision of regions of integration into combinations of hypercubes and simplexes adequately. However, the author has considered how the procedures can be extended or used to cope with problems where the region of integration is already expressed as a union of subregions, each of which is either a simplex or a hypercube.

Until recently the enhancement of computer performance has come from a refinement of the basic Von Neumann architecture and the improved performance of semiconductor components. With the rapid development of LSI technology and the corresponding fall in processor costs

there has been a trend towards multiprocessor architectures offering both parallel and concurrent processing capabilities. Improved performance of problem solving on this type of architecture depends to a large extent on the algorithm employed. If the algorithm can be divided easily into a number of largely independent processes then an improvement in terms of speed of execution can be expected. With an adaptive quadrature procedure the algorithm proceeds by continually subdividing the initial region of integration into more and more subregions each of which is treated in a similar manner on an independent basis. This suggests that an improved performance might be expected on a multiprocessor type computer. Although a machine with the necessary architecture was not at the author's disposal, the theoretical possibilities of this type of approach have been considered.

1.6 Contents of the chapters

Chapter 2 deals with the theory of product type rules. A little background theory relating to product rules is given. The major problem with product rules is the large number of integrand evaluations used, but if sufficient processing power is available they can be used to produce very accurate results for some problems. An algorithm is developed based on product Patterson rules. The algorithm adopts an iterative approach applying higher and higher order Patterson products until the required accuracy is obtained. A product Patterson set of rules was chosen because they form a common point family of rules and a simple method has been devised to store and reuse the integrand evaluations.

Chapter 3 examines the difficulties of testing algorithms. The methods of testing used with reference to one dimensional problems are reviewed and the applicability of these methods to multidimensional algorithms is considered. The ideas of the "battery" test and the performance profile are introduced.

The first part of Chapter 4 contains a detailed description of a basic adaptive multidimensional quadrature procedure for the hypercube. This procedure is written so as to facilitate the testing of various formulae on a variety of test problems. The second part of Chapter 4 describes an analogous procedure for the simplex. These procedures deliver satisfactory results but highlight the problems of testing.

In the first half of Chapter 5 the author approaches the problem of storing integrand evaluations. The justification for storing integrand evaluations is discussed. A method of storing the integrand evaluations in a linked list is introduced along with a method of determining a unique key for each integrand evaluation. Alternative methods of enumerating this key are described and an algorithm based on one method is described. The second half of Chapter 5 continues the theme of storing integrand evaluations by introducing the ideas of scatter storage techniques based on hash codes. An algorithm based on this method is developed, This offers the possibility of increased performance under certain conditions.

In Chapter 6 one of the previous algorithms is modified to use a global subinterval strategy instead of the local strategy used previously. A comparison is made between the two approaches and some

conclusions are reached about the advantages and disadvantages of the global technique.

Chapter 7 defines a method of describing a region as a linked list of subregions, each of which is either a hypercube or a simplex. A procedure is constructed which applies the basic procedures developed previously to a linked list of this form in order to obtain an approximation to an integral over the region thus described. The limitations of this approach are discussed.

The possible advantages of using a multiprocessor type architecture for multidimensional quadrature are considered in Chapter 8. A theoretical algorithm is developed on the basis of the availability of a multiprocessor with certain capabilities.

The final Chapter contains some concluding remarks and a summary of the preceding work.

Chapter 2 Product Formulae

2.1 Introduction

The major classification of quadrature formulae is into product and non product type formulae. This chapter deals with product formulae. A quadrature formula such as :

$$\int_{Rn}\cdots\int w\,(x_1 x_2 \ldots x_n)\; f(x_1 x_2 \ldots x_n)\; dx_1\, dx_2 \ldots dx_n$$

$$\approx \sum_{i=1}^{n} B_i f(v_{i,1} v_{i,2} \ldots v_{i,n}) \;.$$

could be derived from a combination, or product, of formulae for regions of dimension less than n. It is not possible to construct product formulae for arbitary regions Rn, but they can be constructed for some simple regions which are often encountered. The author considers product formulae in relation to two of these simple regions : the hypercube and the simplex.

In most cases n one dimensional formulae, each of degree d, are combined to give a new formula of degree d for Rn. The method for constructing product formulae is the method of separation of variables. Consider the monomial integral :

$$\int_{Rn}\cdots\int w\,(x_1 x_2 \ldots x_n)\; x_1^{\alpha_1} x_2^{\alpha_2} \ldots\; x_n^{\alpha_n}\, dx_1\, dx_2 \ldots dx_n \qquad \ldots(1)$$

If it is possible to find a (non linear) transformation :

$$x_1 = x_1(u_1, u_2, \ldots, u_n)$$

$$x_2 = x_2(u_1, u_2, \ldots, u_n)$$

.

.

.

$x_n = x_n(u_1, u_2, \ldots, u_n)$

which transforms (1) into the product of n single integrals :

$$\int w_1(u_1)\ g_1(u_1)\ du_1$$

$$\int w_2(u_2)\ g_2(u_2)\ du_2$$

.

.

.

$$\int w_n(u_n)\ g_n(u_n)\ du_n$$

and if some suitable formulae are known for the single integrals, then it is possible to combine these formulae to give a formula for Rn.

The most undesirable property of product formulae is that the number of points increases very rapidly as n increases. However, for small n product formulae can be very useful because of their high accuracy.

2.2 The construction of a product formula for the hypercube

Initially consider the case for n = 3. Let Rn be the hypercube $-1 \le x \le 1$ and the weight function w(x,y,z) be equal to 1. Now suppose there exists a one variable formula

$$\int_{-1}^{1} g(x)\,dx \approx \sum_{i=1}^{m} a_i g(\mu_i)$$

which has degree d.

Now construct the product formula from three copies of this formula, that is the formula with m^3 points :

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(x,y,z)\,dxdydz \approx \sum_{\substack{1 \le i_k \le m \\ k=1,2,3}} a_{i_1} a_{i_2} a_{i_3} f(\mu_{i_1} \mu_{i_2} \mu_{i_3}) \quad \ldots.(2)$$

It is not difficult to see that this formula is also of degree d. For if $0 <= \alpha <= d$, $0 <= \beta <= d$, and $0 <= \gamma <= d$ ....(3)

then

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} x^{\alpha} y^{\beta} z^{\gamma} dx\,dy\,dz = \int_{-1}^{1} x^{\alpha} dx \int_{-1}^{1} y^{\beta} dy \int_{-1}^{1} z^{\gamma} dz$$

$$= \sum_{i_1=1}^{m} a_{i_1} \mu_{i_1}^{\alpha} \sum_{i_1=2}^{m} a_{i_2} \mu_{i_2}^{\beta} \sum_{i_1=3}^{m} a_{i_3} \mu_{i_3}^{\gamma} \quad \ldots.(4)$$

$$= \sum_{\substack{1 \le i_k \le m \\ k=1,2,3}} a_{i_1} a_{i_2} a_{i_3} \mu_{i_1}^{\alpha} \mu_{i_2}^{\beta} \mu_{i_3}^{\gamma}$$

Since the set of all α, β, γ which satisfy $0 <= \beta + \gamma + \alpha <= d$, $0 <= \alpha$, $0 <= \beta$, $0 <= \gamma$ is a subset of those which satisfy (3), formula (2) is exact for all monomials of degree less than or equal to d. An argument similar to (4) shows that formula (2) is not exact for x^{d+1} , y^{d+1} and z^{d+1}. Therefore formula (2) has degree d.

Formula (2) has an obvious generalization for any n >= 2.

2.3 General Cartesian Product Formulae

The result of the last section can be generalized for other regions as follows. Assume Rn, Rp and Rq are regions in Euclidean space of dimensions n, p and q respectively, where n = p + q and

$$Rn = \{(x_1, x_2 \ldots x_n) : (x_1, \ldots x_p) \in Rp, (x_{p+1} \ldots x_n) \in Rq\} .$$

Then Rn is called the Cartesian product of Rp and Rq. That is

Rn = Rp * Rq.

Now assume that

$$w(x_1, x_2 \ldots x_n) = w_p(x_1, x_2 \ldots x_p) w_q(x_{p+1} \ldots x_n) \quad \ldots (5).$$

If there exist two formulae, one for Rp and one for Rq :

$$\int_{R_p} \cdots \int w_p(x_1 \ldots x_p) f(x_1 \ldots x_p)\, dx_1 dx_2 \ldots dx_p$$

$$\approx \sum_{i=1}^{N_p} B_{p,i} f(\lambda_{i,1} \ldots \lambda_{i,p}) \text{ of degree d } \ldots (6)$$

$$\int_{R_q} \cdots \int w_q(x_{p+1} \ldots x_n) g(x_{p+1} \ldots x_n)\, dx_{p+1} \ldots dx_n$$

$$\approx \sum_{i=1}^{N_q} B_{q,j} g(\mu_{j,p+1} \cdots \mu_{j,n}) \text{ of degree d } \ldots (7).$$

Then the $N = N_p N_q$ points and coefficients

$$(\lambda_{i,1} \cdots \lambda_{i,p}, \mu_{j,p+1} \cdots \mu_{j,n}) \; B_{p,i} B_{q,j} \quad \ldots (8)$$

$$i = 1, \cdots, N_p \qquad j = 1, \cdots, N_q$$

form an integration formula of degree d = min $(d_p d_q)$ for Rn with weight function (5).

The proof is a consequence of the fact that

$$\int_{R_n}\cdots\int w(x_1 \ldots x_n)\, x_1^{\alpha_1} x_2^{\alpha_2} \ldots . x_n^{\alpha_n}\, dx_1 \ldots dx_n$$

is the product of

$$\int_{R_p}\cdots\int w_p(x_1 \ldots x_p)\, x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_p^{\alpha_p}\, dx_1 \ldots dx_p$$

and

$$\int_{R_q}\cdots\int w_q(x_{p+1} \ldots x_n)\, x_{p+1}^{\alpha_{p+1}} x_{p+2}^{\alpha_{p+2}} \ldots x_n^{\alpha_n}\, dx_{p+1} \ldots dx_n$$

and is analogous to the set of equations (4).

The formula (8) is a Cartesian product formula; the Cartesian product of the formulae (6) and (7).

2.4 The construction of product formulae for the n simplex

Now consider the construction of a product formula for Tn the n simplex with vertices :

(0,0,.....0)

(1,0,.....0)

(0,1,0,...0)

.

.

.

(0,0,.....1).

It is possible to transform Tn onto any other simplex by means of an affine transformation. Therefore it is possible to obtain integration formulae for any given n simplex by an affine

transformation of formulae for Tn.

The integral of a monomial over Tn is

$$\int_0^1 \int_0^{1-x_1} \int_0^{1-x_1-x_2} \cdots \int_0^{1-x_1-\cdots-x_{n-1}} x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n} dx_n dx_{n-1} \ldots dx_1 \quad \ldots.(9).$$

This can be transformed into a product of n single integrals using the following transformation :

$$\begin{aligned}
x_1 &= y_1 && = y_1 \\
x_2 &= y_2(1-y_1) && = y_2(1-x_1) \\
x_3 &= y_3(1-y_2)(1-y_1) && = y_3(1-x_1-x_2) \\
&\cdot \\
&\cdot \\
&\cdot \\
x_n &= y_n(1-y_{n-1})(1-y_{n-2})\ldots(1-y_1) && = y_n(1-x_1-x_2- \ldots -x_{n-1}) \quad \ldots.(10)
\end{aligned}$$

Since the limits of integration for the x are

$0 <= x_i <= 1-x_1- \ldots -x_{i-1}, \quad i = 1,2,\ldots n$

the limits for the y_i will be

$0 <= y_i <= 1 \quad , i = 1,2,\ldots n.$

Since the Jacobian of transformation (10) is

$J = (1-y_1)^{n-1}(1-y_2)^{n-2} \ldots. (1-y_{n-1})$

the monomial integral (9) transforms into

$$\int_0^1 \cdots \int_0^1 (1-y_1)^{\beta_1} \ldots. (1-y_{n-1})^{\beta_{n-1}} y_1^{\alpha_1} \ldots y_n^{\alpha_n} dy_1 dy_2 \ldots dy_n \quad \ldots.(11)$$

$\beta_1 = \alpha_2 + \ldots. + \alpha_n + n-1$

$\beta_2 = \alpha_3 + \ldots. + \alpha_n + n-2$

$\cdot$

.

.

$\beta_{n-1} = \alpha_n + 1$

The integral (11) is a product of n single integrals, where the integral with respect to y_K has the form :

$$\int_0^1 (1-y_K)^{n-K} P_\alpha(y_K)\, dy_K \quad k = 1,2,\ldots n$$

where $P_\alpha(y_K) = y_K^{\alpha_K}(1-y_K)^{\alpha_{K+1}+\cdots+\alpha_n}$ is a polynomial of degree $\alpha = \alpha_K + \ldots\ldots + \alpha_n$ in y_K. Therefore if there exist n one variable formulae of degree d, of the form :

$$\int_0^1 (1-y_K)^{n-K} f(y_K)\, dy_K \approx \sum_{i=1}^{m} A_{K,i}\, f(\mu_{K,i}) \quad \ldots.(12)$$

for $k = 1,2,\ldots n$, these can be combined to give a formula of degree d for Tn.

These results can be summarised as follows: if each of the n formulae (12) has degree d then a formula of degree d for Tn, with $w(x_1,\ldots.x_n) = 1$, is given, in cartesian coordinates, by the m^n points and coefficients

$(\vartheta_{i_1}, \vartheta_{i_1 i_2}, \ldots.\ \vartheta_{i_1 i_2 \ldots i_n}) \quad \ldots.\ (13)$

$A_{1,i_1} A_{2,i_2} \ldots.\ A_{n,i_n}$

$\vartheta_{i_1} = \mu_{1,i_1}$

$\vartheta_{i_1 i_2} = \mu_{2,i_2}(1-\mu_{1,i_1})$

.

.

.

$$\nu_{i_1 i_2 \cdots i_n} = \mu_{n,i_n}(1-\mu_{n-1,i_{n-1}}) \cdots\cdots (1-\mu_{1,i_1})$$

$1 \Leftarrow i_k \Leftarrow m$, $k = 1,2,\ldots n$.

Formula (13) is called the conical product of the one dimensional formulae (12); these are usually taken to be the Gauss-Jacobi formulae.

The above can be generalized to give integration formulae for Tn with a weight function

$x_1^{\delta_1} x_2^{\delta_2} \cdots\cdots x_n^{\delta_n}(1-x_1)^{\epsilon_1} \cdots\cdots (1-x_1- \cdots -x_n)^{\epsilon_n}$.

The product formula is exactly analogous to (13) except in place of the one variable formulae (12) the following formulae must be used.

$$\int_0^1 (1-y_k)^{\beta_k} y_k^{\gamma_k} f(y_k)\, dy_k \approx \sum_{i=1}^{m} A_{k,i} f(\mu_{k,i}) \quad k = 1,2,\ldots n$$

where the $\beta_k, \gamma_k, \delta_k, \epsilon_k$ are related by

$-1 < \gamma_k = \delta_k \quad k = 1, \ldots n$

$-1 < \beta_1 = \delta_2 + \cdots + \delta_n + \epsilon_1 + \cdots + \epsilon_n + n - 1$

$-1 <$

$-1 <$

.

.

$-1 < \beta_{n-1} = \delta_n + \epsilon_{n-1} + \epsilon_n + 1$

$-1 < \beta_n = \epsilon_n$

2.5 The degree of the formulae used in the construction of a product formula

In the above discussion all the product formulae were constructed using products of formulae of the same degree. There is no reason why all the formulae used have to be of the same degree and under certain circumstances there may be advantages in using different degree formulae. For example if it is known that an integrand is "well" behaved in one dimension but "badly" behaved in another then it could be advantageous when forming a product rule for this integrand to use a product of a low and a high order formulae. Thus allowing the distribution of the nodes to reflect the behaviour of the integrand. On the other hand if the behaviour of the integrand is not known then a product of different degree may give a false impression of the integral, in particular if the integrand is well behaved in the dimension where a high order formula has been used and badly behaved in a dimension where a low order formula has been used. Hence the main advantage of using the same degree formulae when constructing a general product formula is that the resulting formula has an an even distribution of nodes, whereas the use of different degree formulae can be advantageous when constructing specific product formulae for integrands of known behaviour.

2.6 A set of product rules based upon Patterson's formulae

This section deals with the construction of a set of product type rules for the hypercube based upon the one dimensional formulae derived by Patterson [48]. Patterson's formulae form a family of interlacing whole interval, common point quadrature formulae of

fairly high order which possess good stability and convergence properties. The rules were produced by Patterson in 1968 as an example of the method which he developed to extend the ideas of Kronrod [28], who first showed how to add a further n+1 points to an n point Gauss-Legendre formula to produce a 2n+1 point formula of degree 3n+1 (n even) or 3n+2 (n odd) in 1965. Patterson began with a 3 point Gauss rule from which he derived a 7 point rule with 3 of the abscissae coinciding with the original Gauss abscissae; the remaining 4 were chosen so as to give the greatest possible increase in polynomial integrating degree and the resulting 7 point rule had degree 11. From the 7 point rule a 15 point rule of degree 23 was derived in a similar manner. Continuing in this fashion Patterson derived rules using 31, 63, 127 and 255 points of respective degree 47, 95, 191 and 383. The nodes and weights for these formulae are given in appendix [2]. These formulae were used in an algorithm for automatic numerical integration over a finite interval [48]. The basis for the algorithm was the successive application of these rules, until the most recent results differ by the tolerance or less. Due to their interlacing form, no integrand evaluations are wasted in passing from one rule to the next and the algorithm has proved to perform reliably and efficiently in it s long period of use. In a survey of available algorithms for numerical quadrature V.A.Dixon [11] states that the family of formulae is ideal for an automatic scheme. Hence it seemed reasonable to adopt these formulae as the starting point when constructing a set of product type rules to be used in an automatic quadrature routine based upon an iterative scheme.

2.7 Construction of a product type multidimensional quadrature routine.

This routine uses a simple iterative scheme, generating and applying successive higher order product rules, based on a Patterson family of one dimensional rules, until either the required accuracy is obtained or the maximum number of integrand evaluations allowed is exceeded. The method is based upon generating the rules in preference to storing them for two reasons; firstly, so that the method can be used for any number of dimensions, and secondly, so that only one copy of the nodes and weights has to be stored.

Basically the algorithm consists of the following :

REPEAT

generate the m^n point product rule based on the m point Patterson rule (m takes the values 3,7,15,31,63,127,255)

compute an estimate to the integral using this rule

UNTIL

either the estimate is to within the given tolerance or the maximum number of integrand evaluations is exceeded

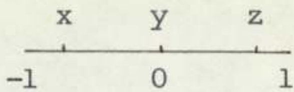
IF the maximum is exceeded

THEN output a suitable message

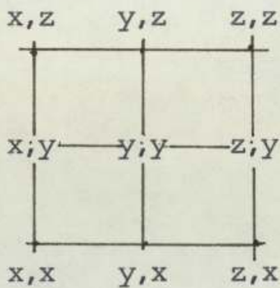
ELSE output the result

FI

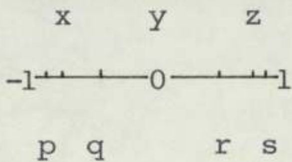
As an example consider a two dimensional problem in which case n=2. Starting with a three point rule (m=3) it is necessary to construct a 9 (m^n) point product rule. Let the 3 points be x, y, z. In one dimension the distribution of the points may be :



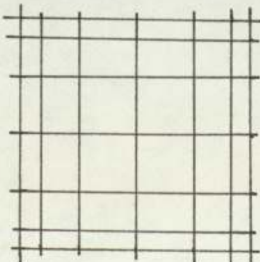
In order to form the 9 point rule 3 copies of the 3 point rule must be distributed across the region at a spacing equivalent to the distribution of the original points. This gives a distribution of :



The next rule has 7 points in one dimension, 3 of which are those of the 3 point rule and the other 4 of which interlace with these. Thus in one dimension :



In order to form the 49 point rule, 7 copies of the 7 point rule must be distributed across the region at a spacing equivalent to the distribution of the original points. Thus giving :



Where the points are (p,p),(p,x),(p,q),..(s,s). It can be seen that

the coordinates of the points of the product rule are constructed by taking all the possible nodes of the one dimensional rule as the first coordinate of a point and all the possible nodes of the one dimensional rule as the second coordinate of a point. Consider the 9 point rule :

the first point is (1st node of 1d rule, 1st node of 1d rule)

the second is (..   , 2nd node of 1d rule)

the third is (..   , 3rd node of 1d rule)

the fourth is (2nd node of 1d rule, 1st node of 1d rule)

....

....

the ninth is (3rd node of 1d rule, 3rd node of 1d rule).

A simple way of representing this is to let the nodes of the original one dimensional rule to be numbered 1, 2, 3. Then the nodes of the product rule can be written in terms of these numbers :

(1,1),(1,2),(1,3) ......(3,3).

Then a point in the product rule, for example (2,3), can be interpreted as the point given by taking the appropriate combination of the nodes of the one dimensional rule, for example the second node as the first coordinate and the third node as the second coordinate. Obviously the process can be reversed, in that the pairs (1,1),(1,2) ... (3,3) can be generated first and the actual points generated by relating the integers to the nodes of a one dimensional rule. This is the method adopted in the algorithm. The procedure to generate the nodes of the m^n point product rule has to work for any value of m or n given the appropriate set of nodes and weights for the one dimensional m point rule. The procedure achieves this by generating the n elements of an array, in the order (1,1,..1),(1,1,..2),...(m,m,..m), which can be interpreted as the

coordinates of a point of the new rule. For example, if n = 3 and the array holds (1,3,2) this can be interpreted as the point x,y,z where x is the first node of the one dimensional rule, y is the third node of the one dimensional rule and z is the second node of the rule. The weight for this node is given by the product of the first, third and second weights for the one dimensional rule. This procedure consists of the following :

```
PROC generate = (INT m,n, REF[]INT array,
                 REF REAL present estimate)VOID :
BEGIN
    {This procedure generates a set of m^n n dimensional points,
    where m is the number of points in a one dimensional rule,
    which make up an n dimensional product rule; the product of n m
    point one dimensional rules}
    FOR i TO m DO
    BEGIN
        array[n] := i ;
        IF n > 1
        THEN generate(m,n-1,array,present estimate)
        ELSE
            interpret the array ;
            increase the number of integrand evaluations;
            add the weight * the integrand evaluation to the
            present estimate
        FI
    END
END of the procedure generate.
```

One of the major advantages of using the Patterson family of rules

is that each higher order rule uses all the previous nodes used by earlier rules. This means that no integrand evaluations, which could be "expensive" for complicated integrands, need to be wasted during the iterative scheme. In order to take advantage of this feature it is necessary to store the integrand evaluations and be able to access them efficiently. Fortunately,using generate the nodes will always be generated in a specific order, even when increasing the value of m. In fact increasing the value of m merely adds new nodes to the sequence of nodes. For example, with a three dimensional problem the nodes would be generated in the following order:

1 , 1 , 1

1 , 1 , 2

1 , 1 , 3

1 , 2 , 1

1 , 2 , 2

...

... etc.

Increasing the value of m would result in the addition of more nodes to the list :

1 , 1 , 1

1 , 1 , 2

1 , 1 , 3

1 , 1 , 4 new node

1 , 1 , 5 new node

1 , 1 , 6 new node

1 , 2 , 1

1 , 2 , 2

...

... etc.

Also, in generating this new list it is easy to see that any nodes generated using values less than or equal to the previous value of m already exist, while nodes generated from values greater than this value of m do not exist. Consequently the integrand evaluations at these nodes need to be computed and stored. Thus the integrand evaluations are conveniently stored in a linked list. A pointer then advances through the list as nodes are generated. If the node already exists then the pointer indicates the element of the list which holds the required integrand evaluation. Otherwise, the pointer indicates the position of insertion of a new list element created to hold the integrand evaluation at this node.

feval for 1,1,1

feval for 1,1,2

feval for 1,1,3

feval for 1,2,1

feval for 1,2,2

feval for 1,2,3

etc.

When m increases these integrand evaluations are used in order until the node 1,1,4 is generated which does not already exist. A new node element is created and the integrand evaluation is added to the list.

feval for 1,1,4

Hence it was necessary to alter the procedure generate slightly as follows:

```
PROC generate =(INT m,n,previousm, REF[]INT array, BOOL exist,
                REF REAL present estimate) VOID :
BEGIN
```

```
{Procedure to generate the nodes of an n dimensional product
 type rule from an m point one dimensional rule}
FOR i TO m DO
BEGIN
    array[i] := i ;
    IF exist
     {if exist is false then the node already contains a
     coordinate which indicates that it does not already exist}
    THEN IF i > previousm
         THEN {node does not already exist}
              exist := FALSE
         FI
    FI ;
    IF n > 1
    THEN generate (m,n-1,previousm,array,exist,
                   present estimate)
    ELSE IF exist
         THEN
     {the integrand has been evaluated at this node previously
     and the required integrand evaluation is the next value in
     the list}
              select the next value in the list ;
              generate the corresponding weight ;
              add the weight multiplied by
              the value to the present estimate
         ELSE {the node has not been used previously}
              interpret the array;
              increase the number of integrand evaluations ;
              create a new list element ;
```

```
                store the integrand evaluation in this element ;
                add this element to the list ;
                add the weight * the integrand evaluation
                to the present estimate
            FI
        FI
    END
END of the procedure generate.
```

This procedure now contains several steps which need to be explained further.

Select the next value in the list :

In order to be able to do this it is necessary to define the elements of the list. Each element needs to contain both an integrand evaluation and a pointer to the next element in the list. It was convenient to define a new mode for the elements of the list :

```
    MODE NODE = STRUCT (REAL feval, REF NODE ptr) .
```

Then selecting the next value in the list required the use of a pointer to indicate the present position in the list. This pointer was called pointer so that the following could be written:

```
    next value := feval of pointer ;
   {extract the value from the list}
    pointer := ptr OF pointer
   {move the pointer on to the next item in the list }
```

Generate the corresponding weight:

This is a little more involved. The weights for the various rules are stored in an array called weights in the required order. That is:

weights for m = 3

.

.

weights for m = 7

.

.

.

weights for m = 15

.

.

etc.

Also a second array called starting points contains the positions of the start of each set of weights. Thus the starting position is merely a function of m and generate the corresponding weight consists of:

```
corresponding weight := 1 ;
FOR i TO number of dimensions
DO corresponding weight TIMES
       weights(starting position[m]+array[i])
```

The value of starting position[m] is another parameter of the procedure generate.

Interpret the array :

This consists of converting the values of the array into the coordinates of the n dimensional point. The point is declared as [1:n]REAL and interpret consists of :

```
FOR i TO number of dimensions
DO point[i] := node[array[i]]
```

This was combined with generating the weight described above.

Create a new list element, store the integrand evaluation and add

this element to the list are achieved by the use of a procedure which will perform all three tasks, given the appropriate parameters. The procedure is called add to list and the parameters it requires are the integrand evaluation and the pointer to the present position in the list.

```
PROC add to list = (REAL feval, REF REF NODE pointer) VOID :
BEGIN
    REF NODE newnode = NODE ; {create a new node on the heap}
    feval OF newnode := feval ;
    ptr OF newnode := pointer ;{pointer dereferenced twice}
    pointer := newnode          {pointer dereferenced once}
END of the procedure add to list.
```

Then it is necessary to move the pointer on to the next item in the list: pointer := ptr OF pointer. This ensures that the next integrand evaluation chosen from the list is the correct one for the given node.

Hence the outline of the recursive procedure generate consists of:

```
PROC generate = (INT m,n,previousm,starting position,
                 REF[]INT array,
                 BOOL exist, REF REAL present estimate)VOID:
BEGIN
{This is a procedure to generate the nodes of an n dimensional
product type rule from an m point one dimensional rule and to
generate an estimate to an integral using this rule}
FOR i TO m DO
BEGIN
    array[n] := i ;
    IF exist
    THEN IF i > previousm
```

```
    THEN exist := FALSE
    FI
FI;
IF n> 1
THEN generate (m,n-1,previousm,starting position,array,
              exist,present estimate)
ELSE IF exist
    THEN
          {the integrand has been evaluated at this node,the
          required integrand evaluation is the next value in
          the list
          generate the corresponding weight}
         corresponding weight := 1;
         FOR i TO number of dimensions
         DO corresponding weight TIMES
              weights[starting position + array[i]] ;
         {add the weight * the next value to the estimate}
         present estimate PLUS (corresponding *
              feval OF pointer) ;
         {move the pointer to the next item in the list}
         pointer := ptr OF pointer
    ELSE {the node has not already been used}
         {interpret the array and generate the weight}
         corresponding weight := 1;
         FOR i TO number of dimensions DO
         BEGIN
            point[i] := node[array[i]] ;
            corresponding weight TIMES
             weights[starting position + array[i]]
```

```
            END ;
            integrand eval := f(point) ;
            present estimate PLUS
               (corresponding weight * integrand eval) ;
            nofe PLUS 1 ;
            {increase the number of integrand evaluations}
            add to list(integrand eval, pointer)
         FI
      FI
   END
END of the procedure generate.
```

The outline of the program based on Patterson's rules consists of :

```
Product type method based on Patterson's rules
WITH segfl-2d FROM pjk-alb-al
BEGIN
   [] REAL nodes = (.......) ;
   [] REAL weights = (.......) ;
   [] INT starting positions = (......) ;
   INT maxpoints , nofe := 0 , n ;
   REAL eps ;
   read((maxpoints,eps,n)) ;
   []INT m = (3,7,15) ;
   INT next := 1 ;
   [1:n]INT array ;
   BOOL exist := FALSE , notgt8 := TRUE , nottoomanyfe := TRUE ;
   REAL result , result1 ;
```

```
MODE NODE = STRUCT(REAL feval,REF NODE ptr) ;


PROC add to list = (REAL feval , REF REF NODE pointer) VOID :

BEGIN .......END;


NODE start := (0.0,NIL) ;

REF NODE pointer := ptr OF start ;


PROC generate = ....... :

BEGIN ..........END ;


generate(m[next],n,0,starting

position[next],array,exist,result) ;


WHILE (exist := TRUE ;

       pointer := ptr OF start ;

       generate ( ...........,result1) ;

       result1 - result > eps)

      AND notgt8 := next < 8

      AND nottoomanyfe := nofe < max

DO result := result1 ;


IF NOT notgt8

THEN print((newline,"all nodes used ",newline))

FI ;


IF NOT nottoomanyfe

THEN print((newline,"Too many integrand evaluations required"))

FI ;
```

```
    print((newline,"The result is : ",resultl,newline))
END
FINISH
```

The complete version of the program is given in appendix [2].

2.8 Testing

The program was tested with the set of test problems given in appendix [1] and using a set of tolerances 0.5, 0.1, 0.01, 0.05,....0.000001. The tables of the results for the test runs are given in appendix [8].

2.9 Conclusions

All the results produced using this program on the limited set of test problems were very accurate but correspondingly "expensive" in terms of time and the number of integrand evaluations used. The results were all far more accurate than the requested tolerance. For example on the two dimensional version of the first test problem with a tolerance of 0.5 the actual error was approximately 0.000001. However the number of integrand evaluations used was 49 as compared to 7 with one of the adaptive methods which produced a result with an actual error of approximately 0.003, and the time taken to compute the result was 77 millunits as compared to 9 millunits for the adaptive method. One of the drawbacks with using product formulae is the large minimum number of integrand evaluations that have to be used; in two dimensions a minimum of 49 integrand

evaluations are used, in three dimensions 343 and in n dimensions 7^n. Also there is a dramatic increase in the number of integrand evaluations used in moving from one product rule to the next higher order rule. For example in two dimensions the number of integrand evaluations used takes the values 49, 225, 961, 3969, 16129 and 165025, and in three dimensions it takes the values 343, 3375, 29791, 250047, 2048383 and 16581375. As the number of dimensions increases the change becomes even more pronounced. Hence, as can be seen from the results, the program stops for quite large tolerances, even though the previous results were far more accurate than the requested tolerance, because the next rule has had to be applied in order to determine the error estimate and either there is insufficient space left on the heap to store the list of integrand evaluations or the time allocation has been exhausted. This suggests that the stopping criterion might be relaxed somewhat.

In conclusion the method is very accurate but expensive both in terms of integrand evaluations used and the storage space required. The storage space required could be reduced to a minimum by not reusing the integrand evaluations but this would defeat the aim of the algorithm and slow the method down considerably with anything but the simplest of problems. Hence if sufficient storage space and time are available then the method is suitable for producing very accurate results for low dimensional problems. At this point in time the method is unsuitable for higher dimensional problems because the vast amount of space and the large number of integrand evaluations involved are beyond the computing power of most available machines.

Chapter 3 The Testing of Quadrature Routines

3.1 Introduction

This chapter deals with the testing and comparison of quadrature procedures. Obviously, it is necessary to test and to compare routines in order to satisfy the authors of the procedures that they actually work, in order to give the user some confidence in using the procedures and in order to justify the inclusion of such procedures in a library. It is generally agreed that the structure of automatic quadrature routines is sufficiently complicated to preclude the possibility of comparison or evaluation by analytic means alone and so it is necessary to adopt numerical experiments as a method of testing and comparison. The choice of a suitable format for these experiments is very important.

From the great number of quadrature routines which have been written for one dimensional problems it would seem that the inclusion of any reasonable new automatic quadrature routine in a software library can be justified by choosing a suitable set of test problems for which the routine produces "better" results than other available routines. A proliferation of automatic quadrature routines has resulted because of the "absence of generally acceptable standards or benchmarks for comparing or evaluating such routines" (Lyness and Kaganove[38]). This would suggest that it is very important to adopt a suitable test and comparison methodology for multidimensional quadrature routines so as to avoid wasting both time and computing resources. Hence the author examined the test methodologies adopted for one dimensional routines before adopting any particular approach

in this work. In one dimensional quadrature two distinct approaches have been developed. The first involves the use of a "battery" type test while the second is based upon a "statistical" or "perfomance profile" approach.

3.2 "Battery" type testing

The most commonly adopted approach in testing one dimensional quadrature routines is the "battery" test. This type of test involves applying a given routine to a predefined set of problems which have known solutions and which vary in difficulty from "well" behaved integrands with no non mathematical difficulties, i.e. difficulties due to the "shape" of the integrand, to "badly" behaved integrands with non mathematical difficulties. Theoretically, it is possible to take the results and compute an overall figure of merit for the given quadrature routine. This method of testing and comparison is based upon two assumptions :

a) That there is a "best" routine applicable to all the problems in the test set.

and

b) That the test set of problems are representative of a wide set of problems for which the routine will perform similarly.

These assumptions are not necessarily valid and this has led to several difficulties in applying this method. Nevertheless, the method is widely adopted because it is so simple to use. One major example of a battery type test was the investigation by Kahaner[24] which was performed in order to choose suitable one dimensional quadrature routines for inclusion in a subroutine library. This investigation highlighted some of the problems associated with

battery tests. A large set of problems, methods, and tolerances were used and the experiment was completely objective in that if there had been a best overall method then it would have been found. Unfortunately, no one method proved consistently better than any other over the complete range of test problems and in the end the choice was made based, principally but not exclusively, upon an experienced but subjective idea of the "best" all rounder in terms of average reliability and average speed for each routine. Using this test method it is very difficult to make any hard and fast decisions as to the general applicability of a routine.

Perhaps the major drawback of the battery test is the way in which the test set is chosen. The integrand functions are normally chosen to be as different as possible so as to obtain a wide generality. However, it does not always follow that a method suitable for a problem A will be suitable for a problem A' (a slight variant on A). In fact minor changes in the choice of integrand may lead to major differences in the performance of the algorithm in some cases (this is a consequence of the nature of the performance profile [39] which is discussed in the next section). Therefore, it is possible for a set of test problems to be either a "lucky" or an "unlucky" choice and to give a false impression, good or bad, of a routine.

3.3 The Performance Profile approach to testing

The performance profile approach to testing adopts the function $v(\text{Equad}(s, \varepsilon\,\text{req}))$ as a means of testing and comparison. The function $v(\text{Equad}(s, \varepsilon\,\text{req}))$ is the average number of integrand evaluations required by the routine to integrate members of a specific problem

family when the quadrature tolerance parameter has been set in such a way that an accuracy εreq is obtained with probability s.

A problem family is chosen such that each of it s members has a particular attribute. An individual member is specified by assigning a numerical value to an additional parameter λ, which may appear in the integrand function $f(x;\lambda)$ or in the integration limits $a(\lambda)$ and $b(\lambda)$. The parameter λ may take any value within a specified range, that is $\lambda- \le \lambda \le \lambda+$.

For example :

$$a = 1 \ , \ b = 2$$

$$f(x;\lambda) = ((x-\lambda)^2 + \mu^2)^{-1} \ , \ \mu = 0.01$$

$$0.998 \le \lambda \le 2.02$$

Each member of this family has a peak of height 100 and a half width 0.01 within or very close to the end of the integration interval.

If the experiments are limited to a single problem family then each run may be specified by two input parameters λ and εquad. Corresponding to each such pair it is possible to define

εact(λ;εquad) the error $|If-Qf|$ of the result returned by the routine and v(λ;εquad) the number of integrand evaluations required by the routine to return this result. A plot of the function εact(λ; εquad) against λ for a fixed value of εquad is called a performance profile. A fundamental property of quadrature routines is that the functions εact(λ;εquad) and v(λ;εquad) are rapidly varying discontinous functions of λ. Because of this they are not suitable as a direct measure of the efficiency af an automatic quadrature routine. The difficulties encountered in using battery testing result from this property since by relying on individual values of the input parameter λ and εquad significant arbitary components have

been introduced into the results which frustrate the evaluation process.

The evaluation technique proposed by Lyness and Kaganove [38] for one dimensional routines is based upon the performance profile but treats the problem family as a whole. An average function value count, $v(\varepsilon quad)$, is used along with the distribution function $\phi(x;\varepsilon quad)$.

$$v(\varepsilon quad) = 1/(\lambda_+ - \lambda_-) \int_{\lambda_+}^{\lambda_-} v(\lambda;\varepsilon quad)d\lambda \quad \ldots. (1)$$

$$\phi(x;\varepsilon quad) = (\text{proportion of values of } \lambda \text{ for which } |\varepsilon act(\lambda;\varepsilon quad)| <= x)$$

$$= 1/(\lambda_- - \lambda_+) \int_{\lambda_-}^{\lambda_+} H(x- |\varepsilon act(\varepsilon quad;\lambda)|)d\lambda \ldots. \quad (2)$$

where H(t) stands for the unit step function (Heaviside function)

$$H(t) = \begin{cases} 1 & t>0 \\ 1/2 & t=0 \\ 0 & t<0 \end{cases} \quad \ldots\ldots. (3)$$

These quantities can be calculated using Monte Carlo integration to approximate the integrals in (1) and (2). Thus m runs are made to obtain a set of results:

$\varepsilon act(\lambda i;\varepsilon quad)$; $v(\lambda i;\varepsilon quad)$ $i = 1,2,\ldots m$. The values of λ are chosen from the range $(\lambda- , \lambda+)$ using a (repeatable) random number generator and the quantities

$$Vm(\varepsilon quad) = 1/m \sum_{i=1}^{m} v(\lambda i;\varepsilon quad) \quad \ldots\ldots.(4)$$

and

$$\phi m(x;\varepsilon quad) = 1/m \ (\text{number of values of i for which } |\varepsilon act(\lambda i;\varepsilon quad)| <= x)$$

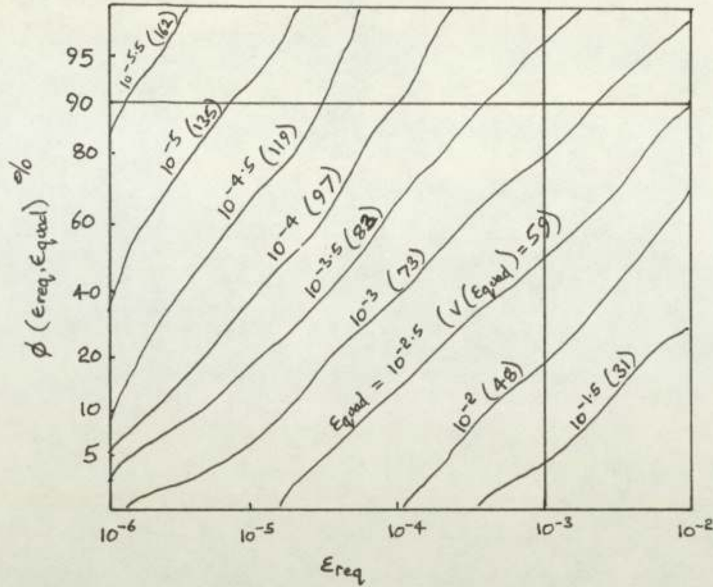
$$= 1/m \sum_{i=1}^{m} H(x- |\varepsilon act(\lambda i;\varepsilon quad)|) \ \ldots.(5)$$

are used as approximations to (1) and (2) respectively.

As in any statistically based experiment the size of the sample has to be chosen with care with a view to the accuracy required in the results. One advantage of this method is that if anybody doubts the results then the distribution function can be recomputed and the conclusions altered if significant differences are found. Once a problem family has been defined and a quadrature routine chosen, along with a value of εquad, then the functions ϕ(x;εquad) and v(εquad) are well defined and can be determined. As illustrated above for example.

In practice Lyness and Kaganove [38] found that relatively small values of m, such as m=100, were sufficient to obtain a clear idea of the form of the functions. However, they actually used values of m=1000 for the sake of being cautious and they produced values of ϕ(x;εquad) to within 1% for most of the range of x.

A set of statistical distribution functions corresponding to different problem families and different automatic quadrature routines provides a wealth of information which could be examined by experts with a view to determining defects or advantages of particular routines in various contexts. The following is a hypothetical set of statistical distribution functions, ϕ(εreq, εquad), plotted as a function of εreq for a given problem family and a specific routine.
Each curve is labelled with the value of εquad and in parentheses the value v(εquad). The ordinate is not linear but scaled in such a way that if log ε req were normally distributed the curve would appear to be a straight line.



From this table it can be seen that if , for example, a value of εquad of 10^-3 is chosen then there is a 79.7% probability of the result being accurate to the required tolerance with a corresponding number of integrand evaluations of 73. This could be compared with the same information from the curves for another routine in order to choose between the two. Sets of tables like this for different routines could be used to compare such things as the reliability of stopping criterion and the "cost" in terms of integrand evaluations for the various routines. This information could then be used either to alter existing methods or as a basis for creating new ones. An alternative application is to use them to provide a non expert, ie the user, with information that he might require for his particular problem.

Although it is unlikely that a user will have a problem which coincides precisely with a member of a problem family which has been investigated already, it is possible, with most difficult problems, to find a salient feature of the integrand which is responsible primarily for the difficulties to be encountered in numerical integration. A problem family with only this feature can be looked upon for guidance. Either these sta tistics would be available

already or they could be specially obtained.

How does a user choose a method and a tolerance? Suppose the problem has a dominant feature which corresponds to a particular problem family and that a set of distribution functions for that problem family and the available routines exists. Then, using the functions the user can determine, using a simple double interpolation process on the statistical distribution curves, the required tolerance εquad to achieve say a 90% success probability and the average cost, v(ε quad), using each method. For example consider the table given previously. If a horizontal line is drawn across at the 90% mark and a vertical line drawn at 10^-3 the intersection can be used to determine the value of εquad required to satisfy the user's conditions. In this case the intersection falls between a choice of 10^-3and 10^-3.5 for εquad with corresponding costs of 83 and 73 integrand evaluations. Hence the user may decide to use a value of 10^-3.3 in which case he would expect a 90% success probability at a cost of 78 integrand evaluations. Hence, the user can decide which method to adopt. Thus the user has to decide both the accuracy he requires and the probability of success that he is prepared to pay for in terms of integrand evaluations. The user is warned unambiguously that the routine may fail, in fact that statistically it will fail.

The process by which **E**quad(s,εreq) and v(εquad) are obtained from the statistical distribution function is a standard procedure involving interpolation. A prospective user need not be burdened with this calculation because a plot of **E**quad(s,εreq) and v(**E**quad(s, εreq)) which is sufficient for the user can be obtained

automatically. A user only needs to glance at these plots in order to obtain a clear idea of the relative cost involved in using any of the routines for a particular confidence level s.

A drawback of this method would appear to be the fact that a number of plots, one for each value of s, are required for every method. If this were the case then the technique would be of no practical use. Fortunately, (Lyness and Kaganove[38]), there is practically no qualitative difference between plots for different values of s, apart from the obvious point that more integrand values are required by each routine for a higher confidence level.

Thus the method of testing and evaluation involves using the quantity $v(\mathbf{E}quad(s, \epsilon req))$ where $s = \phi(\epsilon req, \mathbf{E}quad)$ as a measure of the cost of using a routine.

The method has several advantages:

1.The quantities on which the decisions are based are mathematically defined and can be recalculated. It is a repeatable experiment.

2.Once a problem family has been selected, there is no bias in the treatment.

3. The results are realistic in the sense that they relate to a 'likelihood of failure'. There is no implication that the routine can or should be completely reliable.

4. The results are problem orientated, that is they are in a convenient form for one to select an appropriate routine for a particular problem.

5.Lyness and Kaganove found that their conclusions were compatible with common experience.

6. It is possible to add routines and problem families and so build on currently available results.

The method has the following disadvantages:

1. It is possible to 'rig' a routine for any given problem family.

2. The choice of problem families is a subjective element.

3. It is a relatively expensive procedure.

4. To obtain full benefit, the user has to "tune" the value of εquad.

5. Only accuracy and economy are tested; the user interface, warning messages etc., are disregarded.

Hence, using a performance profile approach to testing it is possible to reach unambiguous if limited conclusions about the quadrature routines under test: routine A is better on average than routine B for problems with a particular salient feature. This method is very costly and not feasible for all situations.

3.4 Comparison between these two approaches to testing

The battery test sets out to demonstrate that a particular integration routine is "better" for a wide class of problems than certain other routines by applying the routines to a limited set of problems and making general assumptions from the results. Unfortunately, due to the very nature of the problems under test, these generalisations do not always follow and consequently the assumptions can sometimes give a false impression,either good or bad, of the routines being compared. On the other hand the

performance profile approach to testing involves considerably more work to reach far more limited conclusions, but the conclusions are unambiguous and appear to give a true impression of the routines under test.

3.5 Possible approach to testing multidimensional quadrature routines

The range of multidimensional integration problems is vast and the possible complexities are far greater than those in one dimensional problems. Since no one integration routine has proved to perform consistently better than all other routines over all problem ranges in one dimensional integration, it is reasonable to assume that no one routine will prove to be consistently better than all other routines over all the problem ranges in multidimensional quadrature. Hence, because of the advantages of the performance profile approach to testing over the battery type test, in that the former allows unambiguous conclusions to be reached for routines over specific problem ranges, it would seem natural to adopt a similar approach in testing multidimensional quadrature routines.

The main advantage of this approach to testing is that it enables the author of a given routine to say with confidence that his routine is suitable for any problems whose dominant feature is one of those featured in the set of problem families to which the routine is applicable. However, that being the case, there is still a subjective element in the testing in that the author of the routine chooses the problem families for the test runs. Certain aspects of the testing must be considered before deciding on the

approach as a suitable basis.

The method is very expensive if sufficient tests are carried out to give meaningful results. The choice of problem families is very important. Unless the problem families reflect the dominant features that are encountered in user problems then the results are of no use. The problem families should be user orientated, with different sets aimed at different applications. The method is quite complex and could involve the would be user in a lot of effort in actually interpreting the available results. This manual approach might be too much effort for the user to bother and therefore it would be better to automate the process but make all the information available should anyone require it, for example algorithm writers who may use the information to improve existing routines or in the writing of new routines.

Hence, although the method has many advantages it cannot be undertaken seriously unless sufficient people, both software writers and software users, agree upon it as a standard so that a single body of information can be built up which can grow as new routines are written. The salient features of the problem families must be chosen with great care and agreed upon. This in itself could involve a large amount of research but would be worth the effort if it avoided the waste of effort that has been seen with respect to one dimensional problems.

3.6 The method of testing adopted in this research

Although performance profile testing is recognised as the best

method of testing and comparing routines it was not feasible to adopt that approach in this work firstly because of the reasons outlined above and secondly because of the cost involved, particularly in terms of computing time. The method actually used was based upon a "battery" type test. It must be stressed that this approach was only used to form some general idea of whether or not the methods examined in the routines are of any use at all even though the results cannot guarantee to give unambiguous conclusions. If the routines perform badly on all the test problems then it can be assumed that they are not suitable as a starting point for multidimensional quadrature and alternative approaches can be adopted. However, since there are so few routines available any that prove to perform reasonably well over a limited "battery" test may be of use to somebody. It can only be stressed that no inference as regards the performance of the routines for other, even similar, problems to those used in the tests can be drawn from the results unless the user is satisfied that the salient feature of the problem in question is the same as the salient feature of one of the test problems and that the performance profile for the problem family to which they both thus belong is well behaved.

3.7 The set of test problems used

The set of test problems ranges from two to four dimensional problems, each of which has a known solution. The problems were chosen so as to illustrate different types of behaviour in the integrand over the region of integration. For example one has a difficulty along one boundary of the region, another has difficulties along two boundaries and a third has a non mathematical

difficulty in the centre of the region. All the test problems were written in the form of procedures which could be called by the various programs in order to evaluate the integrand at a particular node. Each procedure is included in one program segment and full details of the format of both the procedures and the segments are given in appendix [1] along with full details of all the test problems.

Chapter 4 A basic adaptive multidimensional quadrature procedure.

4.1 introduction

This chapter discusses a basic adaptive multidimensional quadrature procedure. The aim of the procedure is to compute an approximation to a multiple integral over a given region, to a given tolerance. The tolerance is supplied by the user. The basic algorithm is a more sophisticated version of an algorithm developed by the author for an M.Sc. project [27]. The basic algorithm formed the starting point of this research into multidimensional quadrature and, consequently, it was written as a research tool rather than as a complete and finished algorithm suitable for inclusion in a software library. Two versions of the procedure have been written; the first using the hypercube as a basic region and the second using the simplex as a basic region, but both follow the overall structure of the algorithm.

4.2 A brief description of the basic approach.

The following is only a brief summary of the more important aspects of the basic approach. The method is based upon a technique used in one dimensional integration and for simplicity the technique as used in one dimensional problems is described before the n dimensional analogy is discussed.

Given a one dimensional problem, a simple approach to find an approximation, S0, to it s evaluation is to apply a basic rule over the entire integration region. However, it is not usually feasible

just to accept this result without any indication of it s accuracy. Therefore, since most rules do not supply an accurate error estimate, it is necessary to compute an error estimate. A convenient approach is to subdivide the region into two or more, usually equal, parts and apply the same basic rule, suitably transformed, to each subregion. Then the sum of the two estimates gives a second approximation to the result, S1, which can be compared with S0 for consistency and in order to generate an error estimate. If the results are inconsistent with some required tolerance then each of the subregions can be subdivided and the rule applied in a similar manner to each new subregion. The sum of the results gives the next estimate, S2, which can be compared with S1 for consistency. Obviously, the process can be repeated to give a sequence of converging approximations to the integral, S0,S1,S2....Sn. This method has the disadvantage that the error at level P is given by considering the difference between the estimate Sp and the estimate at the next level Sp+1. That is, by the very nature of the error estimate it is necessary to go one level further than the accuracy actually required.

An alternative approach to obtaining an error estimate is to use two basic rules to give approximations A0B0, A1B1,...AnBn. Then the difference between the results Ai and Bi in any given region (or subregion) gives an error estimate over that region and the sum of the error estimates can be tested for convergence. This method is not as reliable as the previous one and the two basic rules need to be chosen with care. However, this method is often more convenient, particularly when an adaptive algorithm is being written. With an adaptive method a subregion is dismissed from further consideration

once an estimate has been formed over that subregion which satisfies the allowable tolerance for that subregion. If the first approach is taken then it is necessary to go to the next level of subdivision in order to determine the error estimate for a subregion whereas with this approach the error estimate is given quite simply by forming the difference between the two esimates Ai and Bi. The allowable tolerance for a subregion may be given by dividing the total required tolerance between the number of subregions. Using an adaptive method has the effect of concentrating more nodes in the regions where the integral is comparatively "badly" behaved and less where it is "well" behaved. However, the error method described and the method of subdivision of the tolerance has the disadvantage of imposing the strictest tolerances in the subregions where the function is least well behaved. This often results in the answer being more accurate than the user requested, with the overheads of a higher cost to the user. A slight improvement on the situation can be made by taking advantage of any "spare tolerance" from converged subregions. Instead of dividing the tolerance between all the subregions the sum of the error estimates for the converged subregions is subtracted from the total tolerance requirement and the remaining tolerance is divided between the non converged subregions. This should have the effect of producing a result closer to the required accuracy, with a corresponding saving in computing effort. This is the approach adopted with the multidimensional quadrature procedures, since the number of subregions is likely to be far higher and the potential savings far greater.

Now consider the n dimensional analogy of this technique as applied to a hypercube. Initially, two basic rules are applied over the

hypercube to give two estimates A0 and B0. These are compared for consistency with the error tolerance requested and if the error is too large then the hypercube is subdivided into 2^n (where n is the number of dimensions of the problem) subregions, each of which is a further hypercube. The basic rules, suitably transformed, are then applied to each of the subregions and the sum of these estimates gives the next estimate over the initial hypercube. If at any stage the difference between two estimates over a subregion is less than the present tolerance for that subregion then the subregion is dismissed from further consideration and the spare tolerance is shared amongst the other remaining subregions. A running total is kept of the estimates over the converged subregions. Hence there are two reasons for stopping; either the difference between two estimates over the whole hypercube is less than the required tolerance or convergence has been achieved in all the subregions. The problems of organisation are far more difficult with the n dimensional version of the technique than with the one dimensional version.

4.3 The structure of the basic algorithm

The essence of the algorithm is straightforward and consists of continually subdividing the region of integration into more and more subregions and forming new estimates to the result until convergence is achieved. However, the problems arise in keeping track of the subregions and applying the correct transformation of the basic rules to them.

Perhaps the simplest way to keep track of the subregions is to store

them in a linked list. Initially the list consists of the whole region only, then this is replaced by the list of it s subregions, then this list is replaced by the list of subregions of the subregions, and so on the list grows. As convergence is achieved in various subregions they are removed from the list altogether and their estimates added to the total of contributions from the converged subregions which is part of the final approximation. Thus one reason for terminating the algorithm is that the list of non-converged subregions is empty, indicating convergence in all subregions.

Hence the basic structure of the algorithm using a linked list approach consists of :

```
WHILE NOT converged
DO
    {Compute the estimate of each subregion in the linked list in
    turn}
    IF convergence is not achieved in a subregion
    THEN
        subdivide that subregion and store it s subregions on the
        next linked list of subregions
    ELSE
        dismiss this subregion from further consideration and add
        it s  estimate to the total estimate from converged
        subregions
    FI
    Compare the sum of the estimates from non-converged subregions
    plus the total estimate from converged subregions (i.e. the
```

present approximation to the integral) with the previous approximation.

IF the difference is less than

the required tolerance

OR

the new list of non-converged

subregions is empty

THEN

the method has converged

ELSE

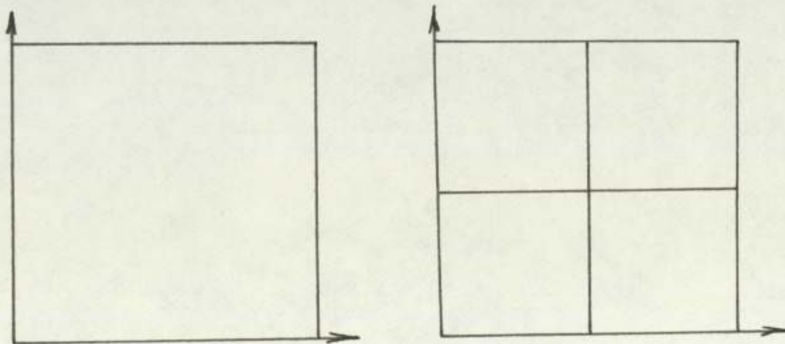
{further computation is required in the non-converged subregions}

FI

OD

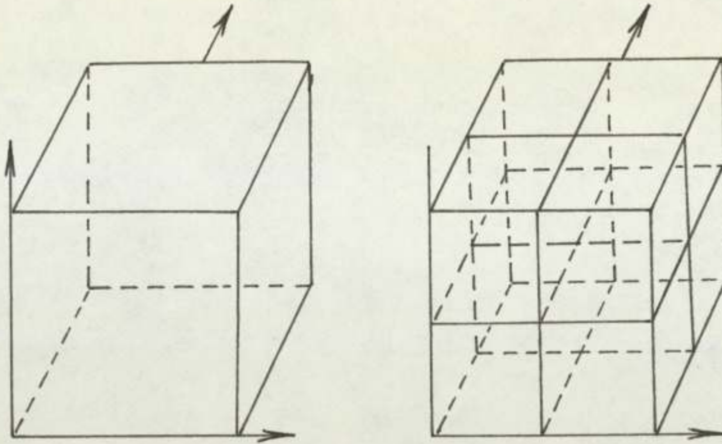
4.4 The subdivision strategy used with the hypercube

It is necessary to subdivide the original hypercube into a number of subregions, each of which is another hypercube. The minimum number of subregions which can be used to achieve this is 2^n. Hence in two dimensions the square is subdivided into four squares thus:



original region subregions

In three dimensions the cube is subdivided into eight cubes thus:



original region subregions

In a similar manner the n dimensional hypercube is subdivided into 2^n subregions.

Now consider what it is necessary to store in each element of the list in order to adequately describe the subregion. Each of the rules used to derive approximations over hypercubes is based upon a set of nodes and associated weights. Most of the rules are given for a particular hypercube but can be transformed to any other hypercube, since they are invariant under an affine transformation. The nodes are usually given with respect to some origin, which is often the centre of the starting hypercube. In order to apply one of these rules to a different hypercube it is necessary to map the rule from the first hypercube to the second by considering a change of origin and a change of scale between the two. Hence to apply any given rule to any given hypercube it is only necessary to know the centre of the hypercube and the scaling factor relating the size, hypervolume, of the hypercube to the hypercube over which the rule is defined. Thus all that has to be stored in each element of the list in order to adequately describe the subregion, so as to be able to apply the basic rule and obtain an approximation to it's integral, is the centre of the subregion and it's associated scaling

factor. In fact, the scaling factor need not be stored with each element since it is the same for any given level of subdivision into subregions and the list only ever contains subregions at one particular level.

4.5 The basic rules used for the hypercube

Certain basic rules are used to compute the estimates to each subregion in the linked list. These rules are written in the form of procedures, each of which requires the same parameters. The parameters consist of the centre of the hypercube over which an estimate is required, scaling factors relating the hypercube to the initial hypercube and a function to evaluate the integral at any given node.

Initially the two basic rules used were Stroud's n+1 point rule of degree two and Stroud's 2n point rule of degree 3 ([57] chapter 8).
Stroud's n+1 point rule consists of:

points coefficients

$(r_{i,1}, r_{i,2}, \ldots r_{i,n-1}, r_{i,n})$** V/(n+1)

where V is the hypervolume

$r_{i,2k-1}$ = sqrt(2/3) . cos(2ikπ/(n+1)

$r_{i,2k}$ = sqrt(2/3) . sin(2ikπ/(n+1)

k = 1,2,...(n/2) i = 0,1,...n

If n is odd

$r_{i,n}$ = (-1)^i / sqrt(3)

The points are the vertices of a regular n simplex and they all lie inside the region.

Stroud's 2n point rule consists of:

points coefficients

$(r_{i,1}, r_{i,2}, \ldots r_{i,n-1}, r_{i,n})$ $V/(2n)$

where V is the hypervolume

$r_{i,2k-1} = \mathrm{sqrt}(2/3) \,.\, \cos(2k-1)i\pi/n$

$r_{i,2k} = \mathrm{sqrt}(2/3) \,.\, \sin(2k-1)i\pi/n$

$k = 1,2,\ldots(n/2) \quad i = 0,1,\ldots 2n$

If n is odd

$r_{i,n} = (-1)i \,/\, \mathrm{sqrt}(3)$

The points are the vertices of a regular n dimensional octahedron and they all lie inside the region.

**{The notation $(u_1, u_2, \ldots u_n ; u_{n+1})$ denotes the set of points consisting of the point $(u_1, u_2, \ldots u_n)$ and all the points which this maps into under the set of all (n+1)! linear transformations of Sn on to itself. These points can be found by forming all the possible permutations of the n+1 coordinates. If all u_i, i = 1,2,....n+1 are distinct this gives (n+1)! different expressions :

$(u_{i1}, u_{i2}, \ldots u_{in} ; u_{i n+1})$.

The first n components of these vectors are the coordinates of the desired points. If not all the u_i are distinct this will result in fewer than (n+1)! points in the set.}

4.6 Defining the integrand

In order to evaluate a particular integral it is necessary to pass the integrand to the routine. This is achieved by writing a procedure to evaluate the integrand at any given node and linking the procedure to the quadrature routine. This can be quite difficult

but was acceptable for this algorithm since it was only intended to be used by the author as a research and development tool. For a quadrature routine intended for a wider audience, possibly people with little or no computing experience, it would be necessary to provide a more agreeable user interface. A suitable approach might be to write a program which took as input an integral in the form:

$$\int \cdots \int_{Rn} y \, dx_1 dx_2 \ldots\ldots dx_n$$

where $y = f(x_1 x_2 \ldots\ldots x_n)$ and Rn is the region of integration. The program would then produce as output a procedure to evaluate the integrand at any given node.

4.7 Segmentation of the program

This program was written primarily as a research tool. Hence, in order to facilitate testing and development the program was written as a sequence of segments which are linked together by the main body of the program. Each of the segments can be replaced by an alternative and the effects considered.

The first segment contains the declarations for the constants and variables which are common to the other segments.

The second segment contains a procedure, MILLTIME, which has no parameters and delivers a LONG INT representation of the time of call. This procedure is used for timing the method as a basis of comparison.

The third and fourth segments contain procedures to apply the basic rules to evaluate an estimate to the integral over a region defined by the parameters. To some extent the program is independent of the basic rules used in that it is possible to use any alternative rule, provided an appropriate segment is written. Thus the testing of various formulae and combinations of formulae was possible by writing alternative segments with the same standard form and name for the procedures to apply the rules.

The fifth segment contains the procedure to evaluate the function defining the integral at any given node. Again multiple copies of this segment were written, each containing a procedure of the same name and form but each evaluating a different integral, thus simplifying the testing of the routine with various integrals.

The final segment is the main body of the program and links all the other segments together. Any further sections of the program which needed to be compared with alternatives could also be taken out as further segments.
The complete program is given in appendix [3.1].

4.8 The simplex as a basic region

As an alternative to the hypercube a simplex is considered as a basic region of integration. A simplex is merely an extension of the two dimensional triangle, so that in n dimensional space it is a figure defined by n + 1 vertices; thus in two dimensions it is a triangle with three vertices, in three dimensions it is a

tetrahedron with four vertices and so on.

One of the main advantages of choosing the simplex as a basic region is the fact that any simplex, regardless of the number of dimensions, can always be divided into two "similar" subregions, similar in that they are both simplexes. Thus the same type of basic rule can be used throughout and the number of subregions will grow less rapidly than is the case with the hypercube, which has to be subdivided into 2^n (where n is the number of dimensions of the problem) in order to produce "similar" subregions. Hence the algorithms for the simplex can follow more closely the method of the one dimensional algorithms where a subdivision into two subintervals has proved to be a better approach. The basic method adopted for the simplex is analogous to that adopted for the hypercube. However, certain features of the simplex, such as the hypervolume and the centroid, are important and these are now discussed in detail along with any slight changes that had to be made to the basic algorithm.

4.9 The hypervolume of a simplex

With many of the formulae for simplexes it is necessary to be able to determine the hypervolume V (Hammer and Stroud's terminology - [19]) or the size $\sigma(S)$ (Silvester's notation - [54]) of the region of integration. Silvester gives the following approach to finding $\sigma(S)$:

Let a simplex be defined by it s n+1 vertices in the n space spanned by the coordinates $x^1, x^2, \ldots x^n$. Let S be the n dimensional simplex whose kth vertex coordinates are x_k^i where i = 1,2,...n. Then the

size $\sigma(S)$ is defined in the following manner :

$$\sigma(S) = \frac{1}{n!} \begin{vmatrix} 1 & x_1^1 & x_1^2 & \dots & x_1^n \\ \cdot & \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \cdot & \dots & \cdot \\ 1 & x_{n+1}^1 & x_{n+1}^2 & \dots & x_{n+1}^n \end{vmatrix}$$

Under this definition the size of a one dimensional simplex is it's length, the size of a two dimensional simplex is it's area, that of a three dimensional simplex it's volume, and so on. However, using this definition the sign of the size is undefined, being dependant upon the way in which the vertices are ordered. Therefore it is usual to adopt $|\sigma(S)|$ as the size of a simplex in any formulae, so that the size is always positive.

Since the term hypervolume has been used in conjunction with the hypercube previously this term will also be used for the simplex throughout the rest of this thesis in preference to the term size, although the two are interchangeable.

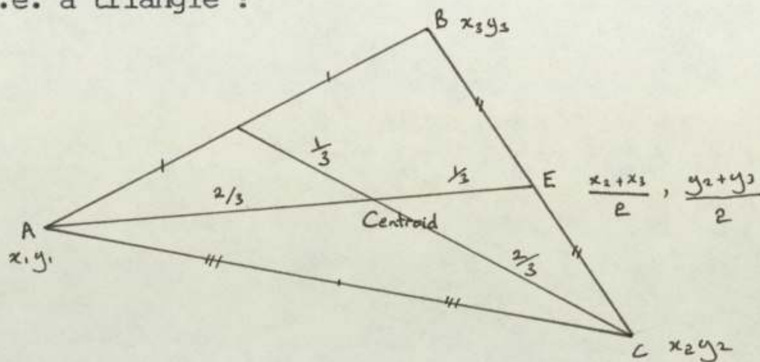
4.10 The centroid of a simplex

The centroid of a simplex is used in many of the integration formulae for the simplex. The centroid of a simplex is defined as the point of intersection of it's medians, that is a point of trisection of each median. Let the vertices of the n simplex Sn be V_0, $V_1, \dots V_n$. Then the centroid, C, of Sn is given by :

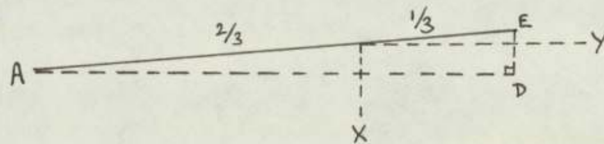
UNIVERSITY LIBRARY

$$C = \sum_{i=0}^{n} Vi \ / \ (n+1)$$

To illustrate this consider the centroid of a two dimensional simplex, i.e. a triangle :



Consider the median AE



$X = x_1 + 2.(AD)/3 = x_1 + 2.((x_1+x_2)/2 - x_1)$

$\quad = (x_1+x_2+x_3)/3$

Similarly

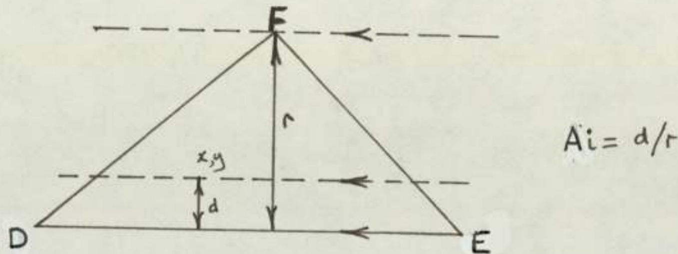
$Y = (y_1+y_2+y_3)/3$

that is the centroid $(X,Y) = \sum_{i=0}^{n} Vi/(n+1)$

4.11 Area coordinates

Area coordinates are a means of describing a point within a triangle. Each point is defined by means of three coordinates Ai,

Bi, Ci. For example one area coordinate of the point (x,y) is given by the ratio d/r where d is the perpendicular distance from the side of the triangle DE to the point (x,y) and r is the distance of the vertex F from the side DE. The other area coordinates of the point (x,y) are defined similarly. Obviously only two area coordinates are required to define a point uniquely.



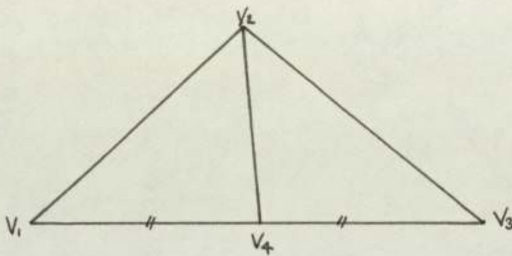
Area coordinates are a useful means of describing the nodes of a quadrature formula. Any triangle can be mapped onto any other triangle by means of a linear transformation and under such a mapping area coordinates are invariant. Hence a quadrature formula developed for a particular triangle can be applied to any other triangle relatively simply provided it s nodes are described in terms of area coordinates.

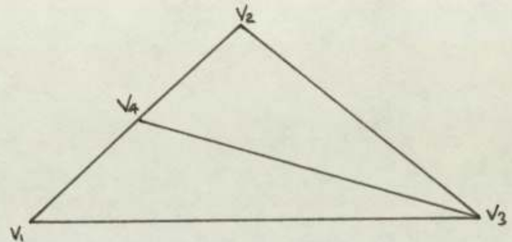
Area coordinates have an obvious extension to n dimensions.

4.12 Subdivision strategy used with the simplex

Any simplex can be subdivided into two further simplexes, hence the overall subdivision strategy used with the simplex is one of continually splitting each subregion into two further subregions. However there are a wide variety of ways in which a simplex can be split into two simplexes. It is preferable to use a subdivision strategy which leads to "compact" subregions since this ensures an even distribution of the nodes of the basic rules used. With non

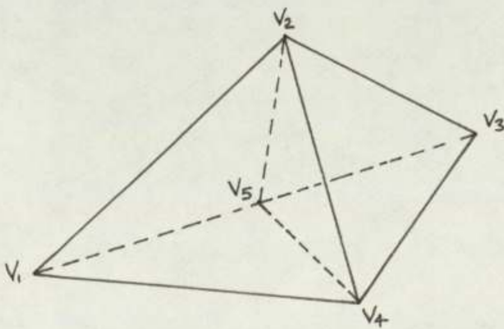
"compact" subregions the nodes could easily become clustered at one end of the region thus giving a false impression of the integrand. Now consider how the subdivision takes place. If the simplex is two dimensional, i.e. a triangle, then a new vertex is formed along one side of the simplex and a line drawn to the opposing vertex in order to divide the original simplex in two. Thus :

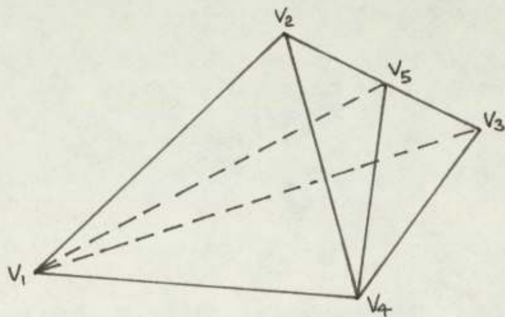




compact non compact

With a three dimensional simplex, i.e. a tetrahedron, a new vertex is formed along one side of the simplex and a plane constructed through the opposing side of the simplex in order to give the two new simplexes. Thus :





compact non compact

It is easy to see that "compact" and equal hypervolume simplexes result from choosing the midpoint of the longest side of the original simplex as the new vertex when generating the two new subregions. This was the method adopted in the algorithms.

The n dimensional analogy is straightforward. In order to split an n dimensional simplex into two a new vertex is formed along one side

of the original simplex, say vertex Vn+1 is formed between the original vertices Vi and Vi+1, then the new vertex replaces one of it s neighbouring vertices in the set of vertices defining the original simplex in order to define the first new simplex and the other neighbouring vertex in the set in order to define the the second new simplex. That is the two new simplexes are defined by the vertices V0,...Vi,Vn+1,Vi+2,...Vn and V0,...Vi-1,Vn+1,Vi+1,...Vn .

4.13 The structure of the algorithm as applied to the simplex

The structure of the algorithm as applied to the simplex follows exactly the structure of the basic algorithm given in section 4.3. The main differences between the application of the algorithm to the simplex and the hypercube is that subregions now refers to simplexes and the subdivision strategy is as described above. The elements in the linked list of subregions are different in that the amount of information required to describe each simplex is not the same as that required to describe each hypercube.

4.14 The basic rules used for the simplex

It is somewhat easier to discuss formulae for the simplex if they are all considered in reference to one particular simplex. Hence the author adopted the notation used by Stroud to define all the formulae used over the following basic simplex.

Let Sn - the n dimensional simplex with n+1 vertices

(0,0,....,0)

(1,0,....,0)

(0,1,....,0)

. . ..

. . ..

. . ..

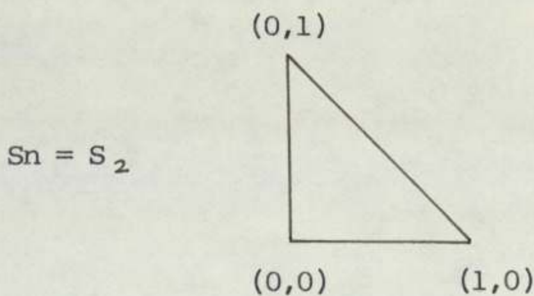
(0,0,....,1)

The hypervolume of Sn is denoted by V = 1/n! .

Any point $(x_1,x_2,...,x_n)$ on the simplex satisfies the following :

$x_1 + x_2 + ... x_n <= 1 \quad x_i >= 0 \quad i = 1,2,...,n.$

For example in two dimensions :



V = 1/2

The two basic rules used initially were Hammer and Stroud's [19] formula for the quadratic polynomial and Lauffer's [30] degree two formula. Both of these are given in Stroud[57].

The first basic rule

Hammer and Stroud's degree 2, n+1 point formula :

points coefficients

(r,r,...r;s) (1/(n+1)) V

where V is the hypervolume

r = (n+2 $\mp$ sqrt(n+2))/(n+1)(n+2)

s = (n+2 $\pm$ n.sqrt(n+2))/(n+1)(n+2)

The upper sign was chosen since this gives a formula with all the points inside the simplex for all n whereas the lower sign gives a formula with all the points outside the simplex for n>=3.
The points of this formula lie along the medians of the simplex.

The second basic rule
Lauffer's degree 2 , (n+1)(n+2)/2 point rule.

points	coefficients
(0,0,...,0;1)	B
(0,0,..0,r;r)	C

where V is the hypervolume

r = 1/2

B = (2-n)*V/(n+1)(n+2)

C = 4*V / (n+1)(n+2)

The points for this formula are the vertices of the simplex and the midpoints of the sides of the simplex.

These two rules were chosen as a pair because they provide an even

distribution of points across the region of integration. To illustrate this consider the geometrical position of the nodes of the two formulae in relation to the two dimensional version of the basic region Sn :

The vertices for this region are (0,0), (0,1) and (1,0).
For the first rule

$$r = (n+2-\mathrm{sqrt}(n+2))/(n+1)(n+2) = 1/6$$

$$s = (n+2+n.\mathrm{sqrt}(n+2))/(n+1)(n+2) = 2/3$$

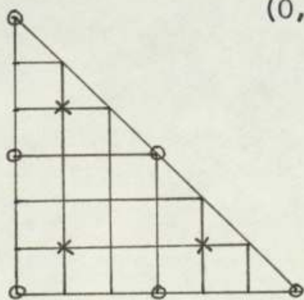
hence the points are (1/6,1/6), (1/6,2/3) and (2/3,1/6)

For the second rule

$$r = 1/2$$

hence the nodes are (0,0), (0,1), (1,0),

(0,1/2), (1/2,0) and (1/2,1/2).



o points for Lauffer's rule

x points for Hammer and Stroud's rule.

The basic rules which are used to compute the estimates over the subregions are written in the form of procedures. The procedures are contained in separate segments and the same parameters are used for

each procedure since this facilitates the simple interchange of alternate formulae. The segments containing these procedures are given in appendix [3.2].

4.15 The program for the simplex

As with the program for the hypercube, the program for the simplex was written as a sequence of segments which are linked together by the main body of the program. Full details of all the segments are given in appendix [3.2].

4.16 The data required by the program

In effect the segment SEGF is part of the input to the program since this defines the function to be integrated. However, a selection of possible integrals and the corresponding segments SEGF are given in appendix[1]. This section is concerned with the data that is required to drive the program once the problem has been defined in the form of segment SEGF.

This data consists of the number of dimensions of the problem, the required number of test runs of the problem, the tolerance for each of these runs and sufficient information to define the region of integration. The first three items are quite straightforward, but the third needs a little more explanation.

First consider the hypercube. Although the basic rules were based initially on formulae derived for the hypercube $-1 \le x \le 1$ it is

possible to solve integrals over other hypercubes using these formulae, by mapping nodes from the initial hypercube to the new region. This approach is acceptable since the rules are invariant under such an affine transformation. The data required to achieve this mapping is as follows:

1. The factor relating the new nodes to the old nodes, i.e. the factor by which the side of the hypercube needs to be divided or multiplied to give the side of the new hypercube.
2. The scaling factor relating the hypervolume of the initial region to the hypervolume of the new region.
3. The centre of the new region.
4. The offset of the centres of the first subregions of the region from it s centre.

Thus for example the data required by the program for the following two problems:

$$\int_{-1}^{+1}\int_{-1}^{+1} f(x_1,x_2)dx_1dx_2 \quad \text{and}$$

$$\int_0^1\int_0^1\int_0^1 f(x_1,x_2,x_3)\, dx_1dx_2dx_3$$

consists of

Problem 1	Problem 2	
2	3	the number of dimensions of the problem
3	4	the number of tests to be performed
.5,.2,..	.5,.2,..	the tolerances one for each test
1	2	divl the division factor relating the new nodes to the old nodes
0.5	0.25	the offset of the subcentres of the subregions

of the new region

1 8 the scaling factor relating the hypervolumes

0.0 0.5 the centre of the new region.

Now consider the simplex. The data required to define the region of integration consists of:

1 the vertices of the simplex

2 the hypervolume of the simplex.

Hence for the problem

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

the data would consist of:

2 the number of dimensions of the problem

4 the number of tests to be performed

.5,.1,...the tolerances one for each test

0,0

0,1 the vertices

1,0

0.5 the hypervolume.

4.17 Testing the two programs

The program for the hypercube was tested using the set of test problems described in appendix [1]. Also the program was tested with an alternative pair of basic rules; namely the compound trapezoidal rule and Ewing's rule, details of which are given in appendix [5.1]. The procedures to evaluate an estimate to the integral using each of these rules are given in appendix [5.1]. The results of the test runs are given in appendix [8].

The program for the simplex was tested using the set of test problems described in appendix [1]. The results of these test runs are given in appendix [9].

The following set of tolerances were used for both programs: 0.5,0.1,0.05,....0.000001. The maximum jobtime for each run of the program was limited to 90 and the maximum core size 90k.

4.18 Conclusions

Both of the programs produced results to within the required tolerance for the majority of the problem and tolerance range. In fact the majority of results were far more accurate than the required tolerance. However, once the tolerances became too small the programs started to fail because of the large number of integrand evaluations required which resulted in either the time running out or no more space being available for the heap to expand. This was more of a problem with the hypercube program than with the simplex program.

It was noted that due to the nature of the subdivision process used with the hypercube the number of integrand evaluations used changed considerably from one tolerance to the next, if any change in the number occurred at all. For example, a specific number of integrand evaluations might be used for all tolerances larger than 0.001 and then a sudden increase in the number of integrand evaluations would occur for the tolerance 0.001. Hence the results produced with an error tolerance slightly larger than 0.001 might have an actual error marginally greater than the requested tolerance while the

results produced with an error tolerance of 0.001 might have an actual error far smaller than the requested tolerance.

The results produced using the hypercube program were not as accurate as those produced by the product Patterson program but the number of integrand evaluations used were far less and the tolerance was still satisfied for the majority of the range. Consequently the range of tolerances for which the method succeeded in producing results was greater than that of the product method.

The results using the second pair of rules (the compound trapezoidal rule and Ewing's rule) were noticeably worse all round than the results using Stroud's rules. Hence, Stroud' rules form a better pair of formulae for this program.

For comparison an Algol68 version of the Fortran routine of Genz [16] was written and tested on some of the test set of problems. However, the results of the comparison were not conclusive in any way and merely highlighted the problems of testing and comparing different routines. The first routine produced slightly better results on some problems or with particular tolerances while the second produced better results on others or with different tolerances.

One problem with this basic adaptive method is the large amount of global store that is used. In particular the heap is used to store the linked lists of subregions. Hence it is in constant use and frequently becomes fragmented which results in garbage collection being invoked. (This is the process of finding all the free space on

the heap and grouping it together.) Unfortunately this process is rather slow and needs to be avoided if at all possible. One solution would be to increase the amount of space allocated to the heap but this is not possible beyond a certain point because of the physical limits of the machine and the upper limit set by the compiler. Alternatively one or two modifications could be made to the program to make the use of the heap more efficient. The basic structure of the algorithm involves creating a list of subregions which is then replaced by a list of subregions of these subregions and the original list discarded. This list is then replaced by another new list of subregions of subregions and discarded. The process continues in this manner using more and more of the heap until convergence is achieved. It would be feasible to avoid some of the garbage collection by keeping a list of the free space which was made up of the discarded lists. Whenever more space was required it could be taken from the free space list, if the list was not empty, instead of using more of the heap. Instead of discarding the used lists they could be tagged on to the free space list.

The stopping criterion for the simplex was altered slightly because of a problem that occurred during the first few test runs. The program was tested on an integral which had a symmetric integrand function and the combination of the symmetry and the subdivision process resulted in the first two subregions having the same estimate and the sum of the two estimates being equal to the initial estimate over the whole region. Hence the program stopped after the first subdivision regardless of the tolerance because the difference between the latest two estimates to the integral was zero. The stopping criterion:

WHILE the list of subregions is not empty

  AND the difference between the latest two estimates

      is greater than the tolerance

was replaced by

WHILE the list of subregions is not empty

In conclusion the two programs produce satisfactory results but there are some slight improvements that could be attempted.

Chapter 5 Storing the integrand evaluations

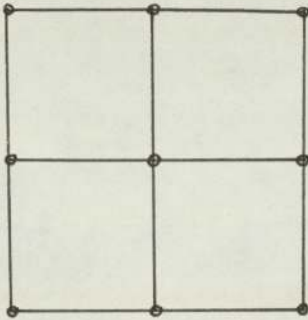
5.1 The need for storing integrand evaluations

With all but the simplest of problems a large number of integrand evaluations are needed to produce a reasonable approximation to the solution. Hence, if the integrand is a complicated expression the time taken to evaluate the integrand at each of the nodes can add considerably to the overall time taken to compute the result. This suggested the possibility of reducing the amount of computation by choosing the basic rules in such a way that some or all of the nodes used at one level of subdivision are used again at subsequent levels of subdivision and storing the integrand evaluations thus avoiding reevaluation of the integrand at the common nodes. Alternatively, the subdivision strategy could be chosen to enable common nodes to be used. In practice it is necessary to select both the basic rules and the subdivision strategy to take full advantage of common nodes.

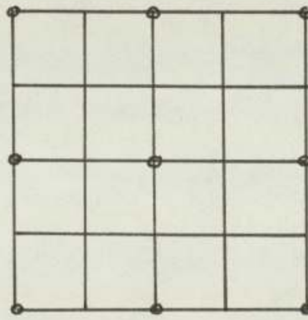
This approach is only feasible provided an efficient and convenient method is developed to store and access the integrand evaluations as the subdivisions take place. Even then, storing the integrand evaluations will only be a practical proposition when reevaluating the integrand at a given node takes longer than the overheads incurred in storing and accessing an integrand evaluation. This restricts stored integrand evaluation techniques to certain problems; those whose "cost" of evaluation is above a certain level. This chapter is concerned with the particular methods of storing the integrand evaluations and the subdivision strategies adopted by the author.

5.2 The basic rules and subdivision strategy for the hypercube

The subdivision strategy used in the previous algorithms consisted of subdividing the n dimensional hypercube into 2^n equal hypervolume subregions, each a hypercube. With this strategy any rules based on the corners of a hypercube, on the centre of a hypercube, or anywhere on a regular mesh(corresponding to the subdivision) result in common nodes on subdivision. For example in two dimensions:



Initial region



First level of subdivision

o - common nodes

In this example 16 new integrand evaluations would be required at the first level of subdivision if the previous integrand evaluations were stored as opposed to 36 integrand evaluations if none were stored.

Several rules are based upon nodes on this type of mesh, for example the product trapezoidal rule and Ewing's rule.

5.3 Storing the integrand evaluations in a linked list

In the first attempt to develop an algorithm which made use of

stored integrand evaluations the author chose to store the integrand evaluations in a linked list which could grow dynamically as the method proceeded. The list could then be searched in order to find an integrand evaluation associated with a given node. Each item in the list had to contain the following information: the numerical value of the integrand evaluation, a key (associating the integrand evaluation with a given node and providing a means of ordering the list), and a pointer to the next item in the list. To satisfy these needs items of an additional mode defined as:

```
MODE ITEM = STRUCT (INT index,
                    REAL fevaluation, REF ITEM ptr).
```

The integrand evaluation is stored in the fevaluation field of an item. The key is stored in the index field of an item and the pointer to the next item in the list is stored in the ptr field. This gives the items of the list but creates the problem of indicating the start and finish of a list. The start of a list is indicated simply by using a pointer, for example REF ITEM head, which is a reference to the first item in the list. In order to signify the end of a list it is necessary to have a null pointer to assign to the ptr field of the last item in the list. Fortunately in Algol 68 NIL fills these requirements.

Thus, the linked list consists of a pointer to the start of the list, the chain of items in the list and the last item in the list with NIL assigned to it s ptr field indicating that there are no further items in the list.

5.4 Ordering the list

When a list is not ordered it is necessary to make a complete search of the list in order to determine that an item does not yet exist. Obviously this is very wasteful and was avoided to some extent by ordering the list. Since the keys are numerical values the natural choice was to order the list by magnitude of keys; the items with the smallest keys appearing at the start of the list while the items with the largest keys appear at the end of the list. With an ordered list a search can stop either when the required item is found or when an item is found with a larger key than the search key. The overheads incurred in ordering the list are few since the list needs to be searched anyway and an unsuccessful search automatically gives the place of insertion of the new item in the list, that is immediately before the first item with a key larger than the search key.

The choice of the keys is a very difficult area. The keys have to be simple to compute yet uniquely linked to the coordinates of the given node. The various methods considered by the author are discussed under the section 5.8 and for the present it is sufficient to know that it is possible to obtain the value of such a key by the use of a procedure called enumerate.

5.5 The generation and use of linked lists

In order to generate and use linked lists it is necessary to be able to create new items of the correct type for the list, insert these items into the list and search the list for an item with a given

key. The list can be generated from an empty list, that is a null pointer to the start of the list, by creating new items globally and linking these items to the list by insertion at the correct position. In Algol 68 global variables can be created and stored in a dynamic storage area called the heap.

Insertion is achieved by the use of a procedure "insert" which creates a new item globally, assigns the appropriate data to it and connects it to the linked list at a given point. The point of insertion of the new item is given by the procedure "searchlist". This procedure searches a list for a given node and delivers either the position of that node in the list, if it exists, or the position in the list where it should be, that is the position of insertion. These procedures are now discussed in detail.

5.6 A procedure to search a linked list

The following procedure "searchlist" searches a list of items,of the type MODE ITEM = STRUCT (INT index, REAL fevaluation, REF ITEM ptr), for a given item and delivers either TRUE if the item is in the list or FALSE if it is not. It also delivers as a parameter a pointer to either the required item or to the position where this item should be inserted. The search takes place by a comparison of the keys in the list which are stored in ascending order of magnitude of the keys. Thus searching merely consists of comparing the search key (that is the key of the given item) with each of the keys of the items in the list one at a time. This process continues until one of the following occurs: the end of the list is reached, a key equal to the search key is found, or a key greater than the search key is

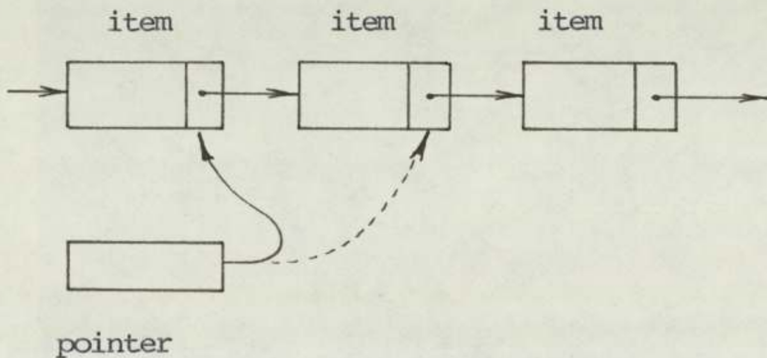
encountered, in which case the given item cannot be in the list. The procedure consists of:

```
PROC searchlist = (REF REF REF ITEM pointer, INT key) BOOL:
BEGIN
{This procedure searches a list for a node with a given key, it
delivers TRUE if the node is in the list and FALSE otherwise}
    BOOL not found := TRUE, possible := TRUE ;
    {not found indicates whether the item has been found while
    possible indicates whether the item could be in the list}
    IF pointer ISNT empty
    {This tests for an empty list and is redundant when only non
    empty lists are used}
    THEN
    WHILE not found AND possible DO
    BEGIN
        REF INT indp = index OF pointer ;
        IF indp = key
        THEN notfound := FALSE  {the item has been found}
        ELSF indp > key
        THEN possible := FALSE
            {item is not in the list}
        ELSE
            IF (ptr OF pointer) IS empty
            THEN possible  := FALSE
                {list is now empty}
            FI;
            pointer := ptr OF pointer
            {move on to the next item in the list}
        FI
```

```
    END
    FI ;
    NOT not found
    {deliver TRUE if the item is in the list and FALSE otherwise}
END of the procedure searchlist.
```

The need for a pointer parameter with mode REF REF REF ITEM is perhaps not obvious and may be understood more easily by considering a diagrammatic representation of what is taking place.

Linked list



The pointer contains a reference to the ptr field of a given item and as the search takes place its value changes accordingly. Suppose the declaration ITEM first; occurs then :

first is of mode REF ITEM

index OF first is of mode REF INT

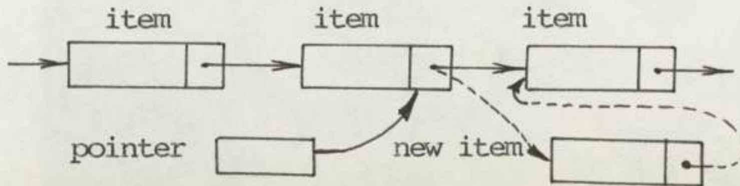
fevaluation OF first is of mode REF REAL

and ptr OF first is of mode REF REF ITEM.

Hence, if the pointer is to contain a reference to the ptr field it needs to be of mode REF REF REF ITEM. It would of course be easier to search at one level of reference less by having a pointer to the item instead of to the ptr field of the previous item. However, this would make insertion more difficult. As it is insertion is simple and consists of making the ptr field of the new item contain the

value held in the ptr field referred to by the pointer and then making the ptr field referred to by the pointer contain the reference to the new item.

Linked list

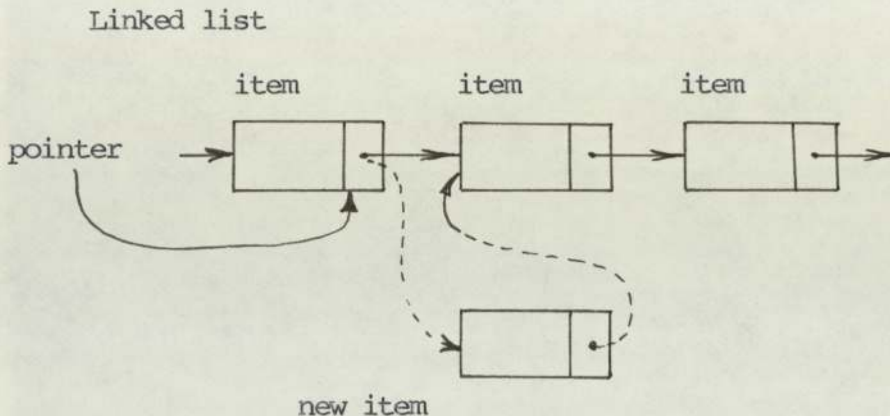


The algorithm written to search a linked list is a satisfactory method of accessing the nodes, however using the basic structure imposes one or two slight disadvantages which might be avoided by using a more complex structure. For example the larger the key is the longer it will take to either find it or to discover that it is not in the list. As the list gets longer the average search time will increase. An alternative approach might be to use a double linked list which can be searched in both directions or to have a sequence of pointers to various positions in the list from whence the search can begin. This could be very useful since the range of key values is known but the distribution is dependent upon the problem because the method is adaptive. A second alternative may be to build up a tree structure rather than a linked list and adopt a binary type search. There is scope for improvement and further development here using the various approaches, however the author had to restrict the work to the method described.

5.7 A procedure to insert an item in a linked list

The following procedure creates an item globally on the heap, assigns the appropriate data to it (i.e. the integrand evaluation and the key) and inserts it in the linked list. The position of

insertion is immediately after the item whose ptr field is indicated by the parameter "pointer". The insertion is achieved by assigning the value of the ptr field indicated by pointer (i.e. the reference to the next item in the list) to the ptr field of the new item and then the reference to the new item is assigned to the ptr field indicated by pointer. Diagrammatically :



Dotted lines indicate the changes made in the links when insertion takes place.

The complete procedure consists of :

```
PROC insert = (REF REF ITEM pointer, INT key, REAL feval) VOID :
BEGIN
    {This procedure creates a new item and inserts it in the list
    at the position indicated by pointer}
    REF ITEM newitem = ITEM ;
    {This creates a new item globally on the heap}
    index OF newitem := feval ;
    {assign the appropriate data to the new item}
    ptr OF newitem := pointer ;
    {Link the new item to the next item in the list}
    pointer := newitem
    {Link the previous item in the list to the new item}
```

END of the procedure insert.

This procedure is quite straightforward and requires little or no explanation. The only place where clarification is required is handling the references. The result of ptr OF newitem becomes pointer is to dereference pointer once to give the reference to the next item in the list, which is then assigned to ptr. When pointer becomes newitem a straightforward assignment is made with no dereferencing.

5.8 Enumerating the keys

The problem of computing a unique key for any given node in any number of dimensions is almost impossible, if the coordinates are unpredictable real numbers. However, the complexity of the problem is reduced to some extent if the basic rules are chosen such that the coordinates which they use are integers or can be converted easily to integers. For example consider the product trapezoidal rule in which the nodes are the corners of an n dimensional hypercube. In two dimensions the starting region may be the square $0 \le x_i \le 1$ i=1,2 and the nodes will then have coordinates (0,0), (0,1), (1,0), and (1,1). If each of these nodes is considered as a two digit number to the base two then each has a unique integer equivalent which may be used as it s key : 00 = 0, 01 = 1, 10 = 2, and 11 = 3. If the region is subdivided into four similar subregions and adjusted versions of the basic rule applied to each then the new nodes would all have coordinates that are multiples of 0.5. For example the new nodes for one of the subregions would be (0.5,0.5), (0.5,1), (1,0.5), and (1,1). These could be converted to integer

pairs by multiplying each coordinate by two to give (1,1), (1,2), (2,1), and (2,2). If each of these is considered as a two digit number to the base 3 (base 3 since there are three possible values for each digit 0,1,2) then each has a unique integer equivalent which may be used as it s key : 11 = 4, 12 = 5, 21 = 7, and 22 = 8. This idea can be continued to the next level of subdivision introducing terms involving multiples of 0.25 and generating integers to the base 5 and so on to any level of subdivision.

Hence using the above approach it is possible to obtain a unique integer key representing a given node to a given base, provided the coordinates of the node are exact multiples of a fraction corresponding to the base. One problem with this is that common nodes would have different integer keys depending on the base of the number system used, which is dependent on the level of subdivision, thus losing all the benefits of using stored integrand evaluation techniques. This can be overcome in one of two ways. The first involves fixing the maximum level of subdivision and hence the maximum base possible. All the keys can then be evaluated to this base using the corresponding division factor. This results in one long list of items each with fairly high order keys. This method suggests the use of an array to store the integrand evaluations instead of a linked list since all the keys will be integers and the range will be known. The range is dependent on the base chosen as the maximum and so the base and thus the level of subdivision would be dependent on the amount of store available for the array. However the number of elements required in this array would be very large and hopefully the majority of them would not be used since the algorithm is adaptive and should concentrate the nodes in the

regions where the integrand is "badly" behaved. If most of the elements are used then this suggests that the integrand is either uniformly "well" behaved or uniformly "badly" behaved and in either case there is no advantage in using an adaptive method.

An alternative approach is to use the bases related to the subdivision level and store the keys thus generated in separate lists. To overcome the problem of common nodes having different keys it is necessary to adopt a strategy whereby any common node is only stored at it s lowest key level. This requires the ability to recognise when a node already exists at a lower key level and to obtain the appropriate key and associated list. Fortunately this is quite easily achieved. Consider the node (1,1) which is likely to be common over several levels of subdivision. The node (1,1) will be stored in the list containing integers to the base 2 and it s key will be 11 = 3. At the next level of subdivision the node (1,1) will have the key 22 = 8 and at the next level of subdivision it will have the key 44 = 24 and so on. Therefore the node (1,1) gives the keys 11, 22, 44, 88, ..etc. at different levels of subdivision. From this it can be seen that if all the digits in the key are even then the node exists at the next level down and so on. Once the key contains an odd digit then it cannot exist at a lower level. Thus the appropriate key and the base for the lowest level at which any node can possibly be stored may be obtained by a simple process of division. One benefit of generating a number of lists in this manner,i.e. one associated with each level of subdivision, is that the integrand evaluations accessed most often will be contained in the shortest lists. At the first level of subdivision for a p point rule there will be p points stored in the first list. At the second

level of subdivision there will be less than p*2^n points stored in the second list, since the initial region is subdivided into 2^n subregions. Of course some points will be common with those from the first level of subdivision, thus reducing the total of points from p*2^n and in the later stages of subdivision the adaptive nature of the algorithm will avoid the use of nodes in certain subregions. This should reduce the overall searchtime for a given key since shorter lists need to be searched.

Both of these approaches can be implemented in the form of a procedure "enumerate" which takes the node as a parameter and evaluates the corresponding key, which is delivered as a second parameter. The first approach is very easy to implement and consists of the following:-

```
PROC enumerate = (REF[]REAL node, REF INT key) VOID :
BEGIN
     {This procedure enumerates a searchkey given a node using the
     first approach}
     key := 0 ;
     FOR i TO n DO key := key * base + ENTIER(node[i]*multiplier)
     {n is the number of dimensions of the problem
      base is the base of the number system being used
      multiplier is the reciprocal of the division factor
      corresponding to the base}
END of the procedure enumerate.
```

The second approach is slightly more complicated and involves some additional parameters. In this case enumerate delivers the key relating to a given node and a pointer to indicate the list in which

a node with this key should be found, if it already exists. The pointer is given by a parameter "ptr" which is used to select from an array of pointers to the various lists; the first element of the array is a pointer to the list of keys to the base 2, the second element is a pointer to the list of keys to the base 3, and so on. The procedure can be broken down into two main parts:

a) find the lowest possible base level for a given node

and

b) evaluate the key for this node based upon the information about the base.

The second part is equivalent to the first version of enumerate given above. The first part involves starting at the preset base level, considering if the node could exist at a lower base level, moving to that level if it could and otherwise stopping. The complete procedure using the second approach consists of the following:

```
PROC enumerate = (REF[]REAL node, REF INT key,ptr,
                    INT b, REAL al) VOID :
BEGIN
    {This procedure enumerates a searchkey, given a node}
    BOOL possible := TRUE ;
    {possible indicates that the node could be stored at a lower
    base level}
    REAL alt := al ;
    {alt is the divisor associated with the given base b}
    INT base := b ;
    While base > 2 AND possible DO
    {2 is the lowest base possible}
```

```
BEGIN
      FOR i TO n DO IF odd(ENTIER(node[i]/alt))
                    THEN possible := FALSE
                    FI ;
      {if any of the digits to this base are odd then the node
      cannot be stored at a lower base level}
      IF possible
      THEN {consider the next base level}
           base := (base + 1) '/'2;
           alt TIMES 2 ;
           ptr MINUS 1
      FI
END ;
{base is not at the correct level for this node, so evaluate
the key}
key := 0;
FOR i TO n DO key := key*base + ENTIER(node[i]/alt)
END of the procedure enumerate.
```

From this procedure it can be seen that enumerating a key for a given node is quite expensive in terms of computer time. Hence, since the procedure is called so often, any improvements in this area would be beneficial. At this point it should be noted that if the time taken to evaluate the key is longer than the time to reevaluate integrand then no saving can be made by storing the integrand evaluations.

An obvious fault with the above procedure is the repetition of the calculation ENTIER(node[i]/alt) which is in itself quite an

expensive operation. This was avoided by the introduction of an array [1:n]INT to store the digits computed from the node. These digits can then be repeatedly divided by two until one of them is odd. This also removes the need for alt and the associated multiplication alt TIMES 2. A further improvement was the removal of the calculation of the base (base := (base+1)'/'2) by storing the possible bases in an array and altering a pointer to this array. The variable "ptr" can be used for this purpose. The procedure enumerate with these alterations consists of the following:

```
PROC enumerate = (REF[]REAL node, REF INT key,ptr,
                 INT b, REAL al) VOID :
BEGIN
     BOOL possible := TRUE ;
     [1:n] INT digits ;
     {compute the digits from node and store them in the array
     digits, if any of the digits are odd set possible to FALSE}
     FOR i TO n DO IF odd(digits[i] := ENTIER(node[i]/al))
                 THEN possible := FALSE
                 {node cannot be stored at a lower base level}
                 FI ;
     {consider the lower base levels}
     WHILE ptr > 1 AND possible DO
     {pointer now selects from an array of bases and the lowest
     possible base level,2, is stored in the first element of the
     array, thus ptr > 1 is equivalent to base > 2 }
     BEGIN
          FOR i TO n DO
               IF odd(digit[i] := digit[i]  2)
```

```
            THEN possible := FALSE
            FI ;
        IF possible
        THEN {consider the next base level}
             ptr MINUS 1
        FI
    END ;
    {ptr now indicates the correct base level for this node, look
    up the base and evaluate the key}
    INT base := bases[ptr] ;
    {bases is the array of base levels}
    key := 0 ;
    FOR i TO n DO key := key*base + digit[i]
END of the procedure enumerate.
```

5.9 The basic program

Storing the integrand evaluations affects the procedures to apply the basic rules, in that stored integrand evaluations are used where possible in preference to reevaluating the integrand, but does not affect the basic program used to apply these procedures. The structure of the program follows exactly the basic structure defined in 4.3. The complete program is given in appendix [5.1].

5.10 Scatter storage techniques

The techniques used so far for storing the integrand evaluations have been based upon the use of linked lists ordered according to the given keys. This approach is quite satisfactory for short lists

but becomes rather too slow when the lists become longer, since the searching involves a linear search of the list.

An alternative approach is to use scatter storage techniques. The fundamental idea behind scatter storage is that the key associated with the desired entry, that is the integrand evaluation in this case, is used to locate the entry in storage. Some transformation is performed on the key to produce an address (sometimes called the hash address or hash code) in a table which holds the key and the entry associated with it. This avoids searching the list. If the hash codes are such that the same hash code may be generated from different keys a method is needed for resolving the collision of keys. This is one reason why the key needs to be stored alongside the entry.

Various methods have been developed to handle collisions and the one that seems to be most suitable to this work is called direct chaining [45]. With this technique part of each entry is reserved as a pointer to indicate where additional entries with the same calculated addresses are to be found, if there are any. Thus all of the entries with the same hash code are to be found in a linked list starting at the address indicated by the hash code. Once a hash code has been generated the corresponding linked list must be searched linearly (possibly ordering the list and using the techniques described previously). However, all the lists will be relatively small if the number of collisions is kept to a minimum. In fact if the data is distributed evenly throughout the scatter storage table no searching will take place until most of the table has been filled. Therefore the larger the size of the table the quicker

access should be, up to a point. The aim of a good hash code is to distribute the data evenly throughout the available store, thus avoiding collisions. More will be said about this later in the section on hash code generation.
A general introduction and useful summary of the scatter storage techniques presently in general use are given in [45].

5.11 Development of an algorithm based upon scatter storage techniques

The following describes the various sections of the algorithm developed to take advantage of the scatter storage techniques described above.

The algorithm follows the previous algorithm to a large extent, but the method of storing the integrand evaluations is altered, a scatter storage technique being used. Thus, to avoid repetition, only the new sections of the algorithm are discussed in detail.

5.12 Key generation

A unique key is required to store alongside each integrand evaluation, since a hash code will be used that allows collisions. The key needs to be compared with a search key to ensure that the correct integrand evaluation is obtained for a given node. A LONG INT value is used for the key, in order to increase the range of possible values. The key is generated uniquely from the coordinates of a given node. It is convenient initially to generate two integer keys (key1 and key2) from the node, which can then be combined to

form a single LONG INT. These two keys are also used in the hash code generation.

```
PROC eval key = (INT key1, key2) LONG INT :
BEGIN
    {This procedure evaluates a LONG INT key given two integer keys
    "key1" and "key2"}
    LONG INT newkey ;
    newkey := LENG key2 ;
    newkey := newkey*(LONG 10000000) ;
    newkey := newkey + (LENG key1) ;
    {This combines the two keys}
    newkey
END of the procedure eval key.
```

The number of dimensions of the problem determines the way in which the two INT keys, "key1" and "key2", are generated. With a two dimensional problem it is simply a case of constructing the first key, "key1", from the first coordinate and the second key, "key2", from the second coordinate. These two integers are generated in a similar manner to the methods used in the procedure enumerate earlier. Each coordinate is a real number in the range 0 - 1 and ENTIER(coordinate*const) is computed to givethe corresponding integer. The variable const is the reciprocal of the fraction associated with the lowest level of subdivision. This is equivalent to finding the integer number of times that the fraction divides into the coordinate but is slightly more efficient since division is slower than multiplication.

With a three dimensional problem the three integers associated with the three coordinates of a node are generated in the same way as above. The largest value of any of these integers depends on the level of subdivision allowed, hence by putting a limit on the level of subdivision it is possible to ensure that only the first fifteen bits of any of these integers are significant. One integer contains twenty four bits and so it is possible to form the first key, "key1", from all the significant bits of one of these integers and the first eight significant bits of the second. similarly "key2" can be formed from the next seven significant bits of the second integer and all the significant bits of the third integer.

Higher dimensional problems require a similar approach and corresponding restriction on the level of subdivision. Obviously more words could be used to represent the keys but this would make the method less efficient. Thus this method is more restrictive in some ways than the previous method.

The procedure which is used to produce the two keys from a given node is as follows:

```
PROC compute keys = (REF INT key1, key2, REF[]REAL x) VOID :
BEGIN
     {This procedure computes the two keys for the given node x}
     CASE (n-1)   {n is the number of dimensions}
     IN( {2d problem}
         key1 := ENTIER(x[1] * const);
         key2 := ENTIER(x[2] * const) );
       ( {3d problem}
```

```
        BITS b ;
        INT digit := ENTIER(x[3] * const);
        b := BIN digit ;
        key1 := ABS ((b SR 8) SL 15)
                  OR (BIN(ENTIER(x[2]*const))))
    OUT SKIP {other dimensions not yet included}
    ESAC
END of the procedure compute keys.
```

Further dimensions can be added as required.

5.13 Hash codes

A good hash code is required to distribute the data evenly throughout the available store. The hash code could be generated from the LONG INT key produced by the procedure "evalkey" but it is slightly easier to make use of the two keys, "key1" and "key2", from which the LONG INT key is generated.

Various methods of hash code generation are available and a good review of the possibilities appears in [45]. Briefly the most commonly adopted approaches are as follows:

a) Choose some bits from the middle of the square of the key, enough bits to be used as an index to address any item in the table. Since the value of the middle bits of the square depend on all the bits of the key there is a high probability that different keys will give rise to different hash codes, more or less independently of whether or not the keys share some common feature.

b) If the keys are multiword items, then some bits from the product

of the words making up the key may be satisfactory as long as care is taken to ensure that the calculated address does not turn out to be zero most of the time.

c) A third approach is to cut the keys up into n-bit sections, where n is the number of bits needed for the hash address, and then form the sum of all these sections. The low order n bits of the sum are then used as the hash address. This method can be used for single word keys as well as for multiword keys.

The method adopted by the author is a slight variation on the third of these basic ideas. The method consists of adding the two INT keys and dividing the result by the maximum possible value of the sum of these two keys. This gives a real value in the range 0 to 1 which can be converted to any required range by multiplying by the number of elements in the range and taking the largest integer less than the resulting real number. The procedure used to generate a hash value given two integer keys is as follows:

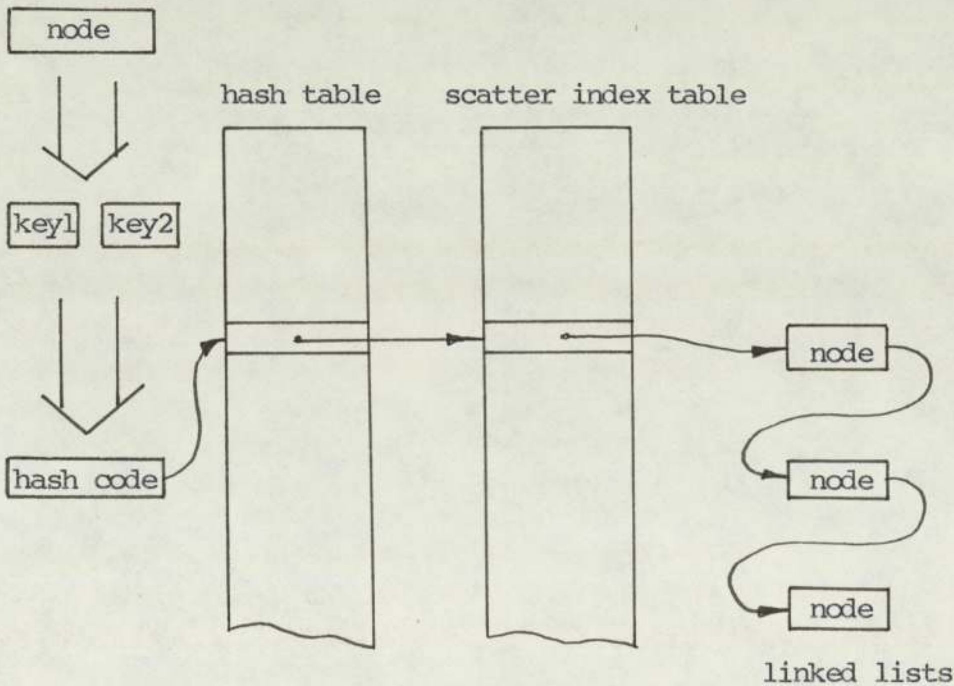
```
PROC compute hash value = (INT key1, key2) INT :
{This procedure generates a hash value}
(ENTIER(1024*ABS((key1+key2)/maxint)))
```

The variable maxint, as it s name suggests, is the maximum value of the sum of the two keys, "key1" and "key2".

5.14 Finding the integrand evaluation at a particular node

In the previous sections the generation of a pair of keys from a given node and the generation of the corresponding hash value from

these keys was discussed. Now the method of applying this hash code to the storing and accessing of integrand evaluations will be approached. First consider the data structure that is to be used. Starting from a node the keys are generated and from the keys the hash code is produced. This gives the position in a table where an index into the scatter storage table for that hash code is located. Within the scatter storage table all the integrand evaluations associated with the nodes which produce the same hash code are stored in a linked list. Pictorially the data structure may be viewed as :



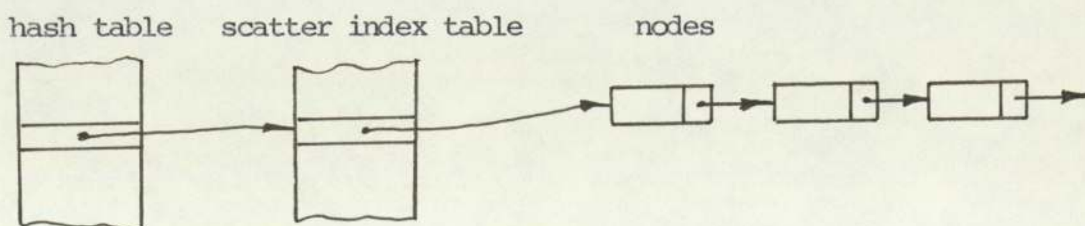
Each node in the linked lists consists of :

a key	an integrand evaluation	a pointer to the next node

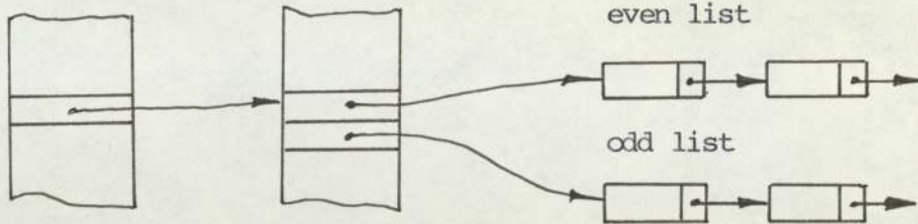
The hash table is simply an array of pointers into the scatter index table. The scatter index table can be either an array of references to lists of integrand evaluations, in which case each element is a single item, or it can be an array of the first nodes in the lists

of integrand evaluations, in which case each element would be a structure. In the first case no space is pre allocated for the storage of integrand evaluations whereas in the second case space to store one integrand evaluation, i.e. one node, is allocated per element of the scatter index table.

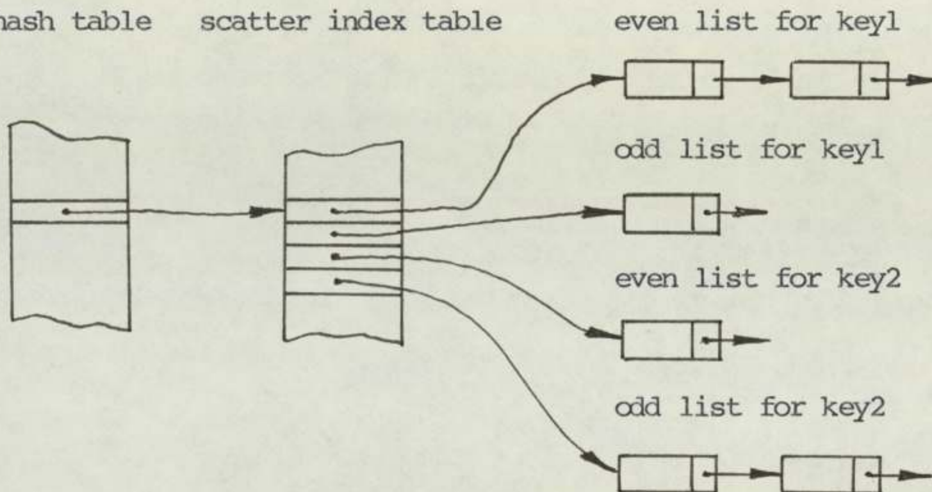
The author adopted the first approach which is somewhat simpler to achieve. Direct chaining is used to handle collisions and in it s simplest form this results in a one to one relationship between the hash table and the scatter index table which could thus be replaced by a single table. However, if the two are kept separate it is possible to adopt alternate strategies as regards the chaining quite easily. For example, if the linked lists hanging off each entry in the scatter index table tend to be long then the amount of time involved in searching through the lists may be a problem. In order to shorten the lists one approach might be to have two entries in the scatter index table for each pointer in the hash table; one pointing to a list of the integrand evaluations with odd valued keys leading to the given hash code and the other pointing to a list of the integrand evaluations with even valued keys leading to the given hash code. Or even four entries, since the searchkey is made up from two keys; even and odd lists for the first key and even and odd lists for the second key. Hence, pictorially the data structure



might be replaced by



or even



The author has only implemented the simplest strategy so far with the aim of investigating the feasibility of the approach. The following describes how the ideas have actually been used in the program.

The main requirement of the program is the evaluation of the integrand at particular nodes as the computation progresses. Hence the program has to either look up the value of the integrand evaluation in the scatter index table, if this particular node has been used before, or compute the integrand evaluation at this node and store the integrand evaluation at the appropriate place in the scatter index table. The process was written in the form of a procedure called "integrand evaluation" which takes as input parameters the node and the procedure describing the integrand and

delivers as it s result the integrand evaluation. The essence of the procedure consists of :

Compute the two keys for the given node;
Combine the two keys to give the searchkey;
Compute the hash value for this node;
{Consider the value stored in the hash table at the position given by the hash code}
IF the hash table entry is empty
THEN
    {the integrand evaluation has not been computed previously}
    compute the integrand evaluation and store it s value in the scatter index table;
    {deliver this integrand evaluation as the result of the procedure}
    alter the entry in the hash table at the position given by the hash code to be the position in the scatter index which contains the pointer to the start of the list in which the new integrand evaluation has just been stored
ELSE
    {the integrand evaluation may have been computed already}
    obtain the pointer to the list of integrand evaluations given by the entry in the scatter index indicated by the entry in the hash table at the position given by the hash code;
    search the list for an entry with the appropriate key;
    IF the key is found
    THEN
        deliver the integrand evaluation associated with this key as the result of the procedure

```
    ELSE
        {add a new node to the list of integrand evaluations}
        compute the integrand evaluation for this node;
        store this integrand evaluation in the new node;
        store the key for this integrand evaluation in the new
        node;
        deliver the integrand evaluation as the result of the
        procedure
    FI
FI
{End of the procedure}
```

In order to search the list a procedure is used that follows the same pattern as the one given in section 5.6, the only difference is that the key used as a search key is a LONG INT as opposed to an INT. Similarly a procedure analogous to the one given in section 5.7 is used to insert an item in the linked list. Full details of these two procedures are given in appendix [5.2].

The complete procedure to find the value of the integrand at a particular node either by looking it up or by evaluating it consists of :

```
PROC integrand evaluation = (REF[]REAL node,
                             PROC(REF[]REAL)REAL f)REAL :
BEGIN
    REAL val ;
    compute keys (key1,key2,node) ;
    newkey := evalkey (key1,key2) ;
```

```
    {form the long int search key}
    hash value := compute hash value (key1,key2) ;
    IF hash value < 0 OR hash value > 1023
    THEN hash value := 1
    FI ;
    REF INT htv = hash table [hash value] ;
    IF htv = 0
    THEN
       htv := next PLUS 1 ;
       val := f(node) ; {evaluate the integrand at this node}
       numeval PLUS 1 ;
       insert (scatter index table[htv],newkey,val)
    ELSE
       pointer := scatter index table [htv] ;
       IF searchlist(pointer,newkey)
       THEN val := fevalOFpointer
       ELSE
          val := f(node) ;
          {evaluate the integrand at this node}
          numeval PLUS 1 ;
          insert (pointer,newkey,val)
       FI
    FI ;
    val {deliver val as the result of the procedure}
END {of the procedure}
```

5.15 Computing an estimate to the integral over the subregions

Initially the compound trapezoidal rule and Ewing's rule were used

to compute the estimates to the integral over the subregions. These two rules have been discussed previously, however the procedures to evaluate the estimates have changed slightly. The procedures now use the procedure "integrand evaluation" described in the last section to find the value of the integrand at a particular node either by looking it up or by evaluating it. The procedure to evaluate an estimate to the integral of a subregion using the compound trapezoidal rule consists of the following :

```
Proc evala = (REAL div, scf, REF[]REAL centre,
              PROC(REF[]REAL)REAL f)REAL :
BEGIN
    REAL estimate := 0.0 ;
    FOR j TO num DO
    {for each node in the rule}
    BEGIN
        [1:n]REAL temp ;
        REF[]REAL t2 = nodes[j,1:n] ;
        {set t2 to point to the next node}
        {transform this node from the region over which the rule
        is defined to the current subregion and store the node
        thus formed in temp}
        FOR k TO n DO temp[k] := centre[k] + t2[k]/div ;
        compute keys (key1,key2,temp) ;
        estimate PLUS integrand evaluation (temp,f)
        {use the procedure integrand evaluation to find the value
        of the integrand at the new node}
    END ;
    estimate DIV scf ;
```

```
    {scale the estimate thus formed}
    estimate
    {deliver the estimate as the result of the procedure}
END {of the procedure evala}
```

The procedure to evaluate an estimate to the integral of a subregion using Ewing's rule consists of the following :

```
PROC evalb = (REAL div, scf, REF[]REAL centre,
              PROC(REF[]REAL)REAL f)REAL :
BEGIN
    REAL estimate := 0.0 ;
    {for each node in the rule}
    FOR j TO num DO
    BEGIN
        [1:n]REAL temp ;
        REF[]REAL t2 = nodese[j,1:n] ;
        {transform this node from the region over which it is
        defined to the current subregion and store the node thus
        formed in temp}
        FOR k TO n DO temp[k] := centre[k] + t2[k]/div ;
        estimate PLUS integrand evaluation (temp,f)
        {use the procedure integrand evaluation to find the value
        of the integrand at the new node}
    END ;
    estimate TIMES constl ;
    {multiply the estimate by the appropriate weight}
    {add the integrand evaluation at the centre of the subregion
    multiplied by the appropriate weight to the estimate}
```

```
    estimate PLUS (integrand evaluation(centre,f)*const2) ;
    {scale the estimate}
    estimate DIV scf ;
    {deliver the estimate as the result of the procedure}
    estimate
END {of the procedure}
```

5.16 The program using scatter storage techniques

The program using scatter storage techniques follows the same pattern as the program using linked lists to store the integrand evaluations. The main difference is the way in which the integrand evaluations are stored. In this program the scatter storage techniques described above are used whereas linked lists were used in the previous program. Full details of the complete program are given in appendix [5.2].

5.17 Testing the two approaches to storing the integrand evaluations

The program based on linked lists and the program based on the scatter storage techniques were tested on the set of test problems described in appendix [1]. The tables of results for the test runs are given in appendix [8].

5.18 Comparison between the two approaches

From the tables of results it can be seen that the program based upon the scatter storage techniques is marginally faster than the program based upon the linked lists. As the number of integrand

evaluations increases the difference between the two methods in terms of time increases. However, the hash code technique is more restrictive than the linked list method because of the way in which the keys are generated. As the number of dimensions increases the maximum level of subdivision allowed decreases. This could be altered by choosing a different representation for the keys. This is an area in which further research could take place if storing the integrand evaluations is justified.

5.19 Conclusions

Both of the programs produced reasonable results which satisfied the required tolerances in the majority of cases. Compared with the non storing methods the programs were somewhat slower but they did reduce the number of integrand evaluations actually evaluated considerably. Unfortunately the rules that are suitable for methods based on storing the integrand evaluations are less accurate than the rules that proved to be best in the non stored methods and consequently more integrand evaluations had to be computed using these methods to satisfy the same tolerances. If a high order pair of formulae with a minimum number of points could be found that produced nodes suitable for storing then the method could be improved considerably.

However, it can be seen from the test results that the overheads involved in storing the integrand evaluations would be justified if the integrand was very "expensive" to compute and a fairly accurate result was required; the stored method does not start to use less integrand evaluations than the non stored methods until the

subdivision process is well under way. For example consider the two dimensional version of the first test problem with a tolerance of 0.001. The basic adaptive method used 63 integrand evaluations and took 78 millunits whereas the hash code method used 225 integrand evaluations but only actually computed 49, the other 176 being reused, and took 559 millunits. Therefore the hash code method took 481 millunits longer but used 14 integrand evaluations less. Hence if the integrand had been more expensive to evaluate and 14 integrand evaluations had taken longer than 481 millunits then the stored method would have been faster. Similarly, with a tolerance of 0.00005 224 less integrand evaluations were used by the stored method but it took 1991 millunits longer. Therefore if it had taken longer than 1991 millunits to compute 224 integrand evaluations then the stored method would have faster. Some timings for the types of operations involved in computing an integrand evaluation are given in appendix [4].

In conclusion the methods adopted to store the integrand evaluations perform satisfactorily but the overheads involved make the programs impractical for anything but "expensive" problems. With further research the overheads might be reduced but this could only be justified if it can be demonstrated that there are significant number of sufficiently complex problems to be solved.

Chapter 6 Global Subdivision Strategies

6.1 Introduction

This chapter is concerned with the effects of different interval subdivision strategies. The algorithms described so far have all been based on what is termed a "local subdivision strategy". However, with respect to one dimensional adaptive quadrature Malcolm and Bruce Simpson [41] suggest that there are certain advantages in using a global subdivision strategy. They state that a global strategy can result in a reduction of the number of subregions used and an error in the final result which is closer to the required tolerance than is possible with a local strategy. Hence these ideas have been extended to multidimensional quadrature and examined.

The two terms local and global acceptance criterion are used exclusively with respect to adaptive quadrature schemes and can best be described as follows. An adaptive quadrature routine can be regarded as an algorithm for processing a sequence of subregions (Sn), the main components of which are :

a) a local quadrature procedure for evaluating an approximation to the integral over the subregion, that is

$$Q(Sn) \approx \int_{Sn} f(x_1 x_2 \ldots x_n)\, dx_1 dx_2 \ldots dx_n$$

b) a method for calculating an estimate to the error in the approximation, that is

$$E(Sn) \approx \left| Q(Sn) - \int_{Sn} f(x_1 x_2 \ldots x_n)\, dx_1 dx_2 \ldots dx_n \right|$$

c) criteria for deciding which subregion in the sequence (Sn) to subdivide at each stage and for deciding when to terminate.

The object of the algorithm is to produce a result, res, which is accurate to a user specified absolute error tolerance $\mathcal{E}$, that is

$$\left| res - \int_{S_n} f(x_1 x_2 \ldots x_n) dx_1 dx_2 \ldots dx_n \right| \leq \mathcal{E}$$

The subregions Sn fall into three distinct categories :

a) Sn has already been subdivided and discarded (n $\in$ dis)

b) Sn has been accepted (n $\in$ acc)

c) Sn is pending further examination (n $\in$ pen).

A program implementing an adaptive quadrature algorithm has to store the list of subregions pending further examination, (Sn $|$ n $\in$ pen) and to accumulate the contributions to the result from the accepted subregions, (Sn $|$ n $\in$ acc). Using a local error criterion a subregion is accepted once the error estimate for that subregion is less than the proportion of the absolute error tolerance given by the hypervolume of the initial region divided by the hypervolume of the subregion, that is

E(Sn) $\leq$ (h(Sn)$\mathcal{E}$ / (b-a))

where h(Sn) is the hypervolume of the subregion Sn.

This criterion has the following features :

a) the decision is based entirely on information available from Sn

b) if the error estimate E(Sn) is in fact a bound for the error in Q(Sn) for every n $\in$ acc when the algorithm terminates then the user's tolerance is guaranteed to be met (assuming exact arithmetic)

With the global error criterion all the subregions which have not

been discarded (discarded subregions are those which have been replaced by further subregions) are retained as pending and the sum of the error estimates for all the pending subregions being less than the absolute error tolerance is used as an acceptance criterion for the entire pending set, that is

$$\sum_{m \in pen} E(Sm) \leq \varepsilon$$

Then if this condition is not satisfied the subregion with the largest error estimate, $m \in pen$ such that

$$E(Sm) \geqslant E(Sn) \text{ for all } n \in pen,$$

can be subdivided and the resulting subregions added to the pending set in place of Sm.

Hence, the local error criterion decreases linearly with the subregion hypervolume and is most stringent as a tolerance in regions where the adaptive process is working at subdivision the hardest, whereas the global strategy selects subregions so that the local errors are roughly equal in magnitude, rather than scaled by the hypervolume of the subregions. Thus algorithms using global strategies should work no harder on subregions where the integrand is difficult to integrate than on subregions where it is "well behaved". Hence a global criterion has the potential for reducing the number of subregions used.

The subregion selection strategy does not affect the order of an adaptive quadrature algorithm but it does appear to influence it s performance in at least three ways:

1) by affecting the number of integrand evaluations required for an integral

2) by changing the domain of integrands which can be handled by the algorithm

3) by affecting the closeness with which the user's tolerance is achieved.

Rapid growth of the number of subregions and the corresponding numbers of integrand evaluations required are perhaps the key factors in multidimensional quadrature routines. Also most quadrature routines produce answers which are far more accurate than required by the tolerance, hence if the global strategy can reduce the number of integrand evaluations by producing a result which has an error closer to the required tolerance, without incurring too many overheads in terms of organisation then it has the potential of improving the efficiency of the algorithms.

6.2 A modification of the basic algorithm for the hypercube to use a global subregion strategy

In the basic algorithm for the hypercube a list was maintained of the non converged subregions of the initial region. This list was processed sequentially and a new list formed of the subregions of the non converged subregions of the first list. As convergence was achieved in a subregion the estimate thus formed was added to the total estimate so far and the subregion discarded from further consideration. In order to apply a global subdivision strategy it is necessary to keep a list of all the subregions and to order the list according to the size of the error estimate for the subregions. The subregion with the largest error estimate is at the start of the list while the subregion with the smallest error estimate is at the

end of the list. Hence the basic structure consists of :

```
BEGIN
    form the initial estimate over the whole region and the
    associated error estimate;
    set up a list consisting of this region ;
    set the total error estimate to be the error estimate for this
    region;
    set the estimate to the result to be the estimate over the
    whole region;
    WHILE not converged
    {i.e. the total error estimate > the required tolerance}
    DO
        consider the subregion with the largest error estimate,
        i.e. the one at the head of the list;
        subtract the estimate for this subregion from the estimate
        to the result;
        subtract the error estimate for this subregion from the
        total error estimate;
        subdivide this region into subregions;
        FOR each subregion
        DO
            compute the estimate for this subregion;
            add this estimate to the total estimate to the
            result;
            compute the error estimate for this subregion;
            add the error estimate for this subregion to the
            total error estimate;
            store the details of this subregion in a node;
```

```
        add this node to the linked list at the correct
        position, i.e. depedent upon the magnitude of the
        error estimate
      OD
      {now the estimate for the subregion with the largest error
      estimate has been replaced by the sum of the estimates
      over the subregions of that subregion]
  OD
END
```

With this approach regions are only subdivided into subregions when a new estimate over that region is required. It should be noted that in this algorithm, unlike with the basic algorithm for the hypercube, that the subregions which make up the linked list are not necessarily all at the same level of subdivision. Hence it is necessary to store details of the level of subdivision of each subregion in order to ensure that the correct transformation of the basic rule is applied. In the basic algorithm over the hypercube all that was required in each element of the list was the centre of the subregion and the pointer to the next item in the list. Rather more is required for the method based upon the global subregion strategy; namely the centre of the subregion, the scaling factor, the transformation factor (div), the error estimate for this subregion, the estimate over this region and the pointer to the next item in the list. Hence the following mode was defined:

```
MODE NODE = STRUCT(REF NODE ptr, REF[]REAL centre,
                   REAL errest, estimate, scf, div)
```

6.2.1 Adding a node to the list of subregions

In the basic program the new nodes corresponding to new subregions were quite simply added to the start of the linked list. However with the global strategy approach it is necessary to maintain the list in descending order of the error estimates. Hence the insertion of a new node in the list involves a search for the correct position of insertion. A straight linear search from the start of the list was adopted.

The search and insertion process consists of :

```
start at the head of the list, set a temp ptr to the head;
IF the list is empty
THEN
    set the head of the list to point to the new node;
    set the ptr of the new node to be nil, i.e. empty
ELSE
    WHILE the error estimate of the temp ptr is greater than
          the error estimate of the new node
      AND the ptr of the temp ptr ISNT nil
    DO
        move the temp ptr on to the next item in the list
    OD;
    {now the first item in the list whose error estimate is smaller
    than the error estimate for the new node has been found}
    IF the ptr of temp ptr is nil
      AND the error estimate of the temp ptr is greater than the
    error estimate of the new node
    THEN
```

{the end of the list has been reached}

set the ptr of the new node to be nil;

set the ptr of temp ptr to point to the new node;

{i.e. add the new node to the end of the list}

ELSE

{add the new node before the item pointed to by temp ptr}

set a temporary node equal to the new node;

set the new node equal to the node pointed to by temp ptr;

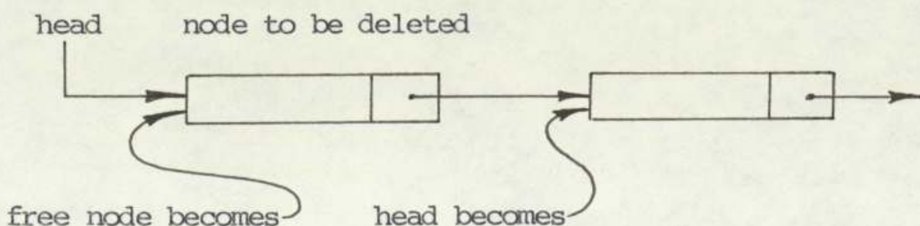
set the ptr of temp ptr equal to the new node;

{in effect this adds the new node after the node pointed to by temp ptr but changes the data of the two nodes}

FI

FI

As well as adding new nodes to the list it is necessary to remove the first item from the list. This is achieved by advancing the head of the list to the second item in the list. Since the head of the list is continually removed and replaced it was essential to make use of the discarded node so as to avoid using the heap too rapidly. This was achieved quite simply by having a variable which could be pointed to the node to be deleted from the head of the list before advancing the head of the list then this free node was used the next time a new node was required, instead of creating a new node on the heap.

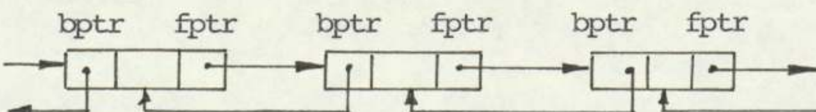


6.2.2 Adopting a doubly linked list to store the details of the subregions

This section presents one alternative method of accessing the integrand evaluations. In the previous section storing the subregions in a simple linked list was considered. However, consider the size of the error estimate for the new subregions. In the majority of cases they will be smaller than the error estimates for the existing subregions (otherwise the overall error estimate would not be decreasing and the method would not be converging). Hence in these cases adopting the simple approach described above would result in the list being searched to or very near to the end passing through all or almost all the nodes in the list on the way with the associated comparisons of error estimate. Obviously it would seem reasonable under the circumstances to start the search at the end of the list and work towards the beginning. Then only in the exceptional cases would a large proportion of the list be searched in order to find the position of insertion of the new node. In order to achieve this it is necessary to alter the structure of the list slightly so as to enable movement up or down the list. A second pointer has to be added to each node to point backwards to the previous node in the list. Each node in the list is of the following mode:

```
MODE NODE = STRUCT(REFNODE fptr, bptr, REF[]REAL centre,
                   REAL errest, estimate, scf, div)
```

where fptr represents forward pointer and bptr represents backward pointer. The structure of the list is of the form:



This alters the part of the algorithm which is involved with finding the position of insertion of the new node and inserting the new node. Adopting the double linked list and starting the search from the end of the list the search and insertion process consists of:

```
Start at the tail of the list
IF the list is empty
THEN
    set the head of the list to point to the new node;
    set the tail of the list to point to the new node;
    set both the forward and backward pointers of the new node to
    be empty, i.e. NIL
ELSE
    WHILE the error estimate of the tempptr
          is less than the error estimate of the new node
      AND the backward ptr of tempptr ISNT NIL
    DO
      move the tempptr back to the previous item in the list
    OD;
    {i.e. find the last item in the list whose error estimate is
    greater than the error estimate of the new node, i.e. the
    position of insertion}
    IF the backward ptr of tempptr IS NIL
       AND the error estimate of the tempptr
          > error estimate of the new node
    THEN
        {the start of the list has been reached and the new node
        needs to be inserted at the head of thhe list}
        set the forward pointer of the new node to be head;
```

```
        set the backward pointer of head to be the new node;
        set head to be the new node;
        set the backward pointer of the new node to be nil
    ELSE
        {add the new node after the item pointed to by temp ptr}
        backward pointer OF new node := temp ptr;
        forward  pointer OF new node := forward pointer OF temp ptr
        ;
        backward ptr OF forward ptr OF temp ptr := new node ;
        forward ptr OF temp ptr := new node
    FI
FI
```

Since it is likely that the magnitude of the error estimates over the subregions ofa particular subregion will be similar it may be more efficient to start the search from the last position of insertion and search either forwards or backwards as required. This involves both the search and insert process described above and the search and insert process described in the last section. Essentially the two processes can be combined in the following manner assuming temp ptr is the last position of insertion of a new node:

```
IF the error estimate of the temp ptr is greater than
   the error estimate of the new node
THEN
    search forward until an item is found with a smaller error
    estimate than that of the new node;
    insert the new node prior to this node
    {this is the process described in the last section}
ELSE
```

search backwards until an item is found with a larger error estimate than that of the new node;

insert the new node after this node

{this is the process described above}

FI

The double linked list is just one alternative to the single linked list that could possibly improve the performance of the algorithm. Another alternative might be to adopt a tree structure. These alternatives are worthy of further research once the global subdivision strategy has proved to be an improvement on the locel strategy.

6.3 The complete program

The program uses the same two basic rules adopted for the local method; namely Stroud's n+1 and 2n point rules. The program is written as a sequence of segments, full details of which are given in appendix [6].

6.4 testing the program

The program was tested using the set of test problems defined in appendix [1] and the tolerances 0.5,0.1,0.5,..etc. The maximum jobtime was limited to 90 and the maximum core size to 90k. The results for the test runs are given in appendix [8].

6.5 Conclusions

The majority of results produced using this method were more accurate than the requested tolerance and the times to produce the results compared favourably with the other methods. In particular the program consistently used less integrand evaluations than the basic local adaptive method and correspondingly less time to produce results that were closer to the required tolerance. Also the change in the number of integrand evaluations used for each tolerance was quite smooth, unlike with the local method where sudden dramatic increases occur as the method progresses from one level of subdivision to the next. Hence, it would appear from the results so far that the global subdivision strategy is preferable to a local subdivision strategy in that it uses less integrand evaluations and the results produced are close to the required tolerance. This program demonstrates that it is possible to adopt a global subdivision strategy without the benefits being lost due to the overheads in implementing the technique.

Further research into storing and accessing the list of subregions could perhaps improve the program further. The program does use a large amount of store in the form of the heap and any reductions would be an improvement.

One problem with this method is that the list of subregions can get very long since all the subregions are retained. One solution might be to set a maximum limit on the length of the list and once this limit is reached begin to discard subregions from the end of the list; that is begin to discard the subregions which have the smallest error estimates. This should not affect the overall strategy if the list is long enough since the number of subregions

actually discarded will be kept to a minimum and subregions will only start to be discarded when the program is close to obtaining the final solution.

In conclusion adopting a global subdivision strategy would appear from the work so far to be beneficial in multidimensional quadrature programs.

Chapter 7 Extension of the methods to other regions

7.1 Introduction

In the previous chapters programs have been developed to approximate integrals over two basic regions; the hypercube and the simplex. Unfortunately the vast majority of practical problems involve rather more complex regions which cannot be transformed easily, if at all, into either of these two regions. However, in many cases it is possible to define a region as the union of a series of subregions, each of which is either a simplex or a hypercube. Under such circumstances it would be feasible to use the programs of the previous chapters to find estimates to the integral over each of these subregions and to sum the results to give an estimate over the whole region. If the final approximation was not sufficiently accurate then the whole process would have to be repeated. This could be very tedious for a complicated region even though the process is relatively straightforward. This chapter is concerned with extending the methods developed so far in order to automate this process.

7.2 Extension of the region of integration

The methods developed for the two basic regions in the previous chapters were based upon the use of a linked list to store the subregions prior to processing. As the methods progress each subregion in the list is either removed from the list because convergence has been achieved in it, in which case the estimate over

that subregion is added to the final estimate, or it is replaced by a list of it s subregions. Consider the list of subregions generated by one of the previous programs. The following applies :

1) At any stage the starting region is defined by the linked list of subregions and possibly a total estimate over the converged subregions

2) The estimate to the integral over the whole region is given by the sum of the estimates to the integral over each of the subregions in the linked list plus the total estimate over the converged subregions.

3) Each of the subregions in the list has the same form, either a simplex or a hypercube, as the original region and a simple transformation of the basic rule is applied to give the estimate over the subregion.

Obviously, 1) and 2) above are essential if the final estimate is to be valid. However, 3) can be relaxed in two ways :

a) it is not essential that the same rule is applied over each subregion, provided an approximation over the region can be formed that is sufficient

b) the subregions need not be of the same form as the original, it is sufficient that the sum of the subregions is equivalent to the whole region.

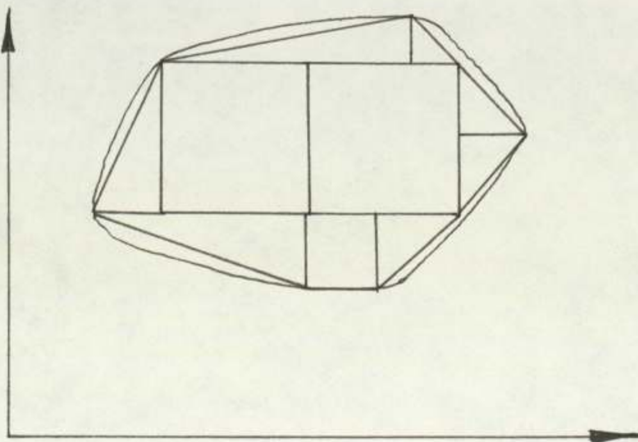
These two relaxations mean that the region defined by the list of subregions need not be a region of a particular shape, since it is sufficient that the sum of the subregions is equivalent to the whole region, and that the subregions could be of any form, so some could be simplexes and some could be hypercubes. Hence this allows the same basic approach to be taken for a region which is not itself either a hypercube or a simplex but which can be defined as a union

of subregions, each of which is either a hypercube or a simplex. It is valid to form the estimate over the region by taking the sum of the estimates over the subregions which make up the union because the basic properties of linearity for multiple integrals apply. In order to develop a program upon this type of region, which is an extension of the regions used in the previous algorithms, it was necessary to amalgamate parts of both the program for the hypercube and the program for the simplex.

7.3 Subdivision of a region into a union of simplexes and hypercubes

The problem of subdivision of a region into a union of simplexes and hypercubes is beyond the scope of this thesis. However this section outlines some of the difficulties involved in the subdivision process and attempts to justify the development of an algorithm based on the simplex and hypercube as subregions by showing that they are useful forms for the subregions.

Consider the subsdivision of an arbitrary two dimensional region of integration into a series of subregions :



It can be seen that it is possible to subdivide the region into a union of squares and triangles and that the area covered by the union can be made as close to that of the whole region as required by taking sufficiently small subregions. The simplex is well suited to subdividing curved edge boundaries while the hypercube is well suited to filling large areas of the region.

There are an infinite number of ways in which the original region could be subdivided and various questions concerning the subdivision process arise immediately :

1) how many subregions should be used ?

2) should uniform or non uniform size subregions be used ?

3) how closely does the border of the union of subregions need to follow the border of the initial region in order to give a satisfactory result ?

4) should one type of subregion be used ? For example is it better to split the squares into two triangles ?

5) should the sides of the squares be parallel with the axes ?

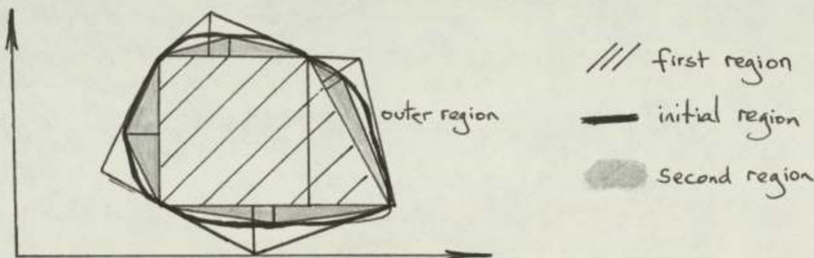
6) is the orientation of the triangles important ?

All of these questions require further research in order to answer them fully. However, from the point of view of writing an algorithm some guidelines can be given as regards some of them. Since the algorithm will progress by automatically subdividing the subregions into still more subregions and the minimum number of integrand evaluations used is determined by the number of subregions in the initial list it is preferable to use as few subregions as possible to define the region. This implies that it is better not to

subdivide hypercubes into simplexes, since although each square can be divided into only two triangles any cube has to be divided into at least six simplexes and the ratio increases as the number of dimensions increases. However, that being the case, if the behaviour of the integrand is known or is suspected to be particularly "bad" in one or more specific areas of the region then it may be a good idea to make the subregions in these areas somewhat smaller. This has the potential for saving some work by the algorithm which would have to find the difficulty otherwise by the subdivision process. If the user knows that at least a certain level of subdivision will be required in order to give the required accuracy then it may be a good idea to use sufficient subregions initially to reflect this level so as to avoid all the computation that would be involved in reaching this level. One advantage of defining the region as a list of subregions is that the user has some influence over the behaviour of the algorithm; by his choice of subregions the user can force the algorithm to use more or less integrand evaluations initially in particular parts of the region. It has been suggested by Lyness [34] that automatic quadrature routines do not allow the user to think and that this is the wrong approach. In definingthe initial region as a linked list of subregions the user is encouraged to think and to take advantage of his knowledge so as to improve the performance of the method.

The question of how closely the border of the union of subregions has to follow the border of the initial region in order to give a satisfactory result is perhaps the most important question. One possible approach is to define two unions to describe the region; one circumscribing the region and one inscribing the region. Then

the approximations over these two unions can be considered as upper and lower bounds on the required estimate. Obviously if the difference between the two is not acceptable then further subregions need to be added so that the unions are closer approximations to the original region. This process could be very expensive and one alternative is to define three unions; the first covering the majority of the region and inscribing the region, the second having it s inner border common with the outer border of the first union and it s outer border inscribing the initial region, and the third also having the common border with the first but having it s outer border circumscribing the initial region. Thus for example:



Then in order to reduce the difference between the upper and lower bounds it is only necessary to redefine the second two unions and compute new estimates to the integral over these regions. The integral over the first union need not be recalculated, but the estimate to it can be added to the estimates over the other two regions to give the overall estimates.

If the subregions which are hypercubes are chosen so that the sides of the hypercubes are parallel to the axes then the transformation of the basic rules is simple and the methods used in the previous algorithms can be used with only slight modification. The orientation of the simplexes is unimportant since the rules can be defined in terms of the vertices.

The same ideas can be extended quite naturally to n dimensions, where the subdivision process becomes complicated.

Hence it is possible to subdivide a region into a union of simplexes and hypercubes. These two subregions lend themselves quite naturally to the subdivision process and using the two in combination would appear quite useful. Thus there is some justification for developing a method based upon a combination of these subregions.

7.4 The basic structure of an algorithm

If it is assumed that the region of integration can be defined as the union of a set of subregions, each of which is either a simplex or a hypercube, then the basic structure of an algorithm to form an estimate to the integral of a function over that region can follow quite closely the structure of the algorithms described previously. Once the linked list of subregions has been formed the method proceeds by continually subdividing each subregion in the list into more subregions until finally an estimate is formed which satisfies the required tolerance. The author chose to adopt a global subdivision strategy from the outset with this algorithm since the list of subregions defining the region of integration could be large to start with in which case a local strategy would result in a very rapid increase in the number of integrand evaluations at each level of subdivision.

Hence the basic structure of the algorithm consists of :

{Form the initial estimate over the entire region}

FOR each subregion in the linked list DO

```
BEGIN
    compute  the  estimates  over  this  subregion  using  suitable
    transformations of the appropriate rules;
    evaluate the error estimate for this subregion;
    {ie. the difference between the two estimates computed }
    store  the details of this subregion and the estimates in a list
    element;
    add  this  list  element to the ordered list of subregions, at a
    position dictated by the magnitude of the error estimate
    {largest at the start, smallest at the end}
    add  the estimate over this subregion to the total estimate over
    the whole region;
    add  the  error  estimate over this subregion to the total error
    estimate over the whole region
END ;

{Now  an  ordered  list  of  subregions  has  been  formed  with  the
subregion  with  largest  error  estimate at the head of the list and
the  subregion  with  the  smallest error estimate at the tail of the
list}
{calculate further estimates until convergence is achieved}
WHILE NOT converged DO
 {ie. the total error estimate is greater than the tolerance}
BEGIN
    {consider the subregion at the head of the list}
    subtract  the  error  estimate for this subregion from the total
    error estimate;
    subtract  the  estimate  for  this  subregion  from  the  total
    estimate;
```

{ie. remove the contributions for this subregion from the running totals, these will now be replaced by the contributions from the subregions of this subregion}

subdivide the subregions into the appropriate number and type of subregions;

FOR each subregion DO

BEGIN

compute the estimates over this subregion using suitable transformations of the appropriate rules;

evaluate the error estimate for this subregion;

add the estimate and the error estimate for this subregion to the running totals for the entire region;

store the details of this subregion and the estimates in a list element;

add this list element to the ordered list of subregions, at a position dictated by the magnitude of the error estimate

END

END ;

{convergence achieved}

output the required results.

Some parts of this algorithm have been discussed previously, the rest will be explained in more detail in the following sections.

7.5 Defining the original region as a linked list

The original region has to be defined as a linked list of subregions, each of which is either a hypercube or a simplex. Each

element of the linked list must contain sufficient information to allow an estimate to be formed over that subregion. The information used in the previous programs to represent the two regions is now discussed.

All the rules for the hypercube are given with respect to a particular starting region ($-1<=x_i<=1$ $i = 1,2,...n$ in this thesis) and in order to form an estimate over an alternative hypercube the nodes of the rule have to be mapped on to the new hypercube and the resulting estimate scaled accordingly. In the previous program the procedures which evaluate the estimates over the subregions have the following parameters : "centre, div, scf, alt". These parameters are sufficient to allow the mapping to take place and the transformed rule to be applied correctly. The parameters indicate :

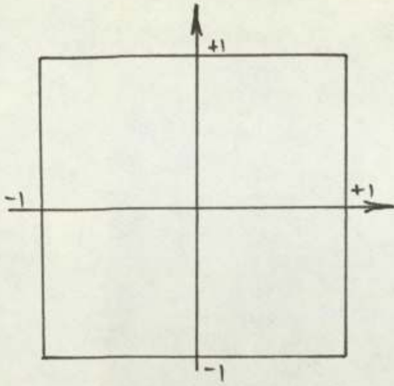
centre - the centre of the subregion to which the rule is to be mapped

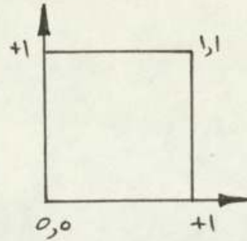
div - the factor relating one side of the original hypercube to one side of the subregion, this is the factor used to scale the nodes from the original region to the subregion

scf - the scaling factor relating the hypervolumes of the initial hypercube and the subregion

alt - the offset of the subregions of the subregion, this is equal to one quarter of one side of the subregion.

For example consider the values of the parameters that would be used to transform a rule from the region $-1<=x_i<=1$ $i = 1,2,...n$ to the subregion $0<=x_i<=1$ $i = 1,2,...n$ for a two dimensional problem.





centre = 0.5, 0.5

div = 2 = side of the original/ side of the subregion

scf = 4 = hypervolume of the initial region/ hypervolume of the subregion

alt = 0.25 = side of the subregion/ 4

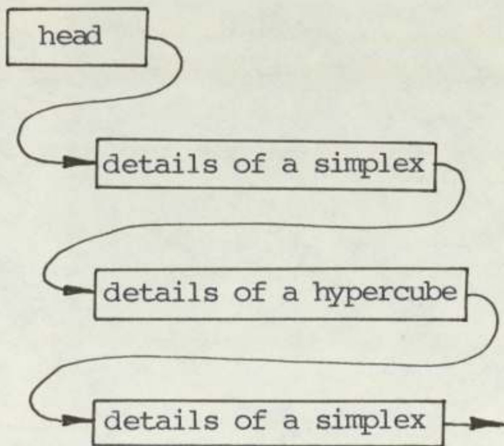
In the linked list defining the subregions for the hypercube program some of these parameters were common to all the subregions at one level and so were dropped. However with this program all the parameters could be different and need to be stored for each element in the list.

The rules for the simplex are often given with respect to a starting simplex, however the nodes can usually be transformed so that they relate to the vertices of the simplex. Hence in the simplex programs the procedures which evaluate the estimates over the subregions have the vertices as one parameter and the centroid and hypervolume as two other parameters, since these are used in many rules. Thus in order to describe a simplex adequately, an element of the list must contain : the vertices of the simplex, the hypervolume and the centroid. In fact the centroid need not be stored since it is computed from the vertices.

So in order to define an original region as a linked list it is necessary to subdivide the region into a number of subregions; for each subregion which is a hypercube the appropriate "centre, div, scf and alt" must be defined and for each subregion which is a simplex the vertices and hypervolume must be defined.

7.6 Storing the linked list

In the previous programs all the elements of each of the linked lists contained either details of simplexes or details of hypercubes. With this extended program the list could contain details of both types of subregion at once. That is the list must be able to take the form :



Thus the pointer of an element is required to point to either an element suitable to hold the details of a simplex or an element suitable to hold the details of a hypercube. In order to define a simplex an element has to contain :

a) the vertices of the simplex

b) the hypervolume of the simplex

c) an estimate for this region

d) an error estimate for this subregion

e) a pointer to the next element in the list

while in order to define a hypercube an element has to contain :

a) the centre of the hypercube

b) div, the factor used to scale the nodes

c) scf, the scaling factor

d) alt,the offset of the centres of the subregions of this subregion

e) an estimate for this subregion

f) an error estimate for this subregion

g) a pointer to the next element in the list.

It was useful to define two new modes to describe these list elements. Suppose a new mode describing a pointer to a simplex or a hypercube existed :

```
 MODE NEXTITEM
```

then the two modes required could be defined as :

```
MODE SIMPLEXITEM = STRUCT(REF[,]REAL vertices,
                          REAL    hypervolume,    estimate,    error
estimate,
                          NEXTITEM ptr)
```

and

```
 MODE HYPERCUBEITEM = STRUCT(REF[]REAL centre, REAL div, scf, alt,
                             estimate, error estimate,
                             NEXTITEM ptr) .
```

The mode NEXTITEM has to be defined so that it can be either a reference to a SIMPLEXITEM or a reference to a HYPERCUBEITEM, depending upon the type of the next subregion in the list.

Fortunately this can be achieved by the use of a union in algol68. Hence the mode NEXTITEM was defined as :

```
MODE NEXTITEM = UNION(REF SIMPLEXITEM, REF HYPERCUBEITEM)
```

Before these two new modes can be used to create space in which to store the list it is necessary to input the appropriate data. The author chose the following format for the input data defining the subregions in the list :

```
type of subregion - simplex or hypercube
.
.
.  data for this subregion
.
.
type of subregion
.
.
.  data for this subregion
.
.
end of list marker .
```

Hence the basis of the algorithm to input and store the linked list consists of the following :

```
WHILE (input the type of subregion ;
     type is not equal to the end of list marker) DO
BEGIN
     IF type = simplex
     THEN {the subregion is a simplex}
          create a new simplex list element ;
```

```
        input the data for this simplex ;
        store the data in the new element ;
        compute the estimates for this subregion ;
        evaluate the error estimate ;
        store the estimate and the error estimate in the new list
        element ;
        add the new element to the sorted list at a position
        according to the magnitude of it's error estimate
    ELSE {the subregion is a hypercube}
        create a new hypercube list element ;
        input the data for this hypercube ;
        store the data in the new element ;
        compute the estimates for this subregion;
        evaluate the error estimate ;
        store the estimate and the error estimate in the new list
        element ;
        add the new element to the sorted list
    FI
END
```

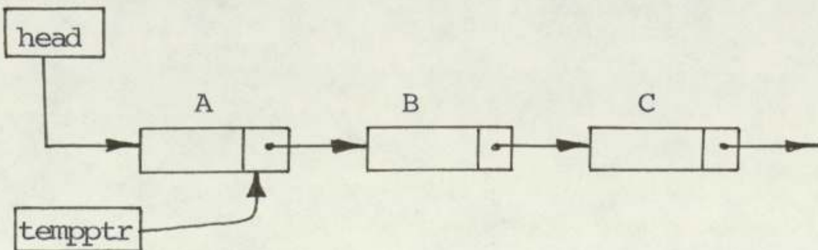
In order to add a new element to the sorted list at the correct position it is necessary to search through the list to find the position of insertion and then to actually insert the new element at this position. This process was written in the form of the procedure "add to list" which consists of :

```
PROC add to list = (REAL e, REF NEXTITEM newnode) VOID :
BEGIN
    REF REF NEXTITEM tempptr := head ;
```

```
{set a temporary pointer to point to the start of the list}
REF SIMPLEXITEM s ;
REF HYPERCUBEITEM h ;
IF head ISNT end of list
   {where REF NEXTITEM end of list = NIL}
THEN {the list isnt empty,
      search for the correct position of insertion}
      BOOL notfound := TRUE ;
      WHILE notfound DO
      BEGIN
           CASE (s,h) ::= tempptr
           IN
            (IF errest OF s > e
             THEN tempptr:= ptr OF s ;
                  {move on to the next item in the list}
                  IF ptr OF s IS end of list
                  THEN notfound := FALSE
                       {end of the list reached}
                  FI
             ELSE notfound := FALSE
                  {position of insertion found}
             FI),
            (IF errest OF h > e
             THEN tempptr := ptr OF h ;
                  {move on to the next item in the list}
                  IF ptr OF h IS end of list
                  THEN notfound := FALSE
                       {end of the list reached}
                  FI
```

```
                ELSE notfound := FALSE
                    {position of insertion found}
                FI)
              ESAC
          END
      FI ;
      {position of insertion found}
      CASE (s,h) ::= newnode
      IN
       (ptr OF s := tempptr ),
       (ptr OF h := tempptr )
      ESAC ;
      {the newnode now points to the next item in the list}
      (REF REF NEXTITEM VAL tempptr) := newnode
      {the previous item in the list now points to the newnode}
  END
```

In the procedure tempptr points to the pointer field of the~~e~~ previous element in the list, thus simplifying the insertion process.



When considering the data of node B tempptr is pointing to the pointer field of node A. Therefore insertion consists of making the pointer field of the newnode equal to the pointer field pointed to by tempptr, i.e. linking the newnode to the node B, and making the pointer field pointed to by tempptr point to the newnode, i.e.

linking node A to the newnode.

8.7 Processing the list of subregions

Once the initial list of subregions has been formed with the subregion with the largest error estimate at the head of the list and the subregion with the smallest error estimate at the tail of the list, this list must be processed. That is, the contribution made by the subregion at the head of the list to the total result must be replaced by the sum of the contributions from the subregions of this subregion. Also the list element defining this subregion must be removed from the list and replaced by the list elements defining it s subregions. This process is complicated somewhat by the fact that the list elements describe either simplexes or hypercubes.

Essentially the process consists of :

```
{consider the subregion at the head of the list}
IF the subregion is a simplex
THEN
     remove the contributions for this simplex from the running
     totals;
     subdivide the subregion into two further simplexes;
     FOR each simplex DO
     BEGIN
          compute the estimates for this subregion;
          add the estimates to the running totals ;
          store the details of this subregion in a simplex list
          element;
          add this element to the list of subregions
```

```
     END
ELSE {the subregion is a hypercube}
     remove  the  contributions  for  this hypercube from the running
     totals;
     subdivide the region into 2^n hypercubes;
     FOR each hypercube DO
     BEGIN
          compute the estimates for this subregion;
          add the estimates to the running totals;
          store  the  details  of  this subregion in a hypercube list
          element;
          add this element to the list of subregions
     END
FI ;
move on to the next element in the list.
```

Moving on to the next element in the list involves setting the head of the list to be the pointer of the head. This operation depends upon the type of subregion described by the element at the head of the list and so is best refined as two distinct operations in the algorithm above. In order to decide the type of subregion described by the head of the list it is necessary to deunite the list element describing the subregion at the head of the list. This is achieved with the aid of a collateral conformity clause in conjunction with a case construction. Adopting this approach results in the following:

```
REF SIMPLEX ITEM s ; REF HYPERCUBE ITEM h ;
{these are temporary pointers, one of which will be set dependant
upon the type of subregion}
CASE (s,h) ::= head
IN
```

({s has been set, the region is a simplex}

process the simplex as above ;

set head to be the pointer of the simplex item),

({h has been set, the region is a hypercube}

process the hypercube as above;

set head to be the pointer of the hypercube item)

ESAC

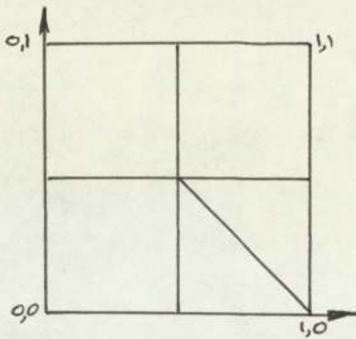
This process is repeated until the required tolerance has been satisfied.

7.8 The complete program

The estimates over the subregions are computed using the basic rules discussed previously. For subregions which are hypercubes Stroud's rules are used, while for subregions which are simplexes Stroud's and Lauffer's rules are used. The complete program consists of a sequence of segments which are linked by the main body of the program. A full description of all the segments and the main body of the program is given in appendix [7].

7.9 A simple test problem

As an illustration of the program a hypercube was subdivided into a number of subregions, each of which was either a hypercube or a simplex. Then the extended program was used to compute an estimate over this union of subregions. The following is the initial region and the chosen subdivision into subregions:



Hence the data defining the union consisted of:

2 the number of dimensions of the problem

H the subregion is a hypercube

0.25 0.25 the centre of the hypercube

16, 4, 0.125 scf, div and alt

H the subregion is a hypercube

0.25 0.75 the centre of the hypercube

16, 4, 0.125 scf, div and alt

H the subregion is a hypercube

0.75 0.75 the centre of the hypercube

16, 4, 0.125 scf, div and alt

S the subregion is a simplex

0.5,0.5 1.0,0.5 1.0,0.0 v

0.125 the hypervolume of the simplex

S the subregion is a simplex

0.5,0.5 1.0,0.5 0.5,0.0 v

0.125 the hypervolume of the simplex

E end of data marker

The program was tested using the following problem:

$$\int_0^1 \int_0^1 \mathrm{sqrt}(x+y)\, dx\, dy$$

The results were more accurate than the requested tolerance but "expensive" in terms of integrand evaluations. However, this problem is intended as an example of how the program is used and would not be a realistic problem for solution using this method.

7.10 conclusions

This chapter has demonstrated one way in which the methods developed for the simplex and the hypercube can be extended to cope with other types of region of integration. If a region can be defined as a union of subregions, each of which is either a simplex or a hypercube, then this is a feasible approach. However the problem of subdivision of the initial region requires careful consideration if the method is to be reliable and efficient.

One limitation with this approach is the large number of subregions that could exist at any one time. One solution to this problem might be to set an upper limit on the number of subregions that are held in the list. Once this limit has been reached the program could save a part of the list on backing store and continue to process the rest of the list. Once a satisfactory estimate over this part of the list had been achieved then the other part of the list could be reinstated in place of the present part and an estimate formed over this part. Then the sum of the two estimates would give the required estimate. Alternatively it would be possible to adopt a policy of discarding subregions from the end of the list once the upper limit on the number of subregions had been reached. Obviously the estimates from the discarded subregions would have to be added to a running total estimate to the final result.

The method as described in this chapter applies one procedure to the hypercube based on a certain pair of rules and a second procedure to the simplex based on a different pair of rules. However, there is no reason why more than one procedure for each region could not be included and a choice made between the available procedures

dependent upon the behaviour of the integrand in the present subregion; the behaviour of the integrand over a particular subregion could be supplied by the user when defining the union of subregions, if he knew it or left blank otherwise. For example a procedure could be included which applied the product Patterson rules to a hypercube. Then if the user knew that the integrand was either uniformly "well" behaved or uniformly "badly" behaved over one or more of the subregions he could supply the information to the program to ensure that the product Patterson procedure was applied over those subregions. Alternatively if the user knew that a different procedure was more applicable to a particular subregion then he could supply the information to ensure the correct choice of procedures for those subregions. Thus the user could be allowed to influence the performance of the method by taking advantage of his knowledge of the integrand. Another way in which the user may be allowed to supply information would be to include the possibility of the initial estimates over all or some of the subregions to form part of the input data. This could also be use to allow the user to restart the program in order to obtain a more accurate result if the list of subregions was output in some form at the end of the computations.

Chapter 8 Multiprocessor Techniques

8.1 Introduction

The methods described so far are very processor intensive and thus limited by the capabilities of the machine on which the programs are run. Obviously one way in which the performance of the methods can be improved is by using a more powerful, faster machine. Until recently the enhancement of computer performance has come from a refinement of the basic Von Neumann architecture and the improved performance of semiconductor components. With the rapid development of LSI technology and the corresponding fall in processor costs there has been a trend towards multiprocessor architectures offering both parallel and concurrent processing capabilities. With a suitable problem the new types of architecture can result in a considerable improvement in the performance of an algorithm. The author considers the problem of multidimensional quadrature to be well suited to solution on a machine with this type of architecture and hence presents a case for their adoption for multidimensional quadrature programs. Since a suitable machine was not at the disposal of the author some of the theoretical possibilities have been considered.

8.2 The architecture of a multiprocessor

Early computers were classified as serial or parallel depending upon the design of the arithmetic and logic unit. However the concept of parallelism has been extended to cover any set of operations carried out in parallel and may occur at any logical or physical level of

the system. Flynn has suggested that it is now appropriate to view architectures in terms of the instruction stream and the data stream. Multiplicities in these streams lead to four basic alternative architectures:

1) single instruction single data - SISD

2) single instruction multiple data - SIMD

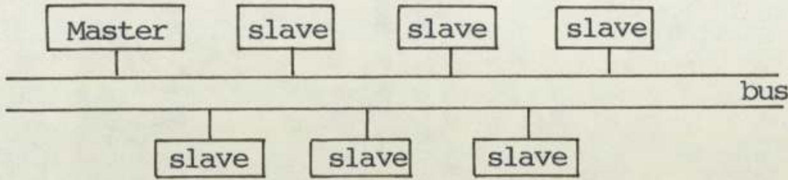
3) multiple instruction single data - MISD

and 4) multiple instruction multiple data - MIMD.

The ideas proposed in this chapter are based upon the attributes of a machine with a multiple instruction stream and a multiple data stream type architecture. That is a machine that must have more than one processor. Multiple processor systems exhibit a diversity of architectures and it is often the case, particularly with multiple microprocessor systems, that the architecture has been designed for a specific task. Frequently the motivation for using multiple processors does not come from throughput considerations but from the many other advantages of distributing both processing power and intelligence, such as modularity, reliability, response time to human interaction, resource sharing and fault tolerance. However the motivation for the adoption of this type of architecture in the context of multidimensional quadrature algorithms is solely an improvement in performance. Hence the overheads involved in coordinating the processors must be weighed against any possible advantages.

{Perhaps it should be stressed that a multiple processor architecture, i.e. a MIMD type architecture, as opposed to an array processor type architecture, i.e. a SIMD type architecture, is being considered. The latter being the type normally associated with speeding up numerical calculations.}

The architecture could be described as a master, slave system where one processor is deemed to be the master in control of distributing tasks to the other processors which are considered to be slaves of the master. All communication between the master and the slaves can take place via a suitable bus.



Each processor has it s own memory and a main memory could be available as part of the master or as a separate entity also connected to the bus. This is perhaps the simplest interconnection topology that could be used and relies on the use of a global bus. This obviously creates the problem of simultaneous requests for the bus and so some form of arbitration is required. Arbitration is merely the name given to the process of deciding which processor obtains exclusive use of the bus. A full introduction to the basic concepts of multiple processor systems is given in Bowen and Buhr [3]. Although this book is mainly concerned with microprocessors the ideas are valid regardless of the size or power of the processors used.

8.3 Multidimensional quadrature as a suitable task for solution on a multiple processor system

When using a multiple processor system as opposed to a single processor system an improvement in the performance of an algorithm can only be expected if the algorithm exhibits some inherent concurrency. That is if the algorithm can be split into a number of

tasks which can be distributed amongst the available processors and processed concurrently so as to take advantage of the available processing power. If the algorithm is strictly sequential in nature then no improvement can be expected.

Consider the nature of the multidimensional quadrature algorithms. In each case the method proceeds by subdividing a region into a number of subregions, forming the estimates over each of these subregions and summing these estimates to give a total estimate over the entire region. The natural concurrency involved in forming the estimates over the subregions has to be removed and a sequential order imposed when adopting the algorithm on a single processor system. Hence it seems reasonable to assume that an improvement in performance of the multidimensional quadrature algorithms can be expected if a multiple processor system is adopted provided the overheads involved in distributing the tasks are not too high. The calculation of the estimates over the subregions of a region or even the entire list of subregions could be processed concurrently.

8.4 An algorithm for use on a multiple processor system

If it is assumed that a machine with an architecture of the type described in the last section is available, where each processor in the system has sufficient power to perform the subdivision process and to generate estimates over the subregions, then it would be feasible to construct an algorithm for multidimensional quadrature for that machine. All the algorithms written previously exhibit some natural concurrency which could be exploited on a multiple processor system. However they all have some drawbacks from this point of

view. Consider the first basic adaptive method as described in 4.3. The method involved processing a list of subregions sequentially. Obviously the calculations for each subregion are not connected and could be performed in parallel, or at least concurrently. However the method could not proceed to the next level of subdivision until the entire list at one level had been processed. Since the list grows to a long length before convergence begins to take place in various subregions it is not feasible to have sufficient processors to perform the calculations for all the subregions at the same time. Hence towards the end of an iteration the situation could arise where many of the processors were idle while only one or two of the processors performed the calculations for the last regions. This might not be a serious drawback but could degrade the overall performance of the system and is not an efficient way of using the processors. In the worst case the number of subregions would be one larger than the number of processors, which would result in all the processors but one waiting while the calculations for the last subregion were performed.

Alternatively consider the global subdivision strategy. With this method the algorithm proceeds by dealing with one subregion at a time. The only obvious benefit of using a multiple processor system in this situation is if the tasks of calculating the estimates over the subregions of the subregion are performed concurrently. The drawback then is the sequential nature of the tasks of processing the list, forming the subregions and checking for convergence. All the processors would be idle while these tasks were performed.

As they stand the basic algorithms suffer from some drawbacks as

regards multiprocessing. However one variation on the global subdivision strategy would appear to solve most of the problems and make the use of a multiple processor system highly beneficial. Although the global subdivision strategy involves processing the subregions in the linked list one at a time there is no reason why work could not be started on other subregions prior to the completion of the calculations for the first subregion. Then processors would only be idle when the number of elements in the list of subregions was less than the number of processors. At first sight this would appear to imply that it is necessary to wait for all the processes to complete before an estimate can be formed since more than one subregion is removed from the list at a time and the order in which the results are returned cannot be guaranteed. However these problems can be avoided by not removing the contributions to the final result from a particular subregion until the sum of the estimates over the subregions of that subregion have been formed. Then the total estimate will always be an approximation to the result but the algorithm cannot be guaranteed to be global since the order in which the estimates over the subregions in the list are returned will be processor dependant. For example if the first three elements of the list are passed to three processors in order to calculate the new estimates over those subregions then it is possible that the computations for the third element may be completed first, in which case the subregion with the largest error estimate has not been replaced first. This should not affect the algorithm considerably but is worth noting. Adopting this approach the basis of an algorithm is :

```
Form the initial estimate to the result, the error estimate and the
list of subregions;
WHILE {convergence has not been achieved}
     the error estimate > tolerance
DO
     IF a processor is available
     THEN
          start the processor working on the subregion at the head
          of the list ;
          {remove this subregion from the list}
          advance the head to the next item in the list
     FI ;
     IF a process is complete
     THEN
          subtract the old estimate over the subregion from the
          total estimate;
          subtract the old error estimate from the total error
          estimate ;
          add the new estimate over the subregions of this subregion
          to the total estimate;
          add the associated error estimate to the total error
          estimate;
          instigate the process of adding the list of new subregions
          to the subregion list
     FI
OD
{Output the results}
```

The algorithm would be executed on the master processor and

semaphores would have to be used to protect certain areas. For example the list of subregions is accessed by the master and the slaves. The master takes information from the head of the list, passes this information to a slave and then advances the head of the list. Obviously it is important that no other processor alters the list while these operations are being performed if corruption of the list is to be avoided. The other process which might attempt to alter the list at the same time is the one which adds the new subregions of a subregion to the list. A problem could occur if a subregion was added at the head of the list just before the master processor advanced the head of the list; the wrong element could be removed from the list. This could be avoided by making the master processor wait until the transfer of all the list of subregions was complete before moving on. However this would be inefficient. A better approach is to use a semaphore to make the two processes mutually exclusive. That is when an element is being added to the list the operations associated with advancing the head cannot take place and alternatively when the operations associated with advancing the head are taking place then an element cannot be added to the list. The master processor could be given priority so that free processors were not kept waiting while new subregions were added to the list. The only time that a processor would have to wait would be when the list of subregions was empty; this situation could arise while a processor was actually adding a list of new subregions to the list of subregions.

All communications take place via the bus. Therefore it is necessary to have some form of arbitration to decide which processor actually gets the bus when more than one request is made for it at any one

time. The proportion of the computation allocated to the master is relatively small. This enables the master to have time to perform this arbitrat ion without degrading the performance. Hence the master is in full control.

8.5 Conclusions

A straightforward extension of the algorithm based upon the global subdivision strategy has been presented which it is suggested would be suitable for use on a machine with a multiple instruction multiple data type architecture. It is felt that considerable improvements in the performance of the algorithm could be expected with a reasonable number of processors if the overheads can be kept to a minimum since a significant number of tasks exhibit a natural concurrency. Obviously it would be best if an architecture was designed to suit the problem. This could be feasible in the near future when it is expected that the availability of numerous processors and facilities to reconfigure the interconnection topology of those processors under software control will be just an extension of the software engineer's tools. However, at the moment some of the languages designed to run on standard multiple processor systems, such as Ada and concurrent Pascal [21], offer many of the facilities required to construct an algorithm of the type described above. Some discussions about using these languages are given in Dawson [10] and Bowen and Buhr [3]. In conclusion it would be possible to construct a program based on the algorithm described which would run on a multiple processor system.

Chapter 9 Conclusions

In the previous chapters the development of software for the approproximate evaluation of multiple integrals has been considered. The work has been based on the construction of reliable software to apply the existing theory in an efficient manner. It is hoped that this work will be a starting point for the development of further software for multidimensional quadrature and that any further advance in the theory of numerical integration will lead to the construction of formulae that are suitable for use in multidimensional quadrature programs. Multidimensional quadrature is such a laborious exercise in terms of computation that it is unlikely that any formulae which are not suitable for use on a computer will ever be used.

One of the first objectives of the work was to construct two adaptive algorithms, one for regions of integration which are hypercubes and the second for regions of integration which are simplexes. These have been developed and the resulting programs perform well but are slightly inefficient in that they tend to produce more accurate results than required at a correspondingly higher "cost". The efficiency of the programs can be improved by adopting a global subdivision strategy as opposed to a local subdivision strategy without affecting the reliability. This was demonstrated by the program which was written to perform a global subdivision strategy over a hypercube type region of integration. Exactly the same strategy could be adopted for the simplex. The program for the hypercube performed well and extended the range of tolerances over which results could be produced before the limits on

time and store were reached by producing results closer to the required tolerance. Further research could improve the methods of storing and accessing the lists of subregions used in the global subdivision strategy program and it is recommended that any further software written to perform adaptive multidimensional quadrature should adopt this subdivision strategy.

Another objective was to consider the possibilities of storing integrand evaluations as a means of reducing the overall time taken to compute an approximation. The techniques for storing integrand evaluations were developed with reference to the hypercube as it was felt that the same techniques could be applied to the simplex if they were successful. Two distinct approaches were taken to storing the integrand evaluations; firstly a linked list or series of linked lists was used and secondly scatter storage techniques were adopted. Both methods reduced the actual number of integrand evaluations used considerably once the methods reached a level of subdivision where a large number of integrand evaluations were required. However both methods proved to be very "expensive", and are not feasible unless the integrand is very "expensive" to evaluate and it is envisaged that a large number of integrand evaluations will be computed to achieve the accuracy required. The major problem with storing the integrand evaluations is that certain integration formulae which lend themselves to the subdivision and storage methods have to be used. Unfortunately these formulae tend to be less accurate and use more integrand evaluations than other available formulae. If formulae were developed with the maximum accuracy using a minimum number of nodes and the nodes were distributed such that they could be reused after the subdivision then the methods would be very

useful.

The construction of product formulae has been reviewed. One program was written to construct and apply an n dimensional product rule. This program is a non adaptive method based upon a straightforward iterative scheme. At each iteration it constructs an n dimensional product rule from the one dimensional rules of Patterson and uses this rule to produce the next approximation. The program demonstrates one method of storing the integrand evaluations so as to take advantage of the nature of the Patterson formulae which form a common point family. The program either produces very accurate results or fails because of the vast number of integrand evaluations required for the next iteration. As an iterative scheme the program is not viable with the present limitations on computing power and resources. However the lower order formulae could be used successfully in conjunction with an adaptive scheme to produce initial accurate results over uniformly behaved regions.

In this thesis working programs have been written to produce approximations over both the hypercube and the simplex. However, the majority of problems that people actually wish to solve do not fit into either of these categories. Therefore a program was written to demonstrate how the methods used over these two regions can be extended to the solution of problems over any region of integration which can be defined as a union of hypercubes and simplexes. This approach can be modified to incorporate alternative formulae and to take advantage of any user knowledge of the integrand. The area of subdivision of regions of integration into unions of simplexes and hypercubes is one which requires further research.

With the changing nature of the architecture of computer systems no work would be complete without considering the future. The possible advantages of using a multiple processor system have been considered and the author suggests that a considerable improvement in the performance of quadrature algorithms could be expected if an architecture of this type was adopted.

The testing of quadrature is very complex and some of the methods have been examined. The author considers that it would be beneficial to carry out research into the testing and comparison of quadrature programs and the types of problem that need to be solved before any further work is carried out in the area of numerical software for multidimensional quadrature.

In this work the improvement of approximations to the result has been achieved by a method of subdivision and the application of formulae to the subregions. As the method progresses a sequence of converging estimates to the result are produced. One possible method of improving the results would be to use an extrapolation technique. Anders [1] and Strom [56] have published papers on particular extrapolation methods for multiple integrals. Lyness and McHugh [40] have developed an extension of Richardson extrapolation for the hypercube and this progressive procedure can be applied to other formulae. Genz [16] applied a variation of this method to improve the results of his multidimensional quadrature program. These techniques could perhaps be applied beneficially to the algorithms described in this thesis with an aim to improving the results without using any more integrand evaluations.

In conclusion this research has resulted in the development of a variety of algorithms for the appropriate evaluation of multiple integrals. On the whole the programs perform well and form the basis for the development of efficient, reliable software for multidimensional quadrature.

APPENDICES

Appendix 1

1.1 The set of test problems

All the problems are written in the form of procedures which evaluate the integrand, defining the problem, at any given node. Each procedure has the same name, f, but is contained in it's own segment. All the segments and the procedures have the following form:

```
Segf12d
BEGIN
    {this is the two dimensional version of the first test problem,
    this is indicated by the 12d in the segment name}
    PROC f = (REF[]REAL x)REAL :
        (sqrt(x[1]+x[2])) ;
    SKIP
END
KEEP f
FINISH
```

1.2 The set of test problems used for the hypercube programs

Segf12d

$$\int_0^1\int_0^1 \mathrm{sqrt}(x_1+x_2)\,dx_1dx_2 \approx 0.975161133$$

Segf13d

$$\int_0^1\int_0^1\int_0^1 \mathrm{sqrt}(x_1+x_2+x_3)\,dx_1dx_2dx_3 \approx 1.205656861$$

Segf14d

$$\int_0^1\int_0^1\int_0^1\int_0^1 \mathrm{sqrt}(x_1+x_2+x_3+x_4)\,dx_1dx_2dx_3dx_4 \approx 1.398180578$$

Segf22d

$$\int_0^1\int_0^1 \mathrm{sqrt}(x_1 * x_2)\,dx_1dx_2 \approx 0.444444444$$

Segf23d

$$\int_0^1\int_0^1\int_0^1 \mathrm{sqrt}(x_1 * x_2 * x_3)\,dx_1dx_2dx_3 \approx 0.296296296$$

Segf32d

$$\int_0^1\int_0^1 1/(4+x_1+x_2)\,dx_1dx_2 \approx 0.201355135$$

Segf33d

$$\int_0^1\int_0^1\int_0^1 1/(4+x_1+x_2+x_3)\,dx_1dx_2dx_3 \approx 0.183354140$$

Segf42d

$$\int_{-1}^{+1}\int_{-1}^{+1} \exp(\sin x_1 \sin x_2)\,dx_1dx_2 \approx 4.151291030$$

Segf43d

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \exp(\sin x_1 \sin x_2 \sin x_3)\,dx_1dx_2dx_3 \approx 8.081734973$$

1.3 The set of test problems used for the simplex programs

Segf12d

$$\int_0^1 \int_0^{1-x_1} sqrt(x_1 + x_2)\, dx_1 dx_2 \approx 0.400000000$$

Segf13d

$$\int_0^1 \int_0^{1-x_1} \int_0^{1-x_1-x_2} sqrt(x_1 + x_2 + x_3)\, dx_1 dx_2 dx_3 \approx 0.142857143$$

Segf22d

$$\int_0^1 \int_0^{1-x_1} sqrt(x_1 * x_2)\, dx_1 dx_2 \approx 0.130951982$$

Segf32d

$$\int_0^1 \int_0^{1-x_1} 1/(4+x_1+x_2)\, dx_1 dx_2 \approx 0.1\ 07425796$$

Segf33d

$$\int_0^1 \int_0^{1-x_1} \int_0^{1-x_1-x_2} 1/(4+x_1+x_2+x_3)\, dx_1 dx_2 dx_3 \approx 0.035148211$$

Segf42d

$$\int_0^1 \int_0^{1-x_1} exp(\sin x_1 \sin x_2)\, dx_1 dx_2 \approx 0.541481025$$

Appendix 2

2.1 The nodes and weights for Patterson's formulae

The following are the nodes and weights for Patterson's one dimensional formulae:

```
 0 SEGNODES
 1 'BEGIN'
 2     'C' THE FOLLOWING ARE THE NODES AND WEIGHTS FOR PATTERSONS
 3         FORMULAE 'C'
 4
 5     []'REAL' NODES =  ( 0.0
 6   , 0.77459666924, 0.96049126871, 0.43424374935, 0.99383196321
 7   , 0.88845923287, 0.62110294674, 0.22338668643, 0.99909812497
 8   , 0.98153114955, 0.92965485743, 0.83672593817, 0.70249620649
 9   , 0.53131974364, 0.33113539326, 0.11248894313, 0.99987288812
10   , 0.99720625937, 0.98868475755, 0.97218287475, 0.94634285837
11   , 0.91037115696, 0.86390793819, 0.80694053195, 0.73975604435
12   , 0.66290966002, 0.57719571005, 0.48361802695, 0.38335932420
13   , 0.27774982202, 0.16823525155, 0.05634431305, 0.99998243035
14   , 0.99959879967, 0.99831663532, 0.99572410470, 0.99149572118
15   , 0.98537149960, 0.97714151464, 0.96663785156, 0.95373000643
16   , 0.93832039778, 0.92034002547, 0.89974489978, 0.87651341448
17   , 0.85064449477, 0.82215625436, 0.79108493380, 0.75748396638
18   , 0.72142308537, 0.68298743109, 0.64227664251, 0.59940393024
19   , 0.55449513263, 0.50768775753, 0.45913001199, 0.40897982123
20   , 0.35740383783, 0.30457644156, 0.25067873030, 0.19589750271
21   , 0.14042423315, 0.08445404008, 0.02818464895, 0.99999759636
22   , 0.99994399621, 0.99976049092, 0.99938033803, 0.99874561447
23   , 0.99780535450, 0.99651414591, 0.99483150280, 0.99272134428
24   , 0.99015137040, 0.98709252795, 0.98351865758, 0.97940628167
25   , 0.97473445975, 0.96948465950, 0.96364062157, 0.95718821611
26   , 0.95011529752, 0.94241156519, 0.93406843616, 0.92507893291
27   , 0.91543758716, 0.90514035881, 0.89418456834, 0.88256884025
28   , 0.87029305555, 0.85735831089, 0.84376688267, 0.82952219464
29   , 0.81462878766, 0.79909229096, 0.78291939412, 0.76611781930
30   , 0.74869629362, 0.73066452124, 0.71203315536, 0.69281376978
31   , 0.67301883023, 0.65266166541, 0.63175643771, 0.61031811372
32   , 0.58836243445, 0.56590588542, 0.54296566650, 0.51955966154
33   , 0.49570640792, 0.47142506587, 0.44673538766, 0.42165768663
34   , 0.39621280606, 0.37042208795, 0.34430734160, 0.31789081207
35   , 0.29119514852, 0.26424337241, 0.23705884559, 0.20966523824
36   , 0.18208649676, 0.15434681148, 0.12647058437, 0.09848239660
37   , 0.07040697604, 0.04226916477, 0.01409388641 ) ;
38
39
40
41
```

```
50      []'REAL' WEIGHTS = (
51        0.8888888888, 0.5555555555, 0.4509165386 6, 0.2684880898 7
52      , 0.1046562260 3, 0.4013974147 8, 0.2255104998 0, 0.1344152552 4
53      , 0.0516032830 0, 0.20062852938, 0.0170017196 3, 0.0929271953 2
54      , 0.1715119091 4, 0.2191568584 0, 0.1127552567 2, 0.0672077543 0
55      , 0.0258075981 0, 0.1003142786 1, 0.0084345657 4, 0.0464628932 6
56      , 0.0857559200 5, 0.1095784210 6, 0.0025447807 9, 0.0164460498 5
57      , 0.0359571033 1, 0.0569795094 9, 0.0768796205 0, 0.0936271099 8
58      , 0.1056698935 8, 0.1119568730 2, 0.0563776283 6, 0.0336038771 5
59      , 0.0129038001 0, 0.0501571393 1, 0.0042176304 4, 0.0232314466 4
60      , 0.0428779600 3, 0.0547892105 3, 0.0012651565 6, 0.0082230079 6
61      , 0.0179785515 7, 0.0284897547 5, 0.0384398102 5, 0.0468135549 9
62      , 0.0528349467 9, 0.0559784365 1, 0.0003632214 8, 0.0025790497 9
63      , 0.0061155068 2, 0.0104982469 1, 0.0154067504 7, 0.0205942339 2
64      , 0.0258696793 3, 0.0310735511 1, 0.0360644327 8, 0.0407155101 2
65      , 0.0449145316 5, 0.0485643304 1, 0.0515832539 5, 0.0539054993 4
66      , 0.0554814043 6, 0.0562776998 3, 0.0281888141 8, 0.0168019385 7
67      , 0.0064319000 5, 0.0250785696 5, 0.0021088152 5, 0.0116157233 2
68      , 0.0214389800 1, 0.0273946052 6, 0.0006326073 2, 0.0041115039 8
69      , 0.0089892757 8, 0.0142448773 7, 0.0192199051 2, 0.0234067775 0
70      , 0.0264174734 0, 0.0279892182 6, 0.0001807395 6, 0.0012895240 8
71      , 0.0030577534 1, 0.0052491234 5, 0.0077033752 3, 0.0102971169 6
72      , 0.0129348396 6, 0.0155367755 6, 0.0180322163 9, 0.0203577550 6
73      , 0.0224572658 3, 0.0242821652 0, 0.0257916269 8, 0.0269527496 7
74      , 0.0277407021 8, 0.0281388499 2, 0.0000505361 0, 0.0003777466 5
75      , 0.0009383698 5, 0.0016811428 7, 0.0025687649 4, 0.0035728927 8
76      , 0.0046710503 7, 0.0058434498 8, 0.0070724900 0, 0.0083428387 5
77      , 0.0096411777 3, 0.0109557333 9, 0.0122758305 6, 0.0135915710 1
78      , 0.0148936416 6, 0.0161732187 3, 0.0174219301 6, 0.0186318482 6
79      , 0.0197954950 5, 0.0209058514 5, 0.0219563663 1, 0.0229409642 3
80      , 0.0238540521 1, 0.0246905247 4, 0.0254457699 7, 0.0261156733 8
81      , 0.0266966229 3, 0.0271855132 3, 0.0275797495 7, 0.0278772514 8
82      , 0.0280764557 9, 0.0281763190 3, 0.0140944070 9, 0.0084009692 9
83      , 0.0032259500 3, 0.0125392848 3, 0.0010544076 2, 0.0058078616 6
84      , 0.0107194900 1, 0.0136973026 3, 0.0003163036 6, 0.0020557519 9
85      , 0.0044946378 9, 0.0071224386 9, 0.0096099525 6, 0.0117033887 5
86      , 0.0132087367 0, 0.0139946091 3, 0.0000903727 3, 0.0006447620 4
87      , 0.0015288767 1, 0.0026245617 3, 0.0038516876 2, 0.0051485584 8
88      , 0.0064674198 3, 0.0077683877 8, 0.0090161082 0, 0.0101788775 3
89      , 0.0112286329 1, 0.0121410826 0, 0.0128958134 9, 0.0134763748 3
90      , 0.0138703510 9, 0.0140694249 6, 0.0000251578 7, 0.0001888732 6
91      , 0.0004691849 2, 0.0008405714 3, 0.0012843824 7, 0.0017864463 9
92      , 0.0023355251 9, 0.0029217249 4, 0.0035362450 0, 0.0041714193 8
93      , 0.0048205888 6, 0.0054778666 9, 0.0061379152 8, 0.0067957855 0
94      , 0.0074468203 3, 0.0080866093 6, 0.0087109650 8, 0.0093159241 3
95      , 0.0098977475 2, 0.0104529257 2, 0.0109781831 5, 0.0114704821 1
96      , 0.0119270260 5, 0.0123452623 7, 0.0127228849 8, 0.0130578366 9
97      , 0.0133483114 6, 0.0135927566 1, 0.0137898747 8, 0.0139386257 4
98      , 0.0140382279 0, 0.0140881595 2, 0.0000069379 4, 0.0000532752 9
99      , 0.0001357549 1, 0.0002492124 0, 0.0003897452 8, 0.0005542953 1
100     , 0.0007402828 0, 0.0009453615 2, 0.0011674841 2, 0.0014049080 0
101     , 0.0016561127 3, 0.0019197129 7, 0.0021944069 3, 0.0024789582 3
102     , 0.0027721957 6, 0.0030730184 3, 0.0033803979 9, 0.0036933779 2
103     , 0.0040110687 2, 0.0043326409 7, 0.0046573173 0, 0.0049843645 6
104     , 0.0053130866 1, 0.0056428181 0, 0.0059729195 7, 0.0063027734 5
105     , 0.0066317812 4, 0.0069593614 1, 0.0072849479 8, 0.0076079896 7
106     , 0.0079279493 3, 0.0082443037 6, 0.0085565435 6, 0.0088641732 1
107     , 0.0091667111 6, 0.0094636899 9, 0.0097546565 4, 0.0100391720 4
108     , 0.0103168123 3, 0.0105871679 0, 0.0108498440 9, 0.0111044611 3
109     , 0.0113506543 2, 0.0115880740 3, 0.0118163858 9, 0.0120352707 9
110     , 0.0122444249 8, 0.0124435601 9, 0.0126324036 4, 0.0128106981 6
111     , 0.0129782022 4, 0.0131346900 9, 0.0132799517 4, 0.0134137930 9
112     , 0.0135360359 3, 0.0136465181 0, 0.0137450934 4, 0.0138316319 1
113     , 0.0139060196 0, 0.0139681588 1, 0.0140179680 4, 0.0140553820 7
114     , 0.0140803519 6, 0.0140928450 7 ) ;
```

2.2 The complete program based on Patterson's rules

```
Product type method based on Patterson's rules
WITH segnodes, segmilltime, segfl-2d FROM pjk-alb-al
BEGIN
     [] INT starting positions = (0,2,6,14,30) ;
     INT maxpoints , nofe := 0 , n ;
     REAL eps ;
     read((maxpoints,eps,n)) ;
     []INT m = (3,7,15) ;
     INT next := 1 ;
     [1:n]INT array ;
     BOOL exist := FALSE , notgt8 := TRUE , nottoomanyfe := TRUE ;
     REAL result , result1 ;
     MODE NODE = STRUCT(REAL feval,REF NODE ptr) ;
     PROC add to list = (REAL feval, REF REF NODE pointer) VOID :
     BEGIN
          REF NODE newnode = NODE ;
          {create a new node on the heap}
          feval OF newnode := feval ;
          ptr OF newnode := pointer ;
          {pointer dereferenced twice}
          pointer := newnode
          {pointer dereferenced once}
     END ;

     REAL  alt, div, scf, cen ;
     INT number of tests ;
     read((n,newline));
```

```
read((number of tests,newline)) ;
[1:number of tests]REAL epsa ;
read((epsa,newline,div,newline,alt,newline,
      scf,newline,cen,newline)) ;
[1:n]REAL centre ;
FOR i TO n DO centre[i] := cen ;
[]INT m = (3,7,15,31,63,127,255) ;
INT next := 1 ;
[1:n]INT array ; [1:n]REAL point ;
BOOL exist, notgt8, nottoomanyfe ;
REAL result, result1, corresponding weight, function eval ;

PROC generate = (INT m,n,previousm,starting position,
                 REF[]INT array,
                 BOOL exist, REF REAL present estimate)VOID:
BEGIN
{This is a procedure to generate the nodes of an n dimensional
product type rule from an m point one dimensional rule and to
generate an estimate to an integral using this rule}
FOR i TO m DO
BEGIN
    array[n] := i ;
    IF exist
    THEN IF i > previousm
         THEN exist := FALSE
         FI
    FI;
    IF n> 1
    THEN generate (m,n-1,previousm,starting position,array,
```

```
                exist,present estimate)
    ELSE IF exist
        THEN
         {the integrand has been evaluated at this node,the
         required integrand evaluation is the next value in
         the list
         generate the corresponding weight}
             corresponding weight := 1;
             FOR i TO n
             DO corresponding weight TIMES
                 weights[starting position + array[i]'/'2+1]
 ;
             {add the weight * the next value to the estimate}
             present estimate PLUS (corresponding *
                 feval OF pointer) ;
             {move the pointer to the next item in the list}
             pointer := ptr OF pointer
        ELSE {the node has not already been used}
             {interpret the array and generate the weight}
             corresponding weight := 1;
             FOR i TO n DO
             BEGIN
                INT nn ; BOOL posnode := ODD(nn:=array[i]) ;
                nn := nn'/'2 + 1 ;
                point[i] :=
                 (IF posnode
                  THEN nodes[nn]
                  ELSE - nodes[nn]
                  FI)/div + centre ;
```

```
                corresponding weight TIMES
                 weights[starting position + nn]
              END ;
              integrand eval := f(point) ;
              present estimate PLUS
                 (corresponding weight * integrand eval) ;
              nofe PLUS 1 ;
              {increase the number of integrand evaluations}
              add to list(integrand eval, pointer)
           FI
        FI
    END
END of the procedure generate.

NODE st := (f(centre),NIL) ;
REF NODE start := st ;
REF REF NODE pointer := start ;

{output the details of this test run}
print((....));
LONG INT before, after ;
FOR i TO number of tests DO
BEGIN
    start := st ; pointer := start ;
    eps := epsa[i] ; nofe := 1 ;
    exist := TRUE ; notgt8 := TRUE ;
    next := 1
    result := 0.0 ;
    before := milltime ;
```

```
        generate(m[next],n,1,starting
        position[next],array,exist,result) ;
        esult DIV scf ;
        pointer := start ;
        WHILE (exist := TRUE ;
          pointer := ptr OF start ;
          generate (m[nextPLUS1],n,m[next-1],starting
          positions[next],array,exist,result1) ;
          result1 DIV scf ;
          ABS(result1 - result) > eps)
         AND notgt8 := next < 8
         AND nottoomanyfe := nofe < max
    DO result := result1 ;

    IF NOT notgt8
    THEN print((newline,"all nodes used ",newline))
    FI ;

    IF NOT nottoomanyfe
    THEN print((newline,"Too many integrand evaluations required"))
    FI ;

    print((newline,"The result is : ",result1,newline))
END
FINISH
```

Appendix 3

3.1 The basic adaptive method for the hypercube

The program now follows in order of the segments:

3.1.1 The first segment

```
SEGCONST
BEGIN
    {This  segment  declares the common constants and variables used
    by other segments}
    INT n ; {n is the number of dimensions of the problem}
    read((n,newline)); {input the value of n}
    INT npl = n + 1 , num = 2^n ;
    {num  is the number of subregions into which each region will be
    divided}
    INT numeval ; {numeval is the number of integrand evaluations}
    REAL constl = sqrt(2/3) , const2 = sqrt(3) ;
    BOOL odd = 2*(ENTIER(n/2))£n;
    SKIP
END
KEEP n,npl,num,constl,const2,numeval,odd
FINISH
```

3.1.2 The second segment

```
SEGMILLTIME
```

```
BEGIN
    {The  following  procedure  is used to give the time of call, as
    given  by  a  real  time clock, in the form of a LONG INT. It is
    used  to make comparative timings by calling it before and after
    an event. }
    PROC milltime = LONG INT :
    BEGIN
        LONG INT CODE 165,6,10 EDOC
    END ;
    SKIP
END
KEEP milltime
FINISH
```

3.1.3 The third segment

This segment contains the first of the two basic rules used in the algorithm. Initially the two basic rules used were Stroud's n+1 point rule of degree 2 and Stroud's 2n point rule of degree 3. The first is applied using the procedure evalA while the second is applied using the procedure evalB (this is defined in the next segment). Since only one of the rules is used to give an estimate which forms part of the final approximation, the other being an intermediate tool used to evaluate an error estimate over a subregion, it is sensible to choose the more accurate of the two rules for this role. Hence in this case, the second rule is the more accurate and evalB is used. For ease, the following convention is adopted when testing any pair of basic rules; with any pair of rules the more accurate of the two, if there is one, is used in the

procedure evalB and the other in the procedure evlaA.

```
SEGEVALA
WITH segconst FROM albumname
BEGIN
    {This  segment  contains a procedure to obtain an estimate to an
    integral over a subregion}
    REAL const3 = num/npl ;
    {Evaluate and store the nodes for Stroud's n+1 point rule}
    [0:n,1:n]REAL nodesa ;
    {array to hold the nodes for this rule}
    FOR k FROM 0 TO n DO
    BEGIN
        REF[]REAL t = nodesa[k,1:n]
        {set  up a reference to one of the nodes, each row contains
        one node}
        FOR r TO n'/'2 DO
        BEGIN
            INT r2 = 2*r ; REAL theta = (r2*k*pi)/npl ;
            t[r2-1] := const1 * cos(theta) ;
            t[r2]   := const1 * sin(theta)
        END ;
        IF odd THEN t[n] := ((-1)^k)/const2 FI
    END ;
    {All the nodes have now been evaluated and stored.
    The  following  is  a  procedure  to evaluate an estimate to the
    integral of a region using Stroud's n+1 point rule}
    PROC evala = (REAL div,scf, REF[]REAL centre,
                PROC (REF[]REAL)REAL f ) REAL :
```

```
BEGIN
      REAL estimate := 0.0 ;
      {estimate is the approximation to the integral}
      {Evaluate the function at each node using the procedure f
      and add the value to the total estimate}
      FOR j FROM 0 TO n DO
      BEGIN
            [1:n]REAL tempnode ;
            {used to store each node in turn}
            REF[]REAL t2 = nodesa[j,1:n] ; {next node}
            {Now transform the node t2 to the given subregion
            defined by centre, scf, and div. Store this newnode
            in tempnode}
            FOR k TO n DO tempnode[k] := centre[k] + t2[k]/div ;
            {Now add the integrand evaluation at this temporary
            node to the estimate}
            estimate PLUS f(tempnode)
      END ;
      {Increment the number of integrand evaluations}
      numeval PLUS npl ;
      {Multiply the estimate by the scaling factor and the
      weight const3 to give the applicable estimate}
      estimate TIMES (const3/scf)
      {Deliver the estimate as the result of the procedure
      evala}
   END ; {end of the procedure evala}
END {end of the segment SEGEVALA}
KEEP evala
{Make the procedure available to any program which links this
```

segment}

FINISH

3.1.4 The fourth segment

```
SEGEVALB
WITH segconst FROM albumname
BEGIN
    {This  segment  contains a procedure to obtain an estimate to an
    integral over a subregion}
    INT n2 = 2*n ;REAL const4 = num/n2 ;
    [1:2n,1:n]REAL nodesb ;
    {array to hold the nodes for this rule}
    {Evaluate and store the nodes for Stroud's 2n point rule}
    FOR k TO n DO
    BEGIN
        REF[]REAL t1 = nodesb[k+n,1:n] ;
        REF[]REAL t2 = nodesb[k  ,1:n] ;
        {set  up references to two of the nodes, each row holds one
        node}
        FOR r TO n'/'2 DO
        BEGIN
            INT r2 = 2*r ;
            {Set  up  the  constants  for  the  derivation  of  the
            nodes}
            INT r2m1 = r2 - 1 ;
            REAL theta = (r2m1*k*pi)/n ;
            {Store the next four nodes generated}
            t1[r2m1] := const1 * cos(theta) ;
            t2[r2m1] := - t1[r2m1] ;
            t1[r2]   := const1 * sin(theta) ;
            t2[r2]   := -t1[r2]
```

```
    END ;
    IF odd
    THEN t1[n] := (-1)^k/const2 ;
         t2[n] := - t1[n]
    FI
END ;
{All the nodes have now been evaluated and stored}
{The  following  is used to evaluate an estimate to the integral
over a region using Stroud's 2n point rule}
PROC evalb = (REAL div,scf, REF[]REAL centre,
              PROC (REF[]REAL)REAL f) REAL :
BEGIN
    REAL estimate := 0.0 ;
    {estimate is the approximation to the integral}
    {Evaluate  the  function at each node using the procedure f
    and add the value to the estimate}
    FOR j TO 2n DO
    BEGIN
         [1:n]REAL tempnode ;
         {used to store each node in turn}
         REF[]REAL t2 = nodesb[j,1:n] ; {next node}
         {Now  transform  the  node  t2  to the given subregion
         defined  by centre,scf, and div. Store this newnode in
         tempnode}
         FOR k TO n DO
          tempnode[k] := centre[k] + t2[k]/div ;
         {Now  add  the  integrand evaluation at this temporary
         node to the estimate}
         estimate PLUS f(tempnode)
```

```
        END ;
        {Increment the number of integrand evaluations}
        numeval PLUS n2 ;
        {Multiply the estimate by the scaling factor and the
        weight const4 to give the applicable estimate for this
        subregion}
        estimate TIMES (const4/scf)
        {Deliver estimate as the result of the procedure evalb}
    END ; {end of the procedure evalb}
    SKIP
END {end of the segment SEGEVALB}
KEEP evalb
{Make the procedure evalb available to any other program}
FINISH
```

3.1.5 The main body of the program

```
Multidimensional Adaptive Quadrature Program 1
WITH segconst,segmilltime,segevala,segevalb,segf FROM albumname
BEGIN
    {This is an adaptive multidimensional quadrature program. It
    delivers an approximation to the integral of the function
    defined in segf over the hypercube defined by the input
    data.The data defines the number of approximations required,
    the tolerance for each and the region of integration. The
    algorithm is based upon a strategy of subdivision of the
    initial hypercube into increasing numbers of subregions and the
    application of the basic rules, defined in segevala and
    segevalb, to form estimates over the subregions. The algorithm
    is adaptive; subregions are dismissed from further
    consideration once a given accuracy has been achieved in them}

    REAL altl,tot,err,divl,scfl,centre,consub ;
    INT number of tests, level of subdivisioin ;

    {Input the number of tests required}
    read((number of tests,newline));
    {Set up an array for the tolerances}
    [1:number of tests]REAL epsa ;
    {Input the tolerances and the data defining the region of
    integration}
    read((epsa,newline,divl,newline,altl,newline,scfl,newline,centre
    )) ;
```

{The program uses a linked list to keep track of the subregions, each node in the list is defined as :}

```
MODE NODE = STRUCT(REF NODE ptr, REF[,]REAL subcentres) ;
```

{The following procedure considers the element of a list of nodes pointed to by head. For each of the subcentres of this node it computes two estimates to the integral over the subregion defined by that subcentre. If the difference is less than the tolerance eps it adds the estimate to the final estimate, increases the number of converged subregions and adds the difference to the total error estimate. Otherwise a new node is added to the list pointed to by newhead. This node contains the subcentres of the subregions of this subregion. Alt, scf and div are parameters defining the subregions.}

```
PROC compute estimate = (REF REF NODE head, newhead,
                        REAL eps, alt, scf, div) REAL :
BEGIN
     REAL result := 0.0 ;
     {result is the estimate delivered by the procedure}
     {Consider each subcentre in turn}
     FOR i TO UPB subcentres OF head DO
     BEGIN
          REF[]REAL centre = (subcentresOFhead)[i,1:n] ;
          {Let centre point to the next subcentre and form the
          two estimates to the integral over the subregion}
          REAL estimatea := evala(div,scf,centre,f),
               estimateb := evalb(div,scf,centre,f),     diff ;
          {Set diff to be the difference between the two
```

```
estimates}
diff := ABS(estimatea - estimateb) ;
result PLUS estimateb ;
IF diff < eps  {test for convergence}
THEN
     tot PLUS estimateb ;
     {add estimateb to the final estimate}
     consub PLUS 1 ;
     {Increment the number of converged subregions}
     err PLUS diff
     {Add the difference to the total error estimate}
ELSE
     {Convergence not achieved in this subregion}
     REF NODE temptr := newhead ;
     newhead := NODE ; {declare a new node}
     ptrOFhead := temptr ;
     {link the new node to the list}
     REF[,]REAL centres = [1:num,1:n]REAL ;
     {Set up the subcentres of this subregion}
     FOR i TO num DO
     BEGIN
          {Set up a pointer to the next subcentre}
          REF[]REAL sub = centres[i, ];
          BITS b := BIN i ;
          {This section of the procedure uses the
          bits pattern of the digit i to determine
          the subcentres of the subregion

          FOR j TO n DO
```

```
                    IF (25-j)ELEM b
                    THEN sub[j] := centre[j] + alt
                    ELSE sub[j] := centre[j] - alt
                    FI
            END ;
            subcentresOFnewhead := centres
        FI
    END ;
    head := ptrOFhead ;
    {Move on to the next node in the list}
    result
END ; {end of the procedure compute estimate}

{Set up the variables ...}
[1:1,1:n]REAL c ; {c is used to represent the centre}
FOR j TO n DO c[1,j] := centre ;
REF NODE nil = NIL ;
LONG INT before, after ;

{Perform the calculations for each test}
FOR i TO number of tests DO
BEGIN
    {Set up the variables for this test}
    REAL sum := 0.0, latest := 0.0, eps := epsa[i], neweps,
        alt  :=  altl,  subregions  :=  1, div := divl, scf :=
    scfl ;
    consub := 0; tot := 0; err := 0;
    level of subdivision := 1; neweps := eps;
    REF NODE head := NODE := (nil,c) ;
```

```
REF NODE newhead := nil ;
BOOL notconverged := TRUE ;
{Start timing this test using the procedure milltime}
before := milltime ;
WHILE notconverged DO
BEGIN
    latest  := sum ;{set latest to the most recent
    estimate}
    sum := tot ;
    {sum will be the present, it is set initially to the
    total estimate from converged subregions}
    {Now process the list of non converged subregions,
    adding  the estimates over each to the present
    estimate sum}
    WHILE head ISNT nil    {while list not empty}
    DO             sum              PLUS            compute
    estimate(head,newhead,neweps,alt,scf,div);
    {All estimates formed and added to sum. A new list of
    non converged subregions has been formed and is
    pointed to by newhead. Test for convergence.}
    IF (newhead IS nil)
       {list of non converged subregions is empty}
       OR (ABS(latest-sum)<eps)
       {the  difference between the two most recent
    estimates
        is less than the required tolerance}
    THEN notconverged := FALSE
         {solution found}
    ELSE
```

```
        {Set up the variables to consider the next list
        of non converged subregions, ie consider the
        next level of subdivision}
        subregions TIMES num ;
        {subregions is equal to the total number of
        subregions at this level of subdivision}
        level of subdivision PLUS 1 ;
        consub TIMES num ;
        {Evaluate the number of converged subregions at
        this level of subdivision, ie consub}
        {Evaluate the tolerance to be applied to each
        subregion. This consists of dividing the
        tolerance minus the error estimate from the
        converged subregions between each of the non
        converged subregions, ie the total number of
        subregions minus the converged subregions at
        this level}
        neweps := (eps-err)/(subregions-consub) ;
        {Alter the scaling factors}
        div TIMES 2 ; alt DIV 2 ; scf TIMES num;
        {Replace the old list of non converged
        subregions by the new one and set the new one to
        empty}
        head := newhead ;
        newhead := nil
    FI
END ;
{Estimate formed for this test, finish timing}
after := milltime ;
```

```
{Output the information required from this test}
print(("The final estimate to the integral is
",sum,newline,
      "The time taken to obtain the result was
",after-before,
      "The number of integrand evaluations was ",numeval,
      "The tolerance for this run was ",eps,
      "The level of subdivision was ",level of
subdivision)) ;
numeval  := 0 {reset the number of integrand evaluations to
0}
    END {end of the calculations loop}
END { end of the program:
    Multidimensional Adaptive Quadrature Program 1 }
FINISH
```

3.2 The basic adaptive program for the simplex

The program for the simplex follows in the order of the segments. However segmilltime is omitted since it is given in 3.1.

3.2.1 Segconst

```
Segconst
BEGIN
     INT n, nfe := 0 ; read((n,newline)) ;
     {n is the number of dimensions}
     [0:n,1:n]REAL vertices ; read((vertices,newline)) ;
     REAL hypervolume ; read((hypervolume, newline)) ;
     INT npl = n+1 ; BOOL odd = ODD n ; SKIP
END
KEEP n, nfe, npl, vertices, hypervolume, odd
FINISH
```

3.2.2 Segcentroid

This segment contains a procedure to find the centroid of a simplex

```
Segcentroid
WITH segconst FROM album
BEGIN
     PROC findcentroid = (REF[]REAL centroid,
                                   REF[,]REAL vertices)VOID:
     BEGIN
          FOR i TO n DO
          BEGIN
               REAL temp := 0.0;
               FOR j FROM 0 TO n DO temp PLUS vertices[j,i] ;
               centroid[i] := temp/npl
```

```
          END
     END ; SKIP
END
KEEP findcentroid
FINISH
```

3.2.3 Segevala

This segment contains the first basic rule.

```
Segevala
WITH segconst FROM album
BEGIN
     REAL const3 = num/npl ;
     {evaluate and store the coordinates for Stroud's rule}
     [0:n,1:n]REAL nodesa ;
     FOR k FROM 0 TO n DO
     BEGIN
          REF[]REAL t = nodesa[k.1:n] ;
          FOR r TO n'/'2 DO
          BEGIN
               INT r2 = 2*r ; REAL theta = (r2*k*pi)/npl ;
               t[r2-1] := const1*cos(theta) ;
               t[r2]   := const1*sin(theta) ;
          END ;
          IF odd THEN t[n] := ((-1)^k)/const2 FI
     END ;
     PROC evala = (REAL div, scf,REF[]REAL centre,
                    PROC(REF[]REAL)REAL f)REAL:
     BEGIN
          REAL estimate := 0.0 ;
```

```
        FOR j FROM 0 TO n DO
        BEGIN
            [1:n]REAL temp ; REF[]REAL t2 = nodesa[j,1:n] ;
            FOR k TO n DO temp[k] := centre[k] + t2[k]/div ;
            estimate PLUS f(temp)
        END ;
        nfe PLUS npl ;
        estimate TIMES (const3/scf)
    END ;
    SKIP
END
KEEP evala
FINISH
```

3.2.4 Segevalb

This segment contains a procedure to apply the second basic rule.

```
Segevalb
WITH segconst FROM album
BEGIN
    INT n2 = 2*n ; REAL const4 = num/n2 ;
    {evaluate and store the coordinates for Stroud's 2n point rule}
    [1:n2,1:n]REAL nodesb ;
    FOR k TO n DO
    BEGIN
        REF[]REAL t1 = nodesb[k+n,1:n] ;
        REF[]REAL t  = nodesb[k,1:n] ;
        FOR r TO n'/'2 DO
        BEGIN
            INT r2 = 2*r ; INT r2m1 = r2 - 1 ;
```

```
            REAL theta = (r2ml*k*pi)/n ;
            t[r2ml] := constl*cos(theta) ;
            tl[r2ml] := - t[r2ml] ;
            tl[r2]  := constl*sin(theta) ;
            tl[r2]  := t[r2]
        END ;
        IF oddTHEN t[n] := (-1)^k/const2 ;
                   tl[n] := - t[n] FI
    END ;
    PROC evalb = (REAL div, scf,REF[]REAL centre,
                 PROC(REF[]REAL)REAL f)REAL:
    BEGIN
        REAL estimate := 0.0 ;
        FOR j TO n2 DO
        BEGIN
            [l:n]REAL temp ; REF[]REAL t2 = nodesb[j,l:n] ;
            FOR k TO n DO temp[k] := centre[k] + t2[k]/div ;
            estimate PLUS f(temp)
        END ;
        nfe PLUS n2 ;
        estimate TIMES (const4/scf)
    END ;
    SKIP
END
KEEP evalb
FINISH
```

3.2.5 The main body of the program

```
First simplex method
WITH segconst, segmilltime, segcentroid, segevala, segevalb, segfl2d
FROM album
BEGIN
     REAL tot, sum, latest, eps, neweps, subregions, consub, err ;
     MODE ELEMENT = STRUCT(REF ELEMENT ptr,
                          REF[,]REAL vertices,
                          REAL hypervolume);
     REF ELEMENT empty = NIL ;

     PROC simpest = (REFREFELEMENT head, newhead,
                    REAL eps)REAL :
     BEGIN
          REF[,]REAL v = vertices OF head ;
          [1:n]REAL c ;
          {c is the centroid of the simplex}
          findcentroid(c,v) ;
          REAL estimatea :=
                 evala(v,c,hypervolumeOFhead,f),
               estimateb :=
                 evalb(v,c,hypervolumeOFhead,f),
               diff ;
          diff := ABS(estimatea-estimateb) ;
          IF diff < eps
          THEN {convergence in this subregion}
               tot PLUS estimateb ;
               consub PLUS 1 ;
```

```
        err PLUS diff
    ELSE
        {subdivide the region}
        {find the longest side of the simplex}
        INT vl,v2 ; REAL longest ;
        FOR i FROM 0 TO n DO
        BEGIN
            REF[]REAL temp = v[i,] ;
            FOR j FROM (i+1) TO n DO
            BEGIN
                REF[]REAL t2 = v[j,] ;
                REAL length := 0.0 ;
                FOR m TO n DO
                length PLUS ((temp[m]-t2[m])^2) ;
                IF length ^ longest
                THEN longest := length;
                    vl := i ;
                    v2 := j
                FI
            END
        END ;
        {the longest side lies between vl and v2,find the
        midpoint of this}
        [l:n]REAL midpoint ;
        REF[]REAL tvl = v[vl,], tv2 = v[v2,] ;
        FOR i TO n DO
          midpoint[i] := (tvl[i]+tv2[i])/2 ;
        {set up the two subregions}
        [l:n]REAL temp ;
```

```
          temp := v[v1,] ;
          v[v1,] := midpoint ;
          REF ELEMENT temptr := newhead ;
          newhead := ELEMENT ;
          REF[,]REAL vert = [0:n,1:n]REAL := v ;
          ptr OF newhead := ELEMENT :=
             (temptr,vert,hypervolumOFhead/2) ;
          v[v1,] := temp ;
          v[v2,] := midpoint ;
          REF[,]REAL vert1 = [0:n,1:n]REAL := v ;
          verticesOFnewhead := vert1 ;
          hypervolumeOFnewhead := hypervolumeOFhead/2
     FI ;
     estimateb
END ;

INT number of tests ;
read((number of tests, newline)) ;
[1:number of tests]REAL epsa ;
read((epsa,newline)) ;
FOR i TO number of tests DO
BEGIN
     BOOL notconverged := TRUE ;
     LONG INT before, after ;
     REF[,]REAL verto = [0:n,1:n]REAL := vertices ;
     REF ELEMENT head := ELEMENT :=
          (empty,verto,hypervolume) ;
     REF ELEMENT newhead := empty ;
     neweps := eps := epsa[i] ;
```

```
    tot := 0.0 ; sum := 0.0 ; latest := 0.0;
    subregions := 1 ; consub := 0 ; err := 0.0;
    nfe := 0 ;
    before := milltime ;
    WHILE notconverged DO
    BEGIN
         latest := sum ;
         sum := tot ;
         WHILE head ISNT empty
         DO (sum PLUS simpest(head,newhead,neweps);
             head := ptrOFhead) ;
         IF newhead IS empty
         THEN notconverged := FALSE
         ELSE
              subregions TIMES 2 ;
              consub TIMES 2 ;
              neweps := (eps-err)/(subregions-consub);
              head := newhead ;
              newhead := empty
         FI
    END ;
    after := milltime ;

    {output the results}
    print(("the final estimate is ",sum,newline,
           "time ",after-before,newline,
           "integrand evaluations ",nfe,newline,
           "tolerance ",eps))
END
```

Appendix 4

The following are approximate timings for some basic operations in algol68r on the I.C.L.1904s. The timings are given in micro units, where 1000 micro units are equivalent to one millunit.

BOOL	assignment	3.5
INT	..	3.5
REAL	..	7.0
[]REAL	..	14.6
[,]REAL	..	30.0
[,,]REAL	..	45.4
[,,,]REAL	..	60.8
+		4.2
-		3.0
*		10.3
/		22.5
ABS		1.8
sqrt		165.1
sin		218.3
cos		215.1
tan		227.1
ln		218.8
exp		264.3
arcsin		410.0
arccos		416.4
arctan		226.6
a^2		72.5
a^3		85.9
a^4		97.7

Appendix 5

This section contains the segments defining the programs for the hypercube using stored integrand evaluation techniques.

5.1 The program using linked list techniques

5.1.1 Segconst

This segment contains the constants used in the other segments

```
Segconst
BEGIN
     INT n, nfe := 0, top, tfe := 0 ;
     read((n,top,newline)) ;
     INT num = 2^n ;
     REAL const1 = 1/3, const2 = 2*num/3 ;
     MODE NODE = STRUCT(INT index, REAL fevaluation,
                               REF NODE ptr);
     [1:top]REF NODE pointerlist ;
     REF NODE nil = NIL ;
     FOR i TO top DO pointerlist[i] := nil ;
     INT base := 2, lptr := 1 ;
     SKIP
END
KEEP n,nfe,tfe,num,NODE,pointerlist,nil,base,top,lptr
FINISH
```

5.1.2 Segmilltime

This segment contains a procedure to give the time of call. It is the same as the one given in appendix 3 and so is omitted.

5.1.3 Segllprocs

This segments contains the procedures concerned with the linked lists.

```
Segllprocs
WITH segconst FROM album
BEGIN
    {this segment contains the linked list procedures}
    PROC searchlist = (REFREFREF NODE pointer,
                    INT key) BOOL :
    BEGIN
        {this procedure searches a list for a node with a given
        key}
        BOOL notfound := TRUE, possible := TRUE;
        IF pointer ISNT nil
        THEN
        WHILE notfound AND possible DO
        BEGIN
            REF INT indp = indexOFpointer ;
            IF indp = key
            THEN notfound := FALSE
            ELSF indp > key
            THEN possible := FALSE
            ELSE IF (ptrOFpointer) IS nil
                THEN possible := FALSE
                FI ;
                pointer := ptrOFpointer
            FI
        END
        FI ;
```

```
        NOT notfound
    END ; {end of the procedure}


    PROC insert = (REFREFNODEpointer, INT key,
                   REAL feval) VOID :
    BEGIN
        {this procedure creates a new node nad inserts it in the
        list at the position indicated by pointer}
        REF NODE newnode = NODE ;
        indexOFnewnode := key ;
        fevaluationOFnewnode := feval ;
        ptrOFnewnode := pointer ;
        {link the newnode to the next node in the list}
        pointer := newnode
    END ;
    SKIP
END
KEEP searchlist, insert
FINISH
```

5.1.4 Segenumerate

This segment contains a procedure enumerate a key from a given node.

```
Segenumerate
WITH segconst FROM album
BEGIN
    PROC enumerate = (REF[]REAL point,
                      REF INT key, ptr,
                      INT b, REAL al)VOID:
```

```
    BEGIN
         BOOL possible := TRUE ;
         REAL alt := al ;
         INT base := b ;
         WHILE base > 2 AND possible DO
         BEGIN
              FOR i TO n DO
                IF ODD(ENTIER(point[i]/alt))
                THEN possible := FALSE
                FI;
              IF possible
              THEN
                 base := (base+1)'/'2 ;
                 alt TIMES 2 ;
                 ptr MINUS 1
              FI
         END ;
         key := 0 ;
         FOR i TO n DO
           key := key*base + ENTIER(point[i]/alt)
    END ;
    SKIP
END
KEEP enumerate
FINISH
```

5.1.5 Segevala

This segment contains the procedure to apply the first basic rule, the compound trapezoidal rule.

```
Segevala

WITH segconst, segllprocs, segenumerate FROM album

BEGIN

     [1:num,1:n]REAL nodes ;

     {set up the nodes for this rule}

     FOR i TO num DO

     BEGIN

          REF[]REAL node = nodes[i,1:n] ;

          BITS b := BIN i ;

          FOR j TO n DO

            IF (25-j)ELEM b

            THEN node[j] := 1

            ELSE node[j] := -1

            FI

     END ;

     PROC evala = (REF[]REAL centre, REAL alt,scf,

                   INT base, ptr, div,

                   PROC(REF[]REAL)REAL f)REAL:

     BEGIN

          REAL estimate := 0.0 ;

          FOR i TO num DO

          BEGIN

               INT ptrindex := ptr ;

               [1:n]REAL temp ;

               REF[]REAL t2 = nodes[i,1:n] ;

               FOR j TO n DO

                 temp[j] := centre[j] + t2[j]/div ;

               INT key := 0 ;

               enumerate(temp,key,ptrindex,base,alt*2);
```

```
            REF REF NODE pt := pointerlist[ptrindex] ;
            estimate PLUS
                (IF pt IS nil
                 THEN REAL tot := f(temp);
                      nfe PLUS 1 ;
                      insert(pt,key,tot) ;
                      tot
                 ELSF searchlist(pt,key)
                 THEN fevalautionOFpt
                 ELSE REAL tot := f(temp) ;
                      nfe PLUS 1 ;
                      insert(pt,key,tot) ;
                      tot
                 FI)
        END ;
        {increment the total number of integrand evaluations used}
        tfe PLUS num ;
        estimate DIV scf ;
        estimate
    END ;
    SKIP
END
KEEP evala
```

5.1.6 Segevalb

This segment contains the procedure to apply the second basic rule, Ewing's rule.

```
Segevalb
WITH segconst, segllprocs, segenumerate FROM album
```

```
BEGIN
     [1:num,1:n]REAL nodes ;
     {set up the nodes for this rule}
     FOR i TO num DO
     BEGIN
          REF[]REAL node = nodes[i,1:n] ;
          BITS b := BIN i ;
          FOR j TO n DO
            IF (25-j)ELEM b
            THEN node[j] := 1
            ELSE node[j] := -1
            FI
     END ;
     REAL const1 = 1/3, const2 = 2*num/3 ;
     PROC evalb = (REF[]REAL centre, REAL alt,scf,
                  INT base, ptr, div,
                  PROC(REF[]REAL)REAL f)REAL:
     BEGIN
          REAL estimate := 0.0 ;
          REF REF NODE pt ;
          INT ptrindex ;
          FOR i TO num DO
          BEGIN
               [1:n]REAL temp ;
               ptrindex := ptr ;
               REF[]REAL t2 = nodes[i,1:n] ;
               FOR j TO n DO
                 temp[j] := centre[j] + t2[j]/div ;
               INT key := 0 ;
```

```
        enumerate(temp,key,ptrindex,base,alt*2);
        pt := pointerlist[ptrindex] ;
        estimate PLUS
             (IF pt IS nil
              THEN REAL tot := f(temp);
                   nfe PLUS 1 ;
                   insert(pt,key,tot) ;
                   tot
              ELSF searchlist(pt,key)
              THEN fevalautionOFpt
              ELSE REAL tot := f(temp) ;
                   nfe PLUS 1 ;
                   insert(pt,key,tot) ;
                   tot
              FI)
   END ;
   estimate TIMES const1 ;
   estimate PLUS
      (INT key := 0;
       ptrindex := ptr ;
       enumerate(centre,key,ptrindex,base,alt*2);
       pt := pointerlist[ptrindex];
       const2*IF searchlist(pt,key)
              THEN fevaluationOFpt
              ELSE REAL tot := f(centre);
                   nfe PLUS 1 ;
                   insert(pt,key,tot);
                   tot
              FI);
```

```
        {increment the total number of integrand evaluations used}
        tfe PLUS (num+1) ;
        estimate DIV scf ;
        estimate
    END ;
    SKIP
END
KEEP evalb
```

5.1.7 The main body of the program

This is the main body of the program using linked lists to store the integrand evaluations.

Stored integrand evaluation program

```
WITH segconst,segmilltime,segllprocs,segevala,segevalb,segf FROM
albumname
BEGIN
    REAL tot:= 0.0, sum := 0.0, latest:= 0.0, eps, neweps, alt,
    subregions:= 1,scf,consub:=0.0, err:=0.0 ;
    INT div ;
    MODE NODE = STRUCT(REF NODE ptr, REF[,]REAL subcentres) ;

    PROC compute estimate = (REF REF NODE head, newhead,
                             REAL eps, alt, scf,
                             INT div, base,
                             ptr) REAL :
    BEGIN
        REAL result := 0.0 ;
        {result is the estimate delivered by the procedure}
        {Consider each subcentre in turn}
```

```
FOR i TO UPB subcentres OF head DO
BEGIN
     REF[]REAL centre = (subcentresOFhead)[i,1:n] ;
     {Let centre point to the next subcentre and form the
     two estimates to the integral over the subregion}
     REAL estimatea :=
     evala(centre,alt,scf,base,ptr,div,f),
          estimateb :=
     evalb(centre,alt,scf,base,ptr,div,f),
          diff ;
     {Set diff to be the difference between the two
     estimates}
     diff := ABS(estimatea - estimateb) ;
     result PLUS estimateb ;
     IF diff < eps  {test for convergence}
     THEN
          tot PLUS estimateb ;
          {add estimateb to the final estimate}
          consub PLUS 1 ;
          {Increment the number of converged subregions}
          err PLUS diff
          {Add the difference to the total error estimate}
     ELSE
          {Convergence not achieved in this subregion}
          REF NODE temptr := newhead ;
          newhead := NODE ; {declare a new node}
          ptrOFhead := temptr ;
          {link the new node to the list}
          REF[,]REAL centres = [1:num,1:n]REAL ;
```

```
                {Set up the subcentres of this subregion}
                FOR i TO num DO
                BEGIN
                     {Set up a pointer to the next subcentre}
                     REF[]REAL sub = centres[i, ];
                     BITS b := BIN i ;
                     {This section of the procedure uses the
                     bits pattern of the digit i to determine
                     the subcentres of the subregion

                     FOR j TO n DO
                          IF (25-j)ELEM b
                          THEN sub[j] := centre[j] + alt
                          ELSE sub[j] := centre[j] - alt
                          FI
                END ;
                subcentresOFnewhead := centres
          FI
     END ;
     head := ptrOFhead ;
     {Move on to the next node in the list}
     result
END ; {end of the procedure compute estimate}

read((eps,newline,alt,newline,div,newline,scf,newline) ;
REAL d ; read(d) ;
[1:1,1:n]REAL c ; {c is used to represent the centre}
FOR j TO n DO c[1,j] := d ;
REF NODE nil = NIL ;
```

```
REF NODE head := NODE := (nil,c) ;
REF NODE newhead := nil ;
BOOL notconverged := TRUE ;
base := 3 ; lptr := 2 ;
LONG INT before, after ;
{Start timing this test using the procedure milltime}
before := milltime ;
WHILE notconverged DO
BEGIN
          latest := sum ;{set latest to the most recent
          estimate}
          sum := tot ;
          {sum will be the present, it is set initially to the
          total estimate from converged subregions}
          {Now process the list of non converged subregions,
          adding the estimates over each to the present
          estimate sum}
          WHILE head ISNT nil    {while list not empty}
          DO sum PLUS compute
          estimate(head,newhead,neweps,alt,scf,div,base,lptr);
          {All estimates formed and added to sum. A new list of
          non converged subregions has been formed and is
          pointed to by newhead. Test for convergence.}
          IF (newhead IS nil)
             {list of non converged subregions is empty}
             OR (ABS(latest-sum)<eps)
             {the difference between the two most recent
          estimates
              is less than the required tolerance}
```

```
THEN notconverged := FALSE
     {solution found}
ELSE
     {Set up the variables to consider the next list
     of non converged subregions, ie consider the
     next level of subdivision}
     subregions TIMES num ;
     {subregions is equal to the total number of
     subregions at this level of subdivision}
     level of subdivision PLUS 1 ;
     consub TIMES num ;
     {Evaluate the number of converged subregions at
     this level of subdivision, ie consub}
     {Evaluate the tolerance to be applied to each
     subregion. This consists of dividing the
     tolerance minus the error estimate from the
     converged subregions between each of the non
     converged subregions, ie the total number of
     subregions minus the converged subregions at
     this level}
     neweps := (eps-err)/(subregions-consub) ;
     {Alter the scaling factors}
     div TIMES 2 ; alt DIV 2 ; scf TIMES num;
     {Replace the old list of non converged
     subregions by the new one and set the new one to
     empty}
     head := newhead ;
     newhead := nil ;
     base := base * 2 - 1 ;
```

```
            lptr PLUS 1
        FI
    END ;
    {Estimate formed for this test, finish timing}
    after := milltime ;
    {Output the information required from this test}
    print(("The final estimate to the integral is
    ",sum,newline,
     "The time taken to obtain the result was ",after-before,
     "The number of integrand evaluations was ",numeval,
     "The tolerance for this run was ",eps,
     "The level of subdivision was ",level of subdivision))
END
FINISH
```

5.2 The program using the scatter stoarge techniques

This section contains the segments used in the scatter storage programs.

5.2.1 Segconst

This segment contains the constants used in the other segments.

```
Segconst
BEGIN
     INT n ; {n is the number of dimensions}
     read((n,newline)) ;
     INT npl = n+1, num = 2^n ;
     INT numeval := 0, hashvalue, key1, key2 ;
     REAL const1 = 1/3, const2 = 2*num/3 ;
     MODE ITEM = STRUCT(LONG INT key, REAL feval,
                        REF ITEM ptr) ;
     [0:1023]REF ITEM scatter index table ;
     FOR i FROM 0 TO 1023 DO
       scatter index table[i] := NIL ;
     [0:1023]INT hash table ;
     LONG INT newkey ;
     REF REF ITEM pointer ;
     INT next := 0 ;
     INT const = (IF n=2 THEN 1048576
                         ELSE 8192 FI),
         maxint = 4194304 ;
     SKIP
END
KEEP n,npl,num,numeval,hash
```

value,key1,key2,const1,const2,ITEM,scatter index table,hash table,newkey,pointer,maxint,next,const
FINISH

5.2.2 Segmilltime

This segment contains the procedure to give the time of call and is described in appendix 3.

5.2.3 Segprocs

This segment contains all the procedures associated with the scatter storage technique.

```
Segprocs
WITH segconst FROM album
     REF ITEM empty = NIL ;
     {procedure to compute a hash value}
     PROC compute hash = (INT key1, key2) INT:
     BEGIN
          ENTIER(1024*ABS((key1+key2)/maxint))
     END ;
     {procedure to compute the key}
     PROC compute keys = (REF INT key1, key2,
                          REF[]REAL x) VOID :
     BEGIN
          CASE (n-1)
          IN
          ({2d problem}
           key1 := ENTIER(x[1]*const) ;
           key2 := ENTIER(x[2]*const) ),
          ({3d problem}
```

```
      BITS b ;
      INT digit := ENTIER(x[3]*const) ;
      b := BIN digit ;
      key1 := ABS((bSR8)SL15)
              OR (BIN(ENTIER(x[2]*const)));
      key2 := ABS((bSL15)
              OR (BIN(ENTIER(x[1]*const))))
     OUT SKIP
      {other dimensions not included}
     ESAC
END ;
{procedure to evaluate a key}
PROC evalkey = (INT key1,key2) LONG INT:
BEGIN
     LONG INT newkey ;
     newkey := newkey * (LONG 10000000);
     newkey := newkey + (LENG key1) ;
     newkey
END ;
{procedure to search a list}
PROC searchlist = (REFREFREF ITEM pointer,
                    LONG INT key) BOOL :
BEGIN
     BOOL notfound := TRUE, possible := TRUE ;
     IF pointer ISNT empty
     THEN
       WHILE notfound AND possible DO
       BEGIN
          REF LONG INT indp = keyOFpointer ;
```

```
            IF indp = key
            THEN notfound := FALSE
             {searchkey = key of item}
            ELSE indp > key
            THEN possible := FALSE
             {item cannot be in the list}
            ELSE IF (ptrOFpointer) IS empty
                THEN possible := FALSE
                 {no more items in the list}
                FI;
                pointer := ptrOFpointer
                {move on to the next item}
            FI
          END
        FI ;
        NOT notfound
        {deliver true if the item is in the list and false if it
        is not in the list}
    END ;
    {procedure to insert an item in a list}
    PROC insert = (REF REF ITEM pointer,
                   LONG INT key,
                   REAL feval) VOID :
    BEGIN
        REF ITEM newitem = ITEM ;
        keyOFnewitem := key ;
        fevalOFnewitem := feval ;
        ptrOFnewitem := pointer ;
        pointer := newitem
```

```
END ;
{procedure to find the value of the integrand at a given node
either by looking it up or by evaluating it}
PROC integrand evaluation = (REF[]REAL node,
            PROC(REF[]REAL)REAL f)REAL:
BEGIN
     REAL val ;
     compute keys(key1,key2,node) ;
     newkey := evalkey(key1,key2) ;
     hash value := comput hash(key1,key2) ;
     REF INT htv = hash table[hash value] ;
     IF htv = 0
     THEN htv := next PLUS 1 ;
          val := f(node) ;
          numeval PLUS 1 ;
          insert(scatter index table[htv],
                newkey, val)
     ELSE pointer := scatter index table[htv];
          IF searchlist(pointer,newkey)
          THEN val := fevalOFpointer
          ELSE val := f(node) ;
               numeval PLUS 1 ;
               insert(pointer,newkey,val)
          FI
     FI ;
     val
END ;
SKIP
END
```

```
KEEP integrand evaluation, compute hash, compute keys, evalkey
FINISH
```

5.2.4 Segevala

This segment contains the procedure to apply the first basic rule, the compound trapezoidal rule.

```
Segevala
WITH segconst, segprocs FROM album
BEGIN
[1:num,1:n]REAL nodes ;
{set up the nodes for this rule}
FOR i TO num DO
BEGIN
     REF[]REAL node = nodes[i,1:n] ;
     BITS b := BIN i ;
     FOR j TO n DO
       IF (25-j)ELEM b
       THEN node[j] := 1
       ELSE node[j] := -1
       FI
END ;
PROC evala = (REF[]REAL centre, REAL div, scf
              PROC(REF[]REAL)REAL f)REAL:
BEGIN
     REAL estimate := 0.0 ;
     FOR i TO num DO
     BEGIN
          [1:n]REAL temp ;
          REF[]REAL t2 = nodes[i,1:n] ;
```

```
          FOR j TO n DO
            temp[j] := centre[j] + t2[j]/div ;
          compute keys(key1,key2,temp) ;
          estimate PLUS integrand evaluation(temp,f)
        END ;
        {increment the total number of integrand evaluations used}
        tfe PLUS num ;
        estimate DIV scf ;
        estimate
    END ;
    SKIP
END
KEEP evala
```

5.2.5 Segevalb

This segment contains the procedure to apply the second basic rule, Ewing's rule.

```
Segevalb
WITH segconst, segprocs FROM album
BEGIN
    [1:num,1:n]REAL nodes ;
    {set up the nodes for this rule}
    FOR i TO num DO
    BEGIN
        REF[]REAL node = nodes[i,1:n] ;
        BITS b := BIN i ;
        FOR j TO n DO
          IF (25-j)ELEM b
          THEN node[j] := 1
```

```
        ELSE node[j] := -1
         FI
     END ;
     REAL const1 = 1/3, const2 = 2*num/3 ;
     PROC evalb = (REF[]REAL centre, REAL div, scf,
                   PROC(REF[]REAL)REAL f)REAL:
     BEGIN
          REAL estimate := 0.0 ;
          FOR i TO num DO
          BEGIN
               [1:n]REAL temp ;
               REF[]REAL t2 = nodes[i,1:n] ;
               FOR j TO n DO
                 temp[j] := centre[j] + t2[j]/div ;
               estimate PLUS integrand evaluation(temp,f)
          END ;
          estimate TIMES const1 ;
          estimate PLUS
            (integrand evaluation(centre,f)*const2) ;
          {increment the total number of integrand evaluations used}
          tfe PLUS (num+1) ;
          estimate DIV scf ;
          estimate
     END ;
     SKIP
END
KEEP evalb
```

5.2.6 The main body of the program

This is exactly the same as the main body of the program for the previous program (5.1.7) and so is not repeated

Appendix 6

This section contains the details of the segments that are used in the global subdivision strategy program for the hypercube. The segments containing the procedures to apply the two basic rules, segevala and segevalb, are the same as those given in appendix 3 and so are not repeated. Segconst merely contains the constants used by the other segments and inputs the number of dimensions of the problem. The main body of the program consists of :

```
Global subdivision strategy
WITH segconst, segmilltime, segevala,segevalb, segfl2d FROM
albumname
BEGIN
     REAL altl, tot, err, divl, scfl, centre, consub ;
     INT number of tests ;
     read((number of tests, newline)) ;
     [1pnumber of tests]REAL epsa ;
     read((epsa,newline,divl,newline,altl,newline,
            scfl,newline,centre,newline)) ;
     {output the details of this testrun}
     print (("testing ..."));
     {set up the mode for the nodes in the subregion list}
     MODE NODE = STRUCT(REF NODE ptr, REF[]REAL centre,
                        REAL errest, estimate, INT level)  ;
     REF NODE empty = NIL ;
     REF NODE head ;
     {the following is a procedure to add a node to a list at a
     position dependant upon the magnitude of it s error estimate}
```

```
PROC addtolist = (REF NODE newnode) VOID :
BEGIN
     REF REF NODE temptr := head ;
     IF temptr ISNT empty
     THEN
          {search the list for the position of insertion}
          REF REAL e = errest OF newnode ;
          BOOL possible := TRUE ;
          WHILE possible AND (errestOFtemptr > e DO
          BEGIN
               IF (ptrOFtemptr) IS empty
               THEN possible := FALSE
               FI ;
               temptr := ptrOFtemptr
          END
     FI ;
     {position of insertion found}
     ptrOFnewnode := temptr ;
     (REF REF NODE VAL temptr) := newnode
END ;
{set up the data structures to hold the scfs, divs and alts
associated with the level of subdivision}
INT toplevel = 50 ;
[0:toplevel]REAL altlev, divlev, scflev ;
REAL presentalt,presentdiv, presentscf ;
altlev[0] := altl * 2 ;
divlev[0] := divl ;
scflev[0] := scfl ;
FOR i TO number of tests DO
```

```
BEGIN
     REAL eps := epsa[i], subregions := 1 ;
     presentalt := altlev[0] ;
     presentdiv := divl ;
     presentscf := scfl ;
     LONG INT before, after ;
     {set up the initial list of subregions and the level  of
     subdivision}
     INT maxlevel := 0 ;
     {maximum level of subdivision so far}
     [1:n]REAL c ;
     FOR j TO n DO c[j] := centre ;
     before := milltime ;
     REAL estimatea := evala(presentdiv,presentscf,c,f),
          estimatea := evala(presentdiv,presentscf,c,f);
     total error estimate := ABS(estimatea - estimateb) ;
     total result := estimateb ;
     head := NODE := (empty,c,total,error estimate,estimateb,1)
     ;
     {while convergence has not been achieved perform the
     process}
     WHILE total error estimate > eps DO
     BEGIN
          {consider the subregion with the largest error
          estimate}
          REAL talt, tscf, tdiv ; INT nextlev ;
          total error estimate MINUS errestOFhead ;
          total result MINUS estimateOFhead ;
          {set up the scaling factors for the subregions of
```

```
this subregion}
IF levelOFhead > maxlevel
THEN maxlevel PLUS 1 ;
     presentscf TIMES num ;
     scflev[levelOFhead] := presentscf ;
     presentalt DIV 2 ;
     altlev[levelOFhead] := presentalt ;
     presentdiv TIMES 2 ;
     divlev[levelOFhead] := presentdiv ;
FI ;
talt := altlev[levelOFhead] ;
tscf := scflev[levelOFhead] ;
tdiv := divlev[levelOFheaD] ;
nextlev := levleOFhead + 1 ;
{consider each subregion in turn}
FOR j TO num DO
BEGIN
REF[]REAL tc = [1:n]REAL ;
REF NODE newnode := NODE ;
levelOFnewnode := nextlev ;
{set up the centre for this subregion}
BITS b := BIN j ;
FOR k TO n DO
tc[k] := (centreOFhead)[k] +
         (IF (25-k)ELEM b
          THEN talt
          ELSE -talt
          FI) ;
{evaluate the estimates for this subregion}
```

```
            estimateOFnewnode := evalb(tdiv,tscf,tc,f) ;
            errestOFnewnode := ABS(evala(tdiv,tscf,tc,f) -
                                        estimateOFnewnode) ;
            centreOFnewnode := tc ;
            {add the newnode to the list of subregions}
            addtolist(newnode) ;
            total error estimate PLUS errestOFnewnoe ;
            total result PLUS estimateOFnewnode
        END ;
        {now the estimate for the subregion at the head of the
        list has been replaced by the sum of the estimates over
        the subregions of that subregion, which now needs to be
        removed from the list}
        head := ptrOFhead
    END ;
    after := milltime ;
    {output the results for this run}
    print(("... "));
    numeval := 0
END
END
FINISH
```

Appendix 7

This appendix contains the segments which make up the extended region global subdivision strategy program. Several of the segments are the same as those used in other programs and so are not repeated. Segcentroid, segevala and segevalb are described in appendix 3.2. Segmilltime is described in appendix 3.1 and segevalc and segevald are equivalent to segevala and segevalb from that appendix. The procedure names used in these two segments are reffered to as evalc and evald instead of evala and evalb in the following program. !1j

6.1 Segconst

This segment contains the constants used in the rest of the segments.

```
Segconst
BEGIN
     INT n ;
     read((n,newline));
     INT npl = n+l , num = 2^n ;
     INT numeval := 0 , nfe := 0 ;
     REAL constl = sqrt(2/3), const2 = sqrt(3) ;
     BOOL odd = ODD n ;
     SKIP
END
KEEP n,npl,num,constl,const2,numeval,odd,nfe
FINISH
```

6.2 The main body of the program

Global subdivision strategy for an extended region

```
WITH segconst, segmilltime, segcentroid, segevala, segevalb
     segevalc, segevald, segfl2d FROM album
BEGIN
     REAL eps ; read((eps,newline));

     {output the details of this testrun}
     print(("Testing ...",newline));

     CHAR type ;
     MODE NEXTITEM ;
     MODE SIMP = STRUCT(REAL errest, hypervolume, estimate,
                        REF[,]REAL vertices,
                        REF NEXTITEM ptr) ;
     MODE HYP = STRUCT(REAL errest, div, estimate, scf, alt,
                       REF[]REAL centre,
                       REF NEXTITEM ptr);
     MODE NEXTITEM = UNION(REF SIMP, REF HYP);
     REF NEXTITEM end of list = NIL ;
     REF NEXTITEM head := end of list ;
     REF SIMP s ; REF HYP h ;
     {procedure to add an item to the list}
     PROC addtolist = (REAL e, REF NEXTITEM newnode) VOID :
     BEGIN
          REF REF NEXTITEM temptr := head ;
          REF SIMP s ; REF HYP h ;
          IF head ISNT end of list
          THEN {the list isnt empty search for
```

```
     the position of insertion }
    BOOL notfound := TRUE ;
    WHILE notfound DO
    BEGIN
         CASE (s,h) ::= temptr
         IN
         (IF errestOFs > e
          THEN temptr := ptrOFs ;
                IF ptrOFs IS end of list
                THEN notfound := FALSE
                FI
          ELSE notfound := FALSE
          FI) ,
         (IF errestOFh > e
          THEN temptr := ptrOFh ;
                IF ptrOFh IS end of list
                THEN notfound := FALSE
                FI
          ELSE notfound := FALSE
          FI)
         ESAC
    END
FI ;
{position of insertion found}
CASE (s,h) ::= newnode
IN
 (ptrOFs := temptr),
 (ptrOFh := temptr)
ESAC ;
```

```
    (REF REF NEXTITEM VAL temptr) := newnode
END;

REAL total := 0.0, total error estimate := 0.0 ;
WHILE (read((type,newline)) ;
      type <> "E") DO
BEGIN
    REF NEXTITEM nextnode = NEXTITEM ;
    IF type = "S"
    THEN
         REF SIMP newnode = SIMP ;
         REF[,]REAL v = [0:n,1:n]REAL ;
         [1:n]REAL cent ;
         read((v,newline,hypervolumeOFnewnode,newline));
         verticesOFnewnode := v ;
         {find the centroid of this subregion}
         findcentroid(cent,v) ;
         REAL                    estimatea               :=
         evala(v,cent,hypervolumOFnewnode,f),
             estimateb                           :=
         evalb(v,cent,hypervolumeOFnewnode,f),
             diff ;
         diff := ABS(estimatea-estimateb) ;
         errestOFnewnode := diff ;
         estimateOFnewnode := estimateb ;
         total PLUS estimateb ;
         total error estimate PLUS diff ;
         nextnode := newnode ;
         {add this new subregion to the list}
```

```
          addtolist(diff,nextnode)
     ELSE
          REF HYP newnode = HYP ;
          REF[]REAL cent = [1:n]REAL ;
          read((cent,newline,scfOFnewnode,newline,
                divOFnewnode,newline,altOFnewnode,
          newline));
          centreOFnewnode := cent ;
          REAL estimatec := evalc(divOFnewnode,
                                  scfOFnewnode,cent,f),
               estimated := evald(divOFnewnode,
                                  scfOFnewnode,cent,f),diff;
          diff := ABS(estimatec-estimated);
          errestOFnewnode := diff ;
          estimateOFnewnode := estimated ;
          total PLUS estimated ;
          total error estimate PLUS diff ;
          nextnode := newnode ;
          addtolist(diff,nextnode)
     FI
END ;

LONG INT before, after ;
before := milltime;
{while convergence has not been achieved
perform the process}
WHILE total error estimate > eps DO
BEGIN
     {consider the subregion with the largesterror estimate}
```

```
{determine the type of subregion}
CASE (s,h) ::= head
IN
({the subregion is a simplex}
 INT v1,v2 ; REAL longest ;
 total MINUS estimateOfs ;
 total error estimate MINUS errestOFs ;
 {subdivide the subregion}
 {find the largest side of the simplex}
 FOR i FROM 0 TO n DO
 BEGIN
     REF[]REAL temp = (verticesOFs)[i,] ;
     FOR j FROM i+1 TO n DO
     BEGIN
          REF[]REAL t2 = (verticesOFs)[j,] ;
          REAL length := 0.0 ;
          FOR m TO n
          DO length PLUS ((temp[m] - t2[m])^2) ;
          IF length > longest
          THEN longest := length;
               v1 := i ; v2 := j
          FI
     END
END ;
{the longest side lies between vertices v1 and v2}
{find the midpoint of the longest side}
[1:n]REAL midpoint ;
REF[]REAL tv1 = (verticesOFs)[v1,],
          tv2 = (verticesOFs)[v2,];
```

```
FOR i TO n
DO midpoint[i] := (tv1[i]+tv2[i])/2 ;
{set up the two subregions}
[1:n]REAL tempv ;
tempv := (verticesOFs)[v1,] ;
(verticesOFs)[v1,] := midpoint ;
REFNEXTITEM nextnode = NEXTITEM ;
REF SIMP newnode = SIMP ;
REF[,]REAL vert = [0:n,1:n]REAL ;
vert := verticesOFs ;
verticesOFnewnode := vert ;
[1:n]REAL cent ;
findcentroid(cent,vert) ;
hypervolumeOFnewnode := (hypervolumeOFs)/2 ;
REAL estimatea := evala(vert,cent,hypervolumeOFnewnode,f),
     estimateb := evalb(vert,cent,hypervolumeOFnewnode,f),
     diff ;
diff := ABS (estimatea -estimateb) ;
errestOFnewnode := diff ;
estimateOFnewnode := estimateb ;
total PLUS estimateb ;
total error estimate PLUS diff ;
nextnode := newnode ;
addtolist(diff,nextnode) ;
{set up the second subregion}
(verticesOFs)[v1,] := tempv ;
(verticesOFs)[v2,] := midpoint ;
REFNEXTITEM nextnode2 = NEXTITEM ;
REF SIMP newnode2 = SIMP ;
```

```
REF[,]REAL vert2 = [0:n,1:n]REAL ;
vert2 := verticesOFs ;
verticesOFnewnode2 := vert2 ;
findcentroid(cent,vert2) ;
hypervolumeOFnewnode2 := (hypervolumeOFs)/2 ;
estimatea := evala(vert2,cent,hypervolumeOFnewnode,f) ;
estimateb := evalb(vert2,cent,hypervolumeOFnewnode,f) ;
diff := ABS (estimatea -estimateb) ;
errestOFnewnode2 := diff ;
estimateOFnewnode2 := estimateb ;
total PLUS estimateb ;
total error estimate PLUS diff ;
nextnode2 := newnode2 ;
addtolist(diff,nextnode2) ;
head := ptrOFs ),
({the subregion is a hypercube}
total MINUS estimateOFh ;
total error estimate MINUS errestOFh ;
FOR i TO num DO
BEGIN
REF NEXTITEM nextnode = NEXTITEM ;
REF HYP newnode = HYP ;
nextnode := newnode ;
REF[]REAL sub = [1:n]REAL ;
BITS b := BIN i ;
FOR j TO n DO
 IF (25-j)ELEM b
 THEN sub[j] := (centreOFh)[j] + altOFh
 ELSE sub[j] := (centreOFh)[j] - altOFh
```

```
        FI ;
       centreOFnewnode := sub ;
       divOFnewnode := (divOFh)*2 ;
       altOFnewnode := (altOFh)/2 ;
       scfOFnewnode := (scfOFh)*num ;
       REAL estimatec := evalc(divOFnewnode,scfOFnewnode,sub,f),
            estimated := evald(divOFnewnode,scfOFnewnode,sub,f),
            diff ;
       diff := ABS(estimatec - estimated) ;
       estimateOFnewnode := estimated ;
       errestOFnewnode := diff ;
       total PLUS estimated ;
       total error estimate PLUS diff ;
       addtolist(diff,nextnode)
    END ;
    head := ptrOFh)
    ESAC
    END ;
    after := milltime ;
    {output the results}
    print(("....."))
    END
END
FINISH
```

Appendix 8

The following pages contain the results for the set of test problems described in appendix 1.2. In the tables of results the various programs are indicated by:

z-p1 : the basic adaptive program using Stroud's rules, as described in chapter 4.

z-p6 : the same program as above, but using the compound trapezoidal rule and Ewing's rule.

z-p5 : the program which uses linked lists to store the integrand evaluations, as described in chapter 5.

z-p7 : the program which uses scatter storage techniques to store the integrand evaluations. This program is described in chapter 5.

z-gss : the program based upon a global subdivision strategy, as described in chapter 6.

z-p10 : the iterative program based upon product Patterson formulae.

The milltime is given in millunits where 1 millunit is equal to 1000 micro units. These timings are used for comparison only.

Test Problem SEGF12D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-p1	0.977975186	0.002814053	8	7		
z-p6	0.951184463	0.023976670	9	9		
z-p5	0.951184463	0.023976670	21	9	5	4
z-p7	0.951184463	0.023976670	21	9	5	4
z-gss	0.977975186	0.002814053	8	7		
z-p10	0.975162070	0.000000937	77	49		
Tolerance		0.1				
z-p1	0.977975186	0.002814053	8	7		
z-p6	0.951184463	0.023976670	10	9		
z-p5	0.951184463	0.023976670	21	9	5	4
z-p7	0.951184463	0.023976670	21	9	5	4
z-gss	0.977975186	0.002814053	8	7		
z-p10	0.975162070	0.000000937	77	49		
Tolerance		0.05				
z-p1	0.977975186	0.002814053	8	7		
z-p6	0.970759029	0.004402104	45	45		
z-p5	0.970759029	0.004402104	115	45	13	32
z-p7	0.970759029	0.004402104	109	45	13	32
z-gss	0.977975186	0.002814053	8	7		
z-p10	0.975162070	0.000000937	77	49		

Test Problem SEGF12D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-p1	0.977975186	0.002814053	8	7		
z-p6	0.974356415	0.000804718	173	153		
z-p5	0.974219356	0.000941777	211	81	21	60
z-p7	0.974219356	0.000941777	198	81	21	60
z-gss	0.977975186	0.002814053	8	7		
z-p10	0.975162070	0.000000937	77	49		
Tolerance		0.005				
z-p1	0.977975186	0.002814053	8	7		
z-p6	0.974371350	0.000789783	218	189		
z-p5	0.974219356	0.000941777	211	81	21	60
z-p7	0.974219356	0.000941777	197	81	21	60
z-gss	0.977975186	0.002814053	8	7		
z-p10	0.975162070	0.000000937	77	49		
Tolerance		0.001				
z-p1	0.975318624	0.000157491	78	63		
z-p6	0.975020763	0.000140370	904	765		
z-p5	0.974983055	0.000178078	672	225	49	176
z-p7	0.974983055	0.000178078	559	225	49	176
z-gss	0.975717695	0.000556562	44	35		
z-p10	0.975162070	0.000000937	77	49		

Test Problem SEGF12D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations total	actual	reused
Tolerance		0.0005				
z-p1	0.975318624	0.000157491	78	63		
z-p6	0.975136183	0.000024950	2839	2313		
z-p5	0.975115419	0.000045714	1021	333	71	262
z-p7	0.975115419	0.000045714	832	333	71	262
z-gss	0.975318624	0.000157491	78	63		
z-p10	0.975162070	0.000000937	77	49		
Tolerance		0.0001				
z-p1	0.975184703	0.000023570	289	231		
z-p6						
z-p5	0.975128330	0.000032803	2577	693	135	558
z-p7	0.975128330	0.000032803	1931	693	135	558
z-gss	0.975193379	0.000032246	230	175		
z-p10	0.975161131	0.000000002	357	225		
Tolerance		0.00005				
z-p1	0.975167719	0.000006586	496	399		
z-p6						
z-p5	0.975152297	0.000008836	3693	909	175	734
z-p7	0.975152297	0.000008836	2487	909	175	734
z-gss	0.975172232	0.000011099	351	259		
z-p10	0.975161131	0.000000002	357	225		

Test Problem SEGF12D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations total	actual	reused
Tolerance		0.00001				
z-pl	0.975162150	0.000001017	1432	1155		
z-p6						
z-p5	0.975159369	0.000001764	15135	2205	407	1798
z-p7	0.975159369	0.000001764	6699	2205	407	1798
z-gss	0.975162994	0.000001861	1150	735		
z-pl0	0.975161133	0.000000000	1499	961		
Tolerance		0.000005				
z-pl	0.975161880	0.000000747	2232	1799		
z-p6						
z-p5	0.975159874	0.000001259	29795	3249	585	2664
z-p7	0.975159874	0.000001259	10117	3249	585	2664
z-gss	0.975161804	0.000000671	1962	1127		
z-pl0	0.975161133	0.000000000	11719	3969		
Tolerance		0.000001				
z-pl	0.975161255	0.000000122	6596	5327		
z-p6						
z-p5						
z-p7	0.975160852	0.000000281	19939	5697	1013	4684
z-pl0						
z-gss	0.975161220	0.000000087	8865	3255		

Test Problem SEGF13D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-p1	1.206755133	0.001098272	14	10		
z-p6	1.190442059	0.015214802	22	17		
z-p5	1.190442059	0.015214802	52	17	9	8
z-p7	1.190442059	0.015214802	49	17	9	8
z-gss	1.206755133	0.001098272	14	10		
z-p10						
Tolerance		0.1				
z-p1	1.206755133	0.001098272	14	10		
z-p6	1.190442059	0.015214802	22	17		
z-p5	1.190442059	0.015214802	52	17	9	8
z-p7	1.190442059	0.015214802	49	17	9	8
z-gss	1.206755133	0.001098272	14	10		
z-p10						
Tolerance		0.05				
z-p1	1.206755133	0.001098272	14	10		
z-p6	1.204112993	0.001543868	199	153		
z-p5	1.204112993	0.001543868	573	153	35	118
z-p7	1.204112993	0.001543868	432	153	35	118
z-gss	1.206755133	0.001098272	14	10		
z-p10						

Test Problem SEGF13D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-pl	1.206755133	0.001098272	14	10		
z-p6	1.205464263	0.000192598	916	697		
z-p5	1.204112993	0.001543868	573	153	35	118
z-p7	1.204112993	0.001543868	432	153	35	118
z-gss	1.206755133	0.001098272	14	10		
z-pl0						
Tolerance		0.005				
z-pl	1.206755133	0.001098272	14	10		
z-p6	1.205502245	0.000154616	1466	1105		
z-p5	1.205321344	0.000335517	1114	289	61	228
z-p7	1.205321344	0.000335517	835	289	61	228
z-gss	1.206755133	0.001098272	14	10		
z-pl0						
Tolerance		0.001				
z-pl	1.206755133	0.001098272	14	10		
z-p6	1.205642775	0.000014086	13279	9809		
z-p5	1.205464263	0.000192598	3958	697	124	573
z-p7	1.205464263	0.000192598	2120	697	124	573
z-gss	1.206755133	0.001098272	14	10		
z-pl0						

Test Problem SEGF13D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.0005				
z-p1	1.206755133	0.001098272	14	10		
z-p6						
z-p5	1.205614025	0.000042836	9380	215		
z-p7	1.205614025	0.000042836	4272	215		
z-gss	1.206755133	0.001098272	14	10		
z-p10						
Tolerance		0.0001				
z-p1	1.205714285	0.000057424	241	170		
z-p6						
z-p5	1.205626658	0.000030203	12658	1785	278	1507
z-p7	1.205626658	0.000030203	5682	1785	278	1507
z-gss	1.205798812	0.000141951	130	90		
z-p10						
Tolerance		0.00005				
z-p1	1.205714285	0.000057424	241	170		
z-p6						
z-p5	1.205637770	0.000019091	42260	3961	571	3390
z-p7	1.205637770	0.000019091	13968	3961	571	3390
z-gss	1.205798812	0.000141951	130	90		
z-p10						

Test Problem SEGF13D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.00001				
z-pl	1.205671980	0.000015119	1149	810		
z-p6						
z-p5						
z-p7						
z-gss	1.205682887	0.000026026	623	410		
z-pl0						
Tolerance		0.000005				
z-pl	1.205661349	0.000004488	2169	1530		
z-p6						
z-p5						
z-p7						
z-gss	1.205669393	0.000012532	1025	650		
z-pl0						
Tolerance		0.000001				
z-pl	1.205657887	0.000001026	7052	4970		
z-p6						
z-p5						
z-p7						
z-pl0						
z-gss	1.205697029	0.000002841	3655	1930		

Test Problem SEGF14D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-p1	1.398860237	0.000679659	20	13		
z-p6	1.388923304	0.009257274	49	33		
z-p5	1.388923304	0.009257274	133	33	17	16
z-p7						
z-gss	1.398860237	0.000679659	20	13		
z-p10						
Tolerance		0.1				
z-p1	1.398860237	0.000679659	20	13		
z-p6	1.388923304	0.009257274	49	33		
z-p5	1.388923304	0.009257274	133	33	17	16
z-p7						
z-gss	1.398860237	0.000679659	20	13		
z-p10						
Tolerance		0.05				
z-p1	1.398860237	0.000679659	20	13		
z-p6	1.397595164	0.000585414	851	561		
z-p5	1.397595164	0.000585414	3689	561	97	464
z-p7						
z-gss	1.398860237	0.000679659	20	13		
z-p10						

Test Problem SEGF14D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-p1	1.398860237	0.000679659	20	13		
z-p6	1.397595164	0.000585414	918	561		
z-p5	1.397595164	0.000585414	3689	561	97	464
z-p7						
z-gss	1.398860237	0.000679659	20	13		
z-p10						
Tolerance		0.005				
z-p1	1.398860237	0.000679659	20	13		
z-p6	1.398141777	0.000038801	13083	8481		
z-p5	1.397595164	0.000585414	3689	561	97	464
z-p7						
z-gss	1.398860237	0.000679659	20	13		
z-p10						
Tolerance		0.001				
z-p1	1.398860237	0.000679659	20	13		
z-p6						
z-p5	1.397978410	0.000202168	7447	1089	177	912
z-p7						
z-gss	1.398860237	0.000679659	20	13		
z-p10						

Test Problem SEGF14D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.0005				
z-pl	1.398860237	0.000679659	20	13		
z-p6						
z-p5						
z-p7						
z-gss	1.398860237	0.000679659	20	13		
z-pl0						
Tolerance		0.0001				
z-pl	1.398249065	0.000068487	359	221		
z-p6						
z-p5						
z-p7						
z-gss	1.398249065	0.000068487	359	221		
z-pl0						
Tolerance		0.00005				
z-pl	1.398222054	0.000041476	699	429		
z-p6						
z-p5						
z-p7						
z-gss	1.398249065	0.000068487	359	221		
z-pl0						

Test Problem SEGF14D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.00001				
z-p1	1.398199665	0.000019087	2394	1469		
z-p6						
z-p5						
z-p7						
z-gss	1.398222054	0.000041476	699	429		
z-p10						
Tolerance		0.000005				
z-p1	1.398187822	0.000007244	5773	3549		
z-p6						
z-p5						
z-p7						
z-gss	1.398200859	0.000020281	2274	1261		
z-p10						
Tolerance		0.000001				
z-p1	1.398181935	0.000001357	33344	20397		
z-p6						
z-p5						
z-p7						
z-p10						
z-gss	1.398185099	0.000004521	8182	3757		

Test Problem SEGF22D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-p1	0.444036916	0.000407528	8	7		
z-p6	0.416666666	0.027777778	10	9		
z-p5	0.416666666	0.027777778	21	9	5	4
z-p7	0.416666666	0.027777778	21	9	5	4
z-gss	0.444036916	0.000407528	8	7		
z-p10	0.444634014	0.000189570	78	49		
Tolerance		0.1				
z-p1	0.444036916	0.000407528	8	7		
z-p6	0.432429799	0.012014645	48	45		
z-p5	0.432429799	0.012014645	115	45	13	32
z-p7	0.432429799	0.012014645	115	45	13	32
z-gss	0.444036916	0.000407528	8	7		
z-p10	0.444634014	0.000189570	78	49		
Tolerance		0.05				
z-p1	0.444036916	0.000407528	8	7		
z-p6	0.432429799	0.012014645	48	45		
z-p5	0.432429799	0.012014645	115	45	13	32
z-p7	0.432429799	0.012014645	108	45	13	32
z-gss	0.444036916	0.000407528	8	7		
z-p10	0.444634014	0.000189570	78	49		

Test Problem SEGF22D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-p1	0.444590168	0.000145724	43	35		
z-p6	0.439870077	0.004574367	223	189		
z-p5	0.439853877	0.004590567	448	153	35	118
z-p7	0.439853877	0.004590567	380	153	35	118
z-gss	0.444590168	0.000145724	43	35		
z-p10	0.444634014	0.000189570	78	49		
Tolerance		0.005				
z-p1	0.444590168	0.000145724	43	35		
z-p6	0.442779912	0.001664532	719	621		
z-p5	0.439853877	0.004590567	448	153	35	118
z-p7	0.439853877	0.004590567	380	153	35	118
z-gss	0.444659322	0.000214878	80	63		
z-p10	0.444634014	0.000189570	78	49		
Tolerance		0.001				
z-p1	0.444590168	0.000145724	43	35		
z-p6	0.444233283	0.000211161	6279	5517		
z-p5	0.442770143	0.001674301	1479	441	91	350
z-p7	0.442770143	0.001674301	1164	441	91	350
z-gss	0.444497782	0.000053338	348	259		
z-p10	0.444453847	0.000009403	358	225		

Test Problem SEGF22D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations total	actual	reused
Tolerance		0.0005				
z-pl	0.444547977	0.000103533	200	147		
z-p6	0.444233474	0.000210970	10227	7929		
z-p5	0.443605645	0.000838799	3501	801	163	638
z-p7	0.443605645	0.000838799	2176	801	163	638
z-gss	0.444474979	0.000030535	644	455		
z-pl0	0.444453847	0.000009403	358	225		
Tolerance		0.0001				
z-pl	0.444547977	0.000103533	200	147		
z-p6						
z-p5	0.444067361	0.000377083	10635	1809	351	1458
z-p7	0.444067361	0.000377083	5260	1809	351	1458
z-gss	0.444449757	0.000005313	3506	1855		
z-pl0	0.444444891	0.000000447	1530	961		
Tolerance		0.00005				
z-pl	0.444547977	0.000103533	200	147		
z-p6						
z-p5	0.444215809	0.000228635	20649	2709	517	2192
z-p7	0.444215809	0.000228635	8102	2709	517	2192
z-gss	0.444447695	0.000003251	7646	3255		
z-pl0	0.444444891	0.000000447	1530	961		

Test Problem SEGF23D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-p1	0.305345904	0.009049608	14	10		
z-p6	0.277368927	0.018927369	21	17		
z-p5	0.277368927	0.018927369	52	17	9	8
z-p7	0.277368927	0.018927369	48	17	9	8
z-gss						
z-p10	0.296485886	0.000189590	676	343		
Tolerance		0.1				
z-p1	0.305345904	0.009049608	14	10		
z-p6	0.285706653	0.010589643	190	153		
z-p5	0.285706653	0.010589643	571	153	35	118
z-p7	0.285706653	0.010589643	430	153	35	118
z-gss						
z-p10	0.296485886	0.000189590	676	343		
Tolerance		0.05				
z-p1	0.305345904	0.009049608	14	10		
z-p6	0.285706653	0.010589643	190	153		
z-p5	0.285706653	0.010589643	571	153	35	118
z-p7	0.285706653	0.010589643	430	153	35	118
z-gss						
z-p10	0.296485886	0.000189590	676	343		

Test Problem SEGF23D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-p1	0.305345904	0.009049608	14	10		
z-p6	0.285706653	0.010589643	190	153		
z-p5	0.285706653	0.010589643	571	153	35	118
z-p7	0.285706653	0.010589643	430	153	35	118
z-gss						
z-p10	0.296485886	0.000189590	676	343		
Tolerance		0.005				
z-p1	0.298748347	0.002452051	242	170		
z-p6	0.294660372	0.001635924	12332	8857		
z-p5	0.291921377	0.004374919	7753	1105	175	930
z-p7	0.291921377	0.004374919	3421	1105	175	930
z-gss						
z-p10	0.296485886	0.000189590	676	343		
Tolerance		0.001				
z-p1	0.297602719	0.001330894	1721	1210		
z-p6						
z-p5	0.291926649	0.004369647	8770	1241	189	1052
z-p7	0.291926649	0.004369647	3829	1241	189	1052
z-gss						
z-p10	0.296305699	0.000009403	6616	3375		

Test Problem SEGF23D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.0005				
z-p1	0.297477407	0.001181111	2871	2010		
z-p6						
z-p5						
z-p7	0.294379170	0.001917126	19164	5321	759	4562
z-gss						
z-p10						

Test Problem SEGF32D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-p1	0.201342281	0.000012854	7	7		
z-p6	0.201388888	0.000033753	8	9		
z-p5	0.201388888	0.000033753	21	9	5	4
z-p7	0.201388888	0.000033753	21	9	5	4
z-gss	0.201342281	0.000012854	7	7		
z-p10						
Tolerance		0.1				
z-p1	0.201342281	0.000012854	7	7		
z-p6	0.201388888	0.000033753	8	9		
z-p5	0.201388888	0.000033753	21	9	5	4
z-p7	0.201388888	0.000033753	21	9	5	4
z-gss	0.201342281	0.000012854	7	7		
z-p10						
Tolerance		0.05				
z-p1	0.201342281	0.000012854	7	7		
z-p6	0.201388888	0.000033753	8	9		
z-p5	0.201388888	0.000033753	21	9	5	4
z-p7	0.201388888	0.000033753	21	9	5	4
z-gss	0.201342281	0.000012854	7	7		
z-p10						

Test Problem SEGF32D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-pl	0.201342281	0.000012854	7	7		
z-p6	0.201388888	0.000033753	8	9		
z-p5	0.201388888	0.000033753	21	9	5	4
z-p7	0.201388888	0.000033753	21	9	5	4
z-gss	0.201342281	0.000012854	7	7		
z-pl0						
Tolerance		0.005				
z-pl	0.201342281	0.000012854	7	7		
z-p6	0.201388888	0.000033753	8	9		
z-p5	0.201388888	0.000033753	21	9	5	4
z-p7	0.201388888	0.000033753	21	9	5	4
z-gss	0.201342281	0.000012854	7	7		
z-pl0						
Tolerance		0.001				
z-pl	0.201342281	0.000012854	7	7		
z-p6	0.201357323	0.000002188	43	45		
z-p5	0.201357323	0.000002188	113	45	13	32
z-p7	0.201357323	0.000002188	108	45	13	32
z-gss	0.201342281	0.000012854	7	7		
z-pl0						

Test Problem SEGF32D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.0005				
z-p1	0.201342281	0.000012854	7	7		
z-p6	0.201357323	0.000002188	43	45		
z-p5	0.201357323	0.000002188	113	45	13	32
z-p7	0.201357323	0.000002188	108	45	13	32
z-gss	0.201342281	0.000012854	7	7		
z-p10						
Tolerance		0.0001				
z-p1	0.201342281	0.000012854	7	7		
z-p6	0.201357323	0.000002188	43	45		
z-p5	0.201357323	0.000002188	113	45	13	32
z-p7	0.201357323	0.000002188	108	45	13	32
z-gss	0.201342281	0.000012854	7	7		
z-p10						
Tolerance		0.00005				
z-p1	0.201342281	0.000012854	7	7		
z-p6	0.201357323	0.000002188	43	45		
z-p5	0.201357323	0.000002188	113	45	13	32
z-p7	0.201357323	0.000002188	108	45	13	32
z-gss	0.201354289	0.000000846	37	35		
z-p10						

Test Problem SEGF32D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.00001				
z-pl	0.201342281	0.000012854	7	7		
z-p6	0.201355273	0.000000138	217	189		
z-p5	0.201355273	0.000000138	591	189	41	148
z-p7	0.201355273	0.000000138	489	189	41	148
z-gss	0.201354289	0.000000846	37	35		
z-pl0						
Tolerance		0.000005				
z-pl	0.201342281	0.000012854	7	7		
z-p6	0.201355273	0.000000138	217	189		
z-p5	0.201355273	0.000000138	591	189	41	148
z-p7	0.201355273	0.000000138	489	189	41	148
z-gss	0.201354784	0.000000351	101	91		
z-pl0						
Tolerance		0.000001				
z-pl	0.201342281	0.000012854	7	7		
z-p6	0.201355144	0.000000009	874	765		
z-p5	0.201355144	0.000000009	3200	765	145	620
z-p7	0.201355144	0.000000009	2109	765	145	620
z-pl0						
z-gss	0.201355081	0.000000054	167	147		

Test Problem SEGF33D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183414502	0.000060362	19	17		
z-p5	0.183414502	0.000060362	51	17	9	8
z-p7	0.183414502	0.000060362	47	17	9	8
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.1				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183414502	0.000060362	19	17		
z-p5	0.183414502	0.000060362	51	17	9	8
z-p7	0.183414502	0.000060362	47	17	9	8
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.05				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183414502	0.000060362	19	17		
z-p5	0.183414502	0.000060362	51	17	9	8
z-p7	0.183414502	0.000060362	47	17	9	8
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		

Test Problem SEGF33D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183414502	0.000060362	19	17		
z-p5	0.183414502	0.000060362	51	17	9	8
z-p7	0.183414502	0.000060362	47	17	9	8
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.005				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183414502	0.000060362	19	17		
z-p5	0.183414502	0.000060362	51	17	9	8
z-p7	0.183414502	0.000060362	47	17	9	8
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.001				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183357996	0.000003856	179	153		
z-p5	0.183357996	0.000003856	567	153	35	118
z-p7	0.183357996	0.000003856	426	153	35	118
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		

Test Problem SEGF33D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.0005				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183357996	0.000003856	179	153		
z-p5	0.183357996	0.000003856	567	153	35	118
z-p7	0.183357996	0.000003856	426	153	35	118
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.0001				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183357996	0.000003856	179	153		
z-p5	0.183357996	0.000003856	567	153	35	118
z-p7	0.183357996	0.000003856	426	153	35	118
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.00005				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183354383	0.000000243	1764	1241		
z-p5	0.183354383	0.000000243	8745	1241	189	1052
z-p7	0.183354383	0.000000243	3803	1241	189	1052
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		

Test Problem SEGF33D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.00001				
z-p1	0.183339494	0.000014646	12	10		
z-p6	0.183354383	0.000000243	1764	1241		
z-p5	0.183354383	0.000000243	8745	1241	189	1052
z-p7	0.183354383	0.000000243	3803	1241	189	1052
z-gss	0.183339494	0.000014646	12	10		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.000005				
z-p1	0.183353173	0.000000967	112	90		
z-p6	0.183354383	0.000000243	1764	1241		
z-p5	0.183354383	0.000000243	8745	1241	189	1052
z-p7	0.183354383	0.000000243	3803	1241	189	1052
z-gss	0.183353173	0.000000967	112	90		
z-p10	0.183354140	0.000000000	616	343		
Tolerance		0.000001				
z-p1	0.183353173	0.000000967	112	90		
z-p6						
z-p5						
z-p7	0.183354383	0.000000243	3803	1241	189	1052
z-p10	0.183353173	0.000000967	112	90		
z-gss	0.183354140	0.000000000	616	343		

Test Problem SEGF42D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations total	actual	reused
Tolerance		0.5				
z-p1	4.000000000	0.151291030	11	7		
z-p6	4.157862768	0.006571738	73	45		
z-p5	4.157862768	0.006571738	125	45	13	32
z-p7						
z-gss	4.000000000	0.151291030	11	7		
z-p10	4.151291620	0.000000590	102	49		
Tolerance		0.1				
z-p1	4.000000000	0.151291030	11	7		
z-p6	4.158404123	0.007113093	189	117		
z-p5	4.157862768	0.006571738	125	45	13	32
z-p7						
z-gss	4.000000000	0.151291030	11			
z-p10	4.151291620	0.000000590	102	49		
Tolerance		0.05				
z-p1	4.146517279	0.004773751	61	35		
z-p6	4.158404123	0.007113093	189	117		
z-p5	4.158404123	0.007113093	354	117	29	88
z-p7						
z-gss	4.146517279	0.004773751	61	35		
z-p10	4.151291620	0.000000590	102	49		

Test Problem SEGF42D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-pl	4.151050452	0.000240578	255	147		
z-p6	4.158404123	0.007113093	189	117		
z-p5	4.158404123	0.007113093	354	117	29	88
z-p7						
z-gss	4.146441756	0.004849274	161	91		
z-pl0	4.151291620	0.000000590	102	49		
Tolerance		0.005				
z-pl	4.151050452	0.000240578	255	147		
z-p6	4.158404123	0.007113093	189	117		
z-p5	4.158404123	0.007113093	354	117	29	88
z-p7						
z-gss	4.148746104	0.002544926	213	119		
z-pl0	4.151291620	0.000000590	102	49		
Tolerance		0.001				
z-pl	4.151229237	0.000061793	982	567		
z-p6	4.151312255	0.000021225	1358	765		
z-p5	4.158404123	0.007113093	354	117	29	88
z-p7						
z-gss	4.151104853	0.000186177	765	399		
z-pl0	4.151291620	0.000000590	102	49		

Test Problem SEGF42D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.0005				
z-pl	4.151277237	0.000013793	1042	595		
z-p6	4.151312255	0.000021225	1358	765		
z-p5	4.151493662	0.000202632	2910	693	137	556
z-p7						
z-gss	4.151194495	0.000096535	1001	511		
z-pl0	4.151291608	0.000000578	471	225		
Tolerance		0.0001				
z-pl	4.151286920	0.000004110	3793	2191		
z-p6	4.151292949	0.000001919	5235	2925		
z-p5	4.151316178	0.000025148	17919	2133	401	1732
z-p7						
z-gss	4.151281468	0.000009652	4003	1631		
z-pl0	4.151291608	0.000000578	471	225		
Tolerance		0.00005				
z-pl	4.151288243	0.000002787	4018	2247		
z-p6	4.151292886	0.000001856	5535	3069		
z-p5	4.151310246	0.000019216	22156	2421	449	1972
z-p7						
z-gss	4.151284707	0.000006323	5531	2079		
z-pl0	4.151291608	0.000000578	471	225		

Test Problem SEGF42D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.00001				
z-p1	4.151290343	0.000000687	15052	8687		
z-p6						
z-p5						
z-p7						
z-gss	4.151290029	0.000001001	53204	7007		
z-p10	4.151291608	0.000000578	471	225		
Tolerance		0.000005				
z-p1	4.151290453	0.000000577	21603	9051		
z-p6						
z-p5						
z-p7						
z-gss						
z-p10	4.151291608	0.000000578	471	225		

Test Problem SEGF43D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.5				
z-pl	8.052928426	0.028806547	21	10		
z-p6	8.093352987	0.011618014	310	153		
z-p5	8.093352987	0.011618014	645	153	35	118
z-p7						
z-gss	8.052928426	0.028806547	21	10		
z-p 10	8.081734977	0.000000004	922	343		
Tolerance		0.1				
z-pl	8.052928426	0.028806547	21	10		
z-p6	8.094772398	0.013037425	1430	697		
z-p5	8.093352987	0.011618014	645	153	35	118
z-p7						
z-gss	8.052928426	0.028806547	21	10		
z-p 10	8.081734977	0.000000004	922	343		
Tolerance		0.05				
z-pl	8.081168598	0.000566375	191	90		
z-p6	8.082329644	0.000594671	2625	1241		
z-p5	8.093352987	0.011618014	645	153	35	118
z-p7						
z-gss	8.081168598	0.000566375	193	90		
z-p 10	8.081734977	0.000000004	922	343		

Test Problem SEGF43D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.01				
z-pl	8.080826280	0.000908693	869	410		
z-p6	8.082100084	0.000365111	18274	7769		
z-p5	8.094772398	0.013037425	4611	697	133	564
z-p7						
z-gss	8.081168598	0.000566375	193	90		
z-p 10	8.081734977	0.000000004	922	343		
Tolerance		0.005				
z-pl	8.079790872	0.001944101	1225	570		
z-p6						
z-p5	8.082329644	0.000594671	9001	1241	189	1052
z-p7						
z-gss	8.08082628	0.000908693	907	410		
z-p 10	8.081734977	0.000000004	922	343		
Tolerance		0.001				
z-pl	8.081708613	0.000026360	1737	730		
z-p6						
z-p5						
z-p7						
z-gss	8.081710739	0.000024234	4702	1850		
z-p 10	8.081734977	0.000000004	922	343		

Test Problem SEGF43D

Program	Estimate	Actual Error	Milltime	Integrand Evaluations		
				total	actual	reused
Tolerance		0.0005				
z-p1	8.081678187	0.000056786	9359	4250		
z-p6						
z-p5						
z-p7						
z-gss	8.081706565	0.000028408	8287	3970		
z-p10						

Appendix 9

The following pages contain the results of the runs of the basic adaptive simplex program on the set of test problems described in appendix 1.3.

The milltime is given in millunits where 1 millunit is equal to 1000 micro units. These timings are used for comparison only.

Program	Estimate	Actual Error	Milltime	Integrand Evaluations
Tolerance		0.5		
segf12d	0.402368927	0.002368927	10	9
segf13d	0.145710672	0.002853529	17	14
segf22d	0.083333333	0.047618649	9	9
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.543068297	0.001587272	13	9
Tolerance		0.1		
segf12d	0.402368927	0.002368927	10	9
segf13d	0.145710672	0.002853529	17	14
segf22d	0.083333333	0.047618649	9	9
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.543068297	0.001587272	13	9
Tolerance		0.05		
segf12d	0.402368927	0.002368927	10	9
segf13d	0.145710672	0.002853529	17	14
segf22d	0.113835450	0.017116532	34	27
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.543068297	0.001587272	13	9

Problem	Estimate	Actual Error	Milltime	Integrand Evaluations
Tolerance		0.01		
segf12d	0.402368927	0.002368927	10	9
segf13d	0.145710672	0.002853529	17	14
segf22d	0.125627117	0.005324865	232	171
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.543068297	0.001587272	13	9
Tolerance		0.005		
segf12d	0.402368927	0.002368927	10	9
segf13d	0.145710672	0.002853529	17	14
segf22d	0.128884797	0.002067185	479	351
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.543068297	0.001587272	13	9
Tolerance		0.001		
segf12d	0.400431916	0.000431916	10	9
segf13d	0.143242691	0.000385548	847	546
segf22d	0.130794666	0.000157316	1815	1323
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.541135811	0.000345214	49	27

Problem	Estimate	Actual Error	Milltime	Integrand Evaluations
Tolerance		0.0005		
segf12d	0.400431916	0.000431916	10	9
segf13d	0.143052325	0.000195182	1109	714
segf22d	0.131059256	0.000107274	3646	2655
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.035138887	0.000009324	14	14
Tolerance		0.0001		
segf12d	0.400431916	0.000431916	135	99
segf13d	0.142892380	0.000035237	1897	1218
segf22d	0.130949391	0.000002591	13929	8361
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.541490925	0.000009900	459	243
Tolerance		0.00005		
segf12d	0.400379259	0.000379259	236	171
segf13d	0.142871008	0.000013865	4085	2618
segf22d	0.130930977	0.000021005	29313	15831
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035138887	0.000009324	14	14
segf42d	0.541490925	0.000009900	459	243

Problem	Estimate	Actual Error	Milltime	Integrand Evaluations
Tolerance		0.00001		
segf12d	0.400379259	0.000379259	236	171
segf13d	0.142861791	0.000004648	10065	6034
segf22d				
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035145980	0.000002231	444	322
segf42d	0.541491025	0.000010000	1484	783
Tolerance		0.000005		
segf12d				
segf13d	0.142859759	0.000002616	23527	13258
segf22d				
segf32d	0.107407407	0.000018389	8	9
segf33d	0.035145980	0.000002231	444	322
segf42d				
Tolerance		0.000001		
segf12d				
segf13d				
segf22d				
segf32d	0.107407407	0.000018389	8	9
segf42d				
segf33d	0.035147809	0.000000402	1068	770

References

[1] E.B.Anders

"An extension of omberg integration procedures to n variables"

J.A.C.M.,v13,1966,pp505-510.

[2] B.Bereanu

"On the convergence of cartesian multidimensional quadrature formulae."

Numer. Math. 19,pp348-350,1972

[3] B.A.Bowen & R.Buhr

"The logical design of multiple microprocessor systems"

pub. Prentice Hall.

[4] R.Burlirsch

"Bemerkungen zur Romberg integration"

Numer. Mathe. 6, pp6-16, 1964

[5] J.J.Cosgrove and P.Cannon

"Array processors"

IEEE SIG on signal processing

[6] G.R.Cowper

"Gaussian quadrature formulas for triangles"

I.J. for N.M. in Eng. 7, pp405-408.

[7] R.Cranley & T.N.L.Patterson

"On the automatic numerical evaluation of definite integrals"

Comp. J. (1971),vol 14,no. 2,pp 189-198

[8] P.J.Davis

"A construction of non negative approximate quadratures"

Math.Comp., V 21. pp 578-582

[9] P.J.Davis & P.Rabinowitz

"Numerical inte gration"

Blaisdell,Waltham,Mass.

[10] Dawson

"Software strategy for multiprocessors"

I.P.C. Business Press vol3,no6,1979.

[11] V.A.Dixon

"Numerical quadrature:A survey of available algorithms"

NPL. Report NAC 36,June 1973

[12] Elise de Donker

"New Euler-Maclaurin expansions and their application to quadrative over the s-dimensional simplex"

Math. Comp. , vol 33,number 147,July 1979,pp 1003-1018

[13] Charles S. Duris

"generating and compounding product type Newton-Cotes quadrature formulae"

ACM Transactions on Mathematical Software, Vol 2,No. 1,March 1976, pp 50-58

[14] Richard Franke

"Orthogonal polynomials and approximate multiple integration"

SIAM J. Numer. Anal. vol8 no4, 1971.

[15] F.N.Fritsch

"On self contained numerical integration formulae for symmetric regions"

SIAM J.Numer.Anal. vol8,no2,1971.

[16] A.C.Genz

"An adaptive multidimensional quadrature procedure"

Computer Physics Communications 4, pp11-15

[17] A.C.Genz

"A procedure for multidimensional quadrature"

NAG D01FAF

[18] S.Haber

"Numerical evaluation of multiple integrals"

SIAM Review vol.12,1970,pp481-526.

[19] P.C.Hammer & A.Stroud

"Numerical integration over simplexes"

MTAC vol10, pp137-139.

[20] P.C.Hammer O.Marlowe & A.Stroud

"Numerical integration over simplexes and cones"

MTAC vol 10, pp130-137.

[21] Per Brinch Hansen

"The programming language concurrent Pascal"

IEEE Trans. on Software Engineering, vol1,no2, 1975.

[22] P.Hillion

"Numerical integration on a triangle"

I.J. for Numer. Methods in Eng., vol11, pp 797-815, 1977

[23] D.R.Hunkins

"Product type multiple integration formulae"

BIT 13 pp408-414

[24] D.K.Kahaner

"Comparison of numerical quadrature formulae"

Mathematical software (1971) Academic Press, pp229-259.

[25] D.K.Kahaner & M.Wells

"An algorithm for n dimensional adaptive quadrature using advanced programming techniques"

LA-UR 76-2310 Los Alamos Scientific Laboratory report (1979)

[26] P.Keast

"Optimal parameters for multidimensional integration"

SIAM J. Numer. Anal. vol10,n05, 1973

[27] P.J.King

"Adaptive multidimensional quadrature procedures"

M.Sc. Project , Aston University 1978

[28] D.Kronrod

"Nodes and weights of quadrature formulae"

English translation from Russian, Consultants Bureau, New York, MR 32 £ 597

[29] G.Lague & R.Baldur

"Extended numerical integration methods for triangular surfaces"

Short Comms. 1976 , C.A.C.M.

[30] Lauffer "Interpolation mehrfacher Integrale" MR 16, 862.

[31] J.N. Lyness

"quadrature over a simplex: part1 .A representation for the integrand function"

SIAM J. Numer. Anal. vol 15, no1 1978

[32] J.N. Lyness

"Quadrature over a simplex part2. A representation for the error functional"

SIAM J. Numer. Anal. vol 15, no5 1978

[33] J.N. Lyness

"The effect of inadequate convergence criteria in automatic routines"

Comp. J. 12, 1969 pp279-281

[34] J.N. Lyness

"Guidelines for automatic quadrature routines"

Information Processing 71,1972

[35] J.N. Lyness

"Moderate degree symmetric quadrature rules for the triangle"

J.Inst. Math. Appl. 1975, 15 , pp 19-32

[36] J.N. Lyness

"An error functional expansion for n dimensional quadrature with an integrand function singular at a point"

Math. Comp. vol30, no 133 1976 pp 1-23

[37] J.N. Lyness

"Quid, Quo quadrature"

Argonne National Laboratory, Illinois

[38] J.N. Lyness & J.J.Kaganove

"A technique for comparing automatic quadrature routines"

Comp. J. vol. 20, no2, pp170-177 , 1977

[39] J.N. Lyness & J.J.Kaganove

"Comments on the nature of automatic quadrature routines"

A.C.M. Trans on Math. Software vol2, no1, 1976 , pp65-81

[40] J.N. Lyness & B.McHugh

"Integration over multidimensional hypercubes 1, a progressive procedure"

Comp. J. vol 6, 1963, pp264-270

[41] M.Malcolm & R.Bruce Simpson

"Local versus global strategies for adaptive quadrature"

A.C.M. Trans on Math. Software vol1, no 2 1975

[42] James Clerk Maxwell

"On approximate multiple integration between limits of summation"

Cambridge Phil. Soc. Proc., vol 3, 1877, pp39-47.

[43] W.McKeeman

"Adaptive integration and multiple integration"

C.A.C.M. vol6, no 8 , 1963

[44] W.D.Maurer

"An improved hash code for scatter storage"

C.A.C.M. vol11, no1, 1968

[45] R.Morris

"Scatter storage techniques"

C.A.C.M. vol 11, no1, 1968

[46] H.O'Hara & F.Smith

"The evaluation of definite integrals by interval subdivision"

Comp. J. 12 1969, pp179-182

[47] J.Oliver

"A doubly adaptive Clenshaw-Curtis quadrature method"

Comp. J. vol15, 1972, pp 141-147

[48] T.Patterson

"Algorithm for automatic numerical integration over a finite interval"

C.A.C.M. no11, vol 16, 1973, pp 694-699

[49] T.Patterson

"The optimum addition of points to quadrature formulae"

Math. Comp. 22, 1968, pp847-856

[50] R.Piessens & A.Haegemans

"Cubature formulae of degree nine for symmetric planar regions"

Math. Comp. vol29, no 131, 1975, pp 810-815

[51] J.Rice

"Mathematical Software"

Academic Press, New York, 1971

[52] I.Robinson

"Methods of numerical integration"

Ph.D. Thesis, Dept. of Inf. Science, Melbourne University, 1973

[53] T.Sasaki

"Multidimensional integration techniques for ill behaved functions" Colloquium on advanced computing methods in theoretical physics C.N.R.S., Marseille (1975).

[54] P.Silvester

"Symmetric quadrature formulae for simplexes"

Math. Comp. vol24, pp95-100

[55] W.Squire

"Numerical evaluation of a class of singular double integrals by symmetric pairing"

Int. J. Num. Meth. in Eng. vol10, pp703-720 , 1976

[56] T.Strom

"Practical error estimates for repeated Richardson extrapolation schemes"

BIT 13 , 1973, pp 196-205

[57] A.H.Stroud

"Approximate calculation of multiple integrals"

Prentice Hall series in automatic computation.

[58] A.H.Stroud

"Some fourth degree integration formulae for simplexes"

Math. Comp. vol30, num 134, 1976, pp 291-294

[59] K.Zabrzewska, J.Dudek and N.Nazarewicz

"A numerical calculation of multidimensional integrals"

Computer Physics Comms. 14, 1978, pp299-309