

An Adaptive Numerical Cubature Algorithm for Simplices

Alan Genz and Ronald Cools

Report TW 273, December 1997



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

An Adaptive Numerical Cubature Algorithm for Simplices

Alan Genz and Ronald Cools*

Report TW273, December 1997

Department of Computer Science, K.U.Leuven

Abstract

A globally adaptive algorithm for numerical multiple cubature of a vector of functions over a collection of n -dimensional simplices is described. The algorithm is based on a subdivision strategy that chooses for subdivision at each stage the subregion (of the input simplices) with the largest estimated error. This subregion is divided into two, three or four equal volume subregions by cutting selected edges. These edges are selected using information about the smoothness of the integrands in the edge directions. The algorithm allows a choice from several imbedded cubature rule sequences for approximate integration and error estimation. A Fortran 90 implementation as part of CUBPACK is also discussed. Testing of the algorithm is described.

Keywords : adaptive integration, cubature, multidimensional integration, simplex.
CR Subject Classification : G.1.4 [Numerical Analysis]: Quadrature and Numerical Differentiation – *adaptive quadrature; multiple quadrature*; G.4 [Mathematical Software]: *efficiency; reliability and robustness*

*Department of Mathematics, Washington State University, Pullman, WA 99164-3113 USA

An Adaptive Numerical Cubature Algorithm for Simplices

Alan Genz *

Department of Mathematics
Washington State University
Pullman, WA 99164-3113 USA
AlanGenz@wsu.edu

Ronald Cools

Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
Ronald.Cools@cs.kuleuven.ac.be

Abstract

A globally adaptive algorithm for numerical multiple cubature of a vector of functions over a collection of n -dimensional simplices is described. The algorithm is based on a subdivision strategy that chooses for subdivision at each stage the subregion (of the input simplices) with the largest estimated error. This subregion is divided into two, three or four equal volume subregions by cutting selected edges. These edges are selected using information about the smoothness of the integrands in the edge directions. The algorithm allows a choice from several imbedded cubature rule sequences for approximate integration and error estimation. A Fortran 90 implementation as part of CUBPACK is also discussed. Testing of the algorithm is described.

Keywords: adaptive integration, cubature, multidimensional integration, simplex

1 Introduction

The problem considered in this paper is the numerical evaluation of integrals in the form

$$I[\mathbf{f}] = \int_T \mathbf{f}(\mathbf{x}) dT,$$

where \mathbf{x} is an n -vector, \mathbf{f} is l -vector and T is a collection of m n -simplices. There has been only limited work done on practical algorithms for this general problem. Most of the research has considered the $n = 2$ case, where T is a triangle. For this case there has been work on the development of integration rules (reviewed in the recent paper by Lyness and Cools [23]), and algorithms ([3, 10, 18, 19, 22]). For the $n = 3$ case, where T is a tetrahedron, there has been more limited rule ([7, 28]) and algorithm [4] development work. For $n > 3$, the main rule development papers are by Silvester [27], Grundmann and Möller [17], Keast [21] and Lyness and Genz [24]. Other rules are described and referenced in the books by Stroud [28] and Engels [11] and the paper by Cools and Rabinowitz [7]. The only general algorithm widely available [26] is an automatic algorithm that uses Grundmann's and Möller's rules, although an experimental algorithm was developed by Kahaner and Wells [20]. A brief description of an earlier version of the algorithm described in this paper was given by Genz [15]. The purposes of this paper are to give a detailed description of that algorithm, along with several significant additions made since the original algorithm was described, and to describe the implementation and testing of the algorithm as part of CUBPACK [8].

In the following sections the details of the algorithm for the general problem will be described. The general adaptive algorithm is a globally adaptive algorithm that has been used extensively for adaptive integration, so only a brief description of the general algorithm will be given here.

*Supported by NATO Collaborative Research Grant CRG 940139, and Flemish FWO and US NSF grants.

The globally adaptive algorithm uses successive refinements or subdivisions of T , where each subdivision is used to provide a better approximation to $I[\mathbf{f}]$. These subdivisions are designed to dynamically concentrate the computational work in the subregions of T where the integrand $\mathbf{f}(\mathbf{x})$ is most irregular, and thus adapt to the behavior of the integrand. The general structure of the globally adaptive algorithm consists of a sequence of stages. Each stage has the following five main steps:

1. Select a subregion with largest estimated error from the current set of subregions.
2. Divide the selected subregion.
3. Apply a local cubature rule to any new subregions.
4. Update the subregion set.
5. Update the global integral and error estimates, and check for termination.

The initial subregion set for the algorithm is the set of the original collection of simplices T . The required input for such an algorithm is T , the integrand $\mathbf{f}(\mathbf{x})$, a limit on the number of \mathbf{f} values allowed, and a requested error tolerance ϵ . The algorithm terminates when the estimated global error is less than ϵ or further subdivision would require too many \mathbf{f} values.

Algorithms of this type have been extensively used for numerical cubature over hyper-rectangular regions (see for example [1]), and also for triangles (e.g. [3]) and tetrahedra [4]. Because procedures for the steps 1, 4 and 5 are the same for all regions, these have been implemented as part of the basic CUBPACK framework [6] and will not be discussed here. Details can be found in [8]. The implementation that we have developed allows the user to make use of specially designed high degree rules if $n = 2$ or $n = 3$. These rules and their error estimates are essentially the same as those used in the TOMS algorithms 706 [3] and 720 [4], and their inclusion in CUBPACK has already been described elsewhere [8], so we do not include any detailed discussion of these special rules here. The rest of this paper will focus on the choice of the local cubature rules, error estimation, subdivision strategy, testing and interface with the CUBPACK framework for the general simplex algorithm.

2 The Local Cubature Rules and Error Estimators

In this section we restrict our discussion to integrands with a single component $f(\mathbf{x})$. For a vector integrand $\mathbf{f}(\mathbf{x})$, the cubature rules discussed in this section can be applied to each of the components in \mathbf{f} . The local cubature rules that we use for the simplex algorithm take the form

$$B[f] = \sum_{i=1}^N w_i f(\mathbf{x}_i) \simeq \int_R f(\mathbf{x}) d\mathbf{x},$$

for a chosen simplex subregion R . With these rules, the points \mathbf{x}_i and weights w_i are chosen to make the rule exact for all polynomials of degree d or less, for some fixed d and n . The rules that are used in practical calculations typically have degrees in the range 5-13, with N a polynomial of degree n in d . The points and weights are found for some standard n -simplex, and then the linear structure of the rule allows an affine transformation, preserving the polynomial degree, to any other finite n -simplex. The use of polynomial integrating rules ensures rapid convergence if the subdivision is fine enough so that the integrand is reasonably smooth locally.

The local rules used by the simplex algorithm are Grundmann and Möller [17] rules. Let G_s denote a degree $2s + 1$ Grundmann and Möller rule. These rules are defined by

$$G_s[f] = \sum_{i=0}^s 2^{-2s} (-1)^i \frac{(2s+1+n-2i)^{2s+1}}{(2s+1+n-i)! i!} \sum_{\substack{\sum_{j=0}^n \beta_j = s-i \\ \beta_0 \geq \dots \geq \beta_n}} \sum f \left(\left(\frac{2\beta_0+1}{2s+1+n-2i}, \dots, \frac{2\beta_n+1}{2s+1+n-2i} \right) \right).$$

The inner sum notation $\sum f(\mathbf{y})$ denotes a sum over all unique permutations of the $(n+1)$ -vector \mathbf{y} . Each f value in the sum uses only the last n components of \mathbf{y} . The rules as given are for an integral over the standard unit n -simplex defined by $\sum_{i=1}^n x_i \leq 1$ for all $x_i \geq 0$. A rule G_s requires $\binom{n+s+1}{s}$ f values, and includes imbedded rules of degrees $2s-1, 2s-3, \dots, 1$.

The type of local error estimators used by our simplex cubature algorithm are ‘‘Norwegian’’ error estimators (see Berntsen and Espelid [2]). These error estimators require a sequence of null rules, which are combined to produce the final error estimate. Our simplex algorithm uses two sets of null rules. One set of the null rules used in the simplex algorithm has null rules that are constructed from the difference between a degree $2s+1$ Grundmann and Möller rule and a lower degree Grundmann and Möller rule. Define a degree $2i+1$ null rule N_i by $N_i[f] = G_s[f] - G_i[f]$, for $0 \leq i < s$. The second set of null rules $\hat{N}_i[f]$ is defined by $\hat{N}_i[f] = G_s[f] - L_i[f]$, where L_i is some degree $2i+1$ rule, different from $G_i[f]$, for $0 \leq i < s$. Our implementation allows $0 < s \leq 4$ and the L rules implemented are: a) ($i=3$) a degree 7 Mysovskikh [25] rule that requires $3(n+1)(n+2)/2$ additional points, b) ($i=2$) a degree 5 Stroud rule [29] that requires $(n+1)(n+2)$ additional points, and c) and d) special ($i=1$) degree 3 and ($i=0$) degree 1 rules that use subsets of the degree 5 Stroud rule points.

The degree 7 Mysovskikh rule is described in a paper in Russian that is not easily accessible, so we give some details here of our implementation of this rule. The Mysovskikh rule structure is similar to the Grundmann and Möller degree 7 rule. If we let $\{\mathbf{y}_i\}$ be the sequence of (\mathbf{y}) 's for the $f(\mathbf{y})$ values for the Grundmann and Möller rule, and $\{\mathbf{y}'_i\}$ be the corresponding sequence for the Mysovskikh rule, and the sequences $\{W_i\}$ and $\{W'_i\}$ be the corresponding sequences for the weights, the two rules are compared in Table 1.

Table 1: Grundmann and Möller and Mysovskikh Generators and Weights

i	\mathbf{y}_i	W_i	\mathbf{y}'_i	W'_i
1	$(\frac{1}{n+1}, \frac{1}{n+1}, \dots, \frac{1}{n+1})$	$-\frac{(n+1)^7}{2^6(n+4)!3!}$	$= \mathbf{y}_1$	W'_1
2	$(\frac{3}{n+3}, \frac{1}{n+3}, \dots, \frac{1}{n+3})$	$\frac{(n+3)^7}{2^6(n+5)!2!}$	$(1 - n\alpha_1, \alpha_1, \dots, \alpha_1)$	W'_2
3	$(\frac{5}{n+5}, \frac{1}{n+5}, \dots, \frac{1}{n+5})$	$-\frac{(n+5)^7}{2^6(n+6)!}$	$(1 - n\alpha_2, \alpha_2, \dots, \alpha_2)$	W'_3
4	$(\frac{3}{n+5}, \frac{3}{n+5}, \frac{1}{n+5}, \dots, \frac{1}{n+5})$	$-\frac{(n+5)^7}{2^6(n+6)!}$	$= \mathbf{y}_4$	$= W_4$
5	$(\frac{7}{n+7}, \frac{1}{n+7}, \dots, \frac{1}{n+7})$	$\frac{(n+7)^7}{2^6(n+7)!}$	$(1 - n\alpha_3, \alpha_3, \dots, \alpha_3)$	W'_5
6	$(\frac{5}{n+7}, \frac{3}{n+7}, \frac{1}{n+7}, \dots, \frac{1}{n+7})$	$\frac{(n+7)^7}{2^6(n+7)!}$	$(\frac{4}{n+7}, \frac{4}{n+7}, \frac{1}{n+7}, \dots, \frac{1}{n+7})$	$\frac{10(n+7)^7}{3^6(n+7)!}$
7	$(\frac{3}{n+7}, \frac{3}{n+7}, \frac{3}{n+7}, \frac{1}{n+7}, \dots, \frac{1}{n+7})$	$\frac{(n+7)^7}{2^6(n+7)!}$	$= \mathbf{y}_7$	$= W_7$
8	-	-	$(\frac{11}{2(n+7)}, \frac{5}{2(n+7)}, \frac{1}{n+7}, \dots, \frac{1}{n+7})$	$\frac{2^6(n+7)^7}{3^8(n+7)!}$

Using $a_i = (n+1)\alpha_i - 1$ for $i = 1, 2, 3$, Mysovskikh shows that the a_i 's are the zeros of the polynomial

$$\begin{aligned}
p(z) = & -144(142528 + n(23073 - 115n)) \\
& -12(6690556 + n(2641189 + n(245378 - 1495n)))z \\
& -16(6503401 + n(4020794 + n(787281 + n(47323 - 385n))))z^2 \\
& - (n+7)(6386660 + n(4411997 + n(951821 + n(61659 - 665n))))z^3.
\end{aligned}$$

If the a_i 's can be found, the weights W'_2 , W'_3 and W'_5 are given by

$$\begin{aligned}
W'_2 &= \frac{U_7 - (a_2 + a_3)U_6 + a_2a_3U_5}{a_1^5(a_1^2 - (a_2 + a_3)a_1 + a_2a_3)}, \\
W'_3 &= \frac{U_7 - (a_1 + a_3)U_6 + a_1a_3U_5}{a_2^5(a_2^2 - (a_1 + a_3)a_2 + a_1a_3)}, \\
W'_5 &= \frac{U_7 - (a_2 + a_1)U_6 + a_2a_1U_5}{a_3^5(a_3^2 - (a_2 + a_1)a_3 + a_2a_1)},
\end{aligned}$$

where

$$U_5 = -\frac{6^3(52212 - n(6353 + n(1934 - 27n)))}{23328(n+6)!},$$

$$U_6 = \frac{6^4(7884 - n(1541 - 9n))}{23328(n+6)!},$$

$$U_7 = -\frac{6^5(8292 - n(1139 - 3n))}{23328(n+7)!}.$$

Mysovskikh's paper contains a table of α_i 's and W'_2, W'_3 and W'_5 for $4 \leq n \leq 20$, accurate to eight decimal digits. The values for α_1 in this table are approximately $\frac{1}{n+3}$ (to four digits). In our implementation, a_1 is determined to machine accuracy by applying the Newton iteration to $p(z)$ with $\frac{n+1}{n+3} - 1 \approx (n+1)\alpha_1 - 1 = a_1$ as a starting value. Then $p(z)$ is deflated, and a_2 and a_3 are determined using the quadratic formula. After finding accurate values for the α_i 's and W'_2, W'_3 and W'_5 , W'_1 is computed using

$$W'_1 = \frac{1}{n!} - (n+1)(W'_2 + W'_3 + W'_5 + n(W'_4 + W'_6 + 2W'_8 + (n-1)W'_7/3)/2).$$

The weights for the degree 5 Stroud rule are not given explicitly in Stroud's paper, but we found some moderately simple expressions for these weights that we also report here. The generators for this rule are $(\frac{1}{n+1}, \frac{1}{n+1}, \dots, \frac{1}{n+1})$, and $(1-nr_i, r_i, \dots, r_i)$, for $i = 1, 2$, and $((1-(n-1)u_i)/2, (1-(n-1)u_i)/2, u_i, \dots, u_i)$, for $i = 1, 2$, with respective weights S_1, S_2, S_3, S_4 and S_5 , where $r_1 = \frac{n+4-\sqrt{15}}{n^2+8n+1}$, $r_2 = \frac{n+4+\sqrt{15}}{n^2+8n+1}$, $u_1 = \frac{n+7+2\sqrt{15}}{n^2+14n-11}$, and $u_2 = \frac{n+7-2\sqrt{15}}{n^2+14n-11}$. If we define $\lambda_i = 1 - (n+1)r_i$ and $\delta_i = (1 - (n+1)u_i)/2$ for $i = 1, 2$, then

$$S_2 = \frac{2(27-n) - \lambda_2(13-n)(n+5)}{\lambda_1^4(\lambda_1 - \lambda_2)(n+5)!}, \quad S_3 = \frac{2(27-n) - \lambda_1(13-n)(n+5)}{\lambda_2^4(\lambda_2 - \lambda_1)(n+5)!},$$

$$S_4 = \frac{2 - \delta_2(n+5)}{\delta_1^4(\delta_1 - \delta_2)(n+5)!}, \quad S_5 = \frac{2 - \delta_1(n+5)}{\delta_2^4(\delta_2 - \delta_1)(n+5)!}, \quad S_1 = \frac{1}{n!} - (n+1)(S_2 + S_3 + n(S_4 + S_5)/2).$$

The special degree 3 rule uses the first three of the degree 5 Stroud rule generators, with weights

$$S'_2 = \frac{2 - \lambda_2(n+3)}{\lambda_1^2(\lambda_1 - \lambda_2)(n+3)!}, \quad S'_3 = \frac{2 - \lambda_1(n+3)}{\lambda_2^2(\lambda_2 - \lambda_1)(n+3)!}, \quad S'_1 = \frac{1}{n!} - (n+1)(S_2 + S_3).$$

Finally, the special degree 1 rule uses only the second of the degree 5 Stroud rule generators, with weight $S''_2 = \frac{1}{(n+1)!}$.

The correct implementation of these rules was tested by checking for the exact (to machine precision) integration of monomials of the appropriate degrees. This test was satisfied for all of the rules that we implemented. We were also able to accurately reproduce Mysovskikh's table.

The actual error estimators require the use of scaled orthogonal null rules, so we define the sequence $\{e_i\}_{i=1}^{2s}$ to be the set obtained by orthogonalizing and scaling (using Berntsen's and Espelid's procedure) the sequence $\hat{N}_{s-1}, N_{s-1}, \dots, \hat{N}_0, N_0$. We also define related quantities E_i and reduction factors r_i required by the Norwegian algorithm, by $E_i = \sqrt{e_{2i}^2 + e_{2i-1}^2}$, for $i = 1, 2, \dots, s$ and $r_i = E_i/E_{i+1}$, for $i = 1, 2, \dots, s-1$. Finally, we define $r = \max(r_i)$, $\bar{E} = \max(E_i)$ and our final error estimate $E[f]$ by

$$E[f] = \begin{cases} C_e(C_t\bar{E} + (1-C_t)E_1) & \text{if } r \geq 1 \\ rC_eE_s & \text{if } r < 1 \end{cases}$$

where $C_e = s(3C_t + (44 + s(7s - 32))(1 - C_t)/24)$, and $0 \leq C_t \leq 1$. C_t is a user selected tuning parameter. For $C_t = 1$, a very conservative error estimate is produced, and this is the most reliable case. For $C_t = 0$, a liberal error estimate is produced.

When the integrand is a vector, the error estimate $E[f]$ is applied to each component of \mathbf{f} , and the result is an l -vector of error estimates for the estimated integral of \mathbf{f} over the selected subregion. In order to make decisions about possible further subdivisions, the globally adaptive algorithm requires a scalar measure of the error for that subregion. For our simplex algorithm, we use $\|E[\mathbf{f}]\|_\infty$ as an overall measure of the error for a selected subregion. This quantity is used for subsequent subdivision decisions by the region processor part of the globally adaptive algorithm. When the algorithm terminates, the error estimate returned for each component of \mathbf{f} is the sum, taken over all of the subregions in the final subregion list, of the estimated subregion errors for that component.

In the Fortran 90 implementation of the algorithm the user is allowed to choose $s = 1, 2, 3$ or 4 (degree $d = 3, 5, 7$ or 9). The costs, in terms of \mathbf{f} values, for computing the rules and error estimates are given in Table 2, for $1 < n \leq 10$.

Table 2: \mathbf{f} Values Needed for Local Rules and Error Estimators

$d \backslash n$	2	3	4	5	6	7	8	9	10
3	7	9	11	13	15	17	19	21	23
5	16	23	31	40	50	61	73	86	100
7	32	49	86	126	176	237	310	396	496
9	65	114	201	315	470	675	940	1276	1695

The degree 3 and 5 rules are not recommended for most problems. It is the opinion of the authors that these low degree rules with simpler error estimates should only be considered for high dimensional problems. Two special rules are also provided for $n = 2$ and $n = 3$. For $n = 2$, a 37 point degree 13 rule [3] is available, and for $n = 3$, a 43 point degree 8 rule [4] is available.

3 The Subdivision Method

At each stage in the globally adaptive algorithm a selected subregion needs to be subdivided. A simple natural subdivision method is to cut at the midpoint of each edge, but this produces 2^n pieces, and this is too many when n is greater than 3 or 4. Another problem with this subdivision is that it is not as adaptive as the method we use because this subdivision is done without any analysis of the integrand, even though the error for the integral over a selected subregion is often due to irregularity of the integrand in only a small number of directions. Our subdivision strategy, which uses a division of the largest error subregion into at most four new pieces, and which takes account of differences in integrand behavior in different directions, allows the algorithm to proceed from one stage to the next in a controlled manner.

The subdivision procedure that we use, is a modified version of a procedure first described in [15] and further developed (for $n = 2$ only) in [12]. Once a subregion has been selected for subdivision, the globally adaptive algorithm used by CUBPACK will recommend a subdivision into at most 2, 3 or 4 pieces, depending on the current progress of the integration. Our subdivision procedure then divides the subregion by cutting one, two or three edges of the selected subregion to produce a 2-division, 3-division or 4-division of the selected subregion. An n -simplex has $n(n+1)/2$ edge directions, and our algorithm chooses subdivision directions from these directions.

In order for the algorithm to be efficient, a method is needed for selecting good edges for subdivision, and therefore some measure of integrand irregularity is needed. A popular measure of integrand irregularity that has been successfully used with adaptive algorithms for hyper-rectangles is a fourth difference of the integrand. We follow this approach, using modifications of the methods described in [15] and [12]. Our simplex algorithm uses fourth differences centered at the centroid of the selected simplex. Let $\mathbf{v}_{k,0}, \mathbf{v}_{k,1}, \dots, \mathbf{v}_{k,n}$ be the vertices of the current largest error simplex subregion T_k , and let the edge directions be given by $\mathbf{d}_{i,j} = \mathbf{v}_{k,j} - \mathbf{v}_{k,i}$, for $0 \leq i < j \leq n$. Now define $f_{i,j}(\alpha) = f(\mathbf{c} + \alpha \mathbf{d}_{i,j} / (5(n+1)))$, with $\mathbf{c} = \sum_{i=0}^n \mathbf{v}_{k,i} / (n+1)$ (the centroid of T_k). Then a fourth difference operator for the $(i,j)^{th}$ direction is given by

$$D_{i,j}(f) = \|\mathbf{d}_{i,j}\|_1 |6f_{i,j}(0) - 4(f_{i,j}(2) + f_{i,j}(-2)) + (f_{i,j}(4) + f_{i,j}(-4))|.$$

The scaling factor $\|\mathbf{d}_{i,j}\|_1$ is used to provide some bias for division of very long edges. All of the points where $f(\mathbf{x})$ is computed for these differences lie within T_k . Our general algorithm is designed to allow for vector integrands \mathbf{f} . When \mathbf{f} has more than one component, we define $D_{i,j} = \|D_{i,j}(\mathbf{f})\|_1$.

The edges for subdivision are edges (i,j) where $D_{i,j}$ is large. Let $D_{i_s,j_s} = \max_{i<j}(D_{i,j})$. If a 2-division has been recommended, then let $\mathbf{v}_c = (\mathbf{v}_{k,i_s} + \mathbf{v}_{k,j_s})/2$. Two new subregions are produced from T_k that have the same vertices as T_k except that the i_s and j_s vertices are respectively replaced by \mathbf{v}_c . For a 3 or 4-subdivision let $D_{i_t,j_t} = \max_{i<j,(i,j)\neq(i_s,j_s)}(D_{i,j})$. If a 4-division has been recommended, and $D_{i_t,j_t} > D_{i_s,j_s}/2$ (the two edges with largest D values have similar D values), then our algorithm first divides T_k into two pieces using the algorithm for a 2-division and then halves each of the two new pieces by bisection of the (i_t, j_t) edge for each piece. If either a 3-division has been recommended, or a 4-division is recommended but $D_{i_t,j_t} < D_{i_s,j_s}/2$, then our algorithm considers two possible 3-divisions. Let vertex index l_s be defined by $D_{i_s,l_s} + D_{l_s,j_s} = \max_{l,l\neq i_s,l\neq j_s}(D_{i_s,l} + D_{l,j_s})$. The vertices indexed by i_s, j_s and l_s define a triangle. Now order these vertices, by exchanging i_s and j_s if necessary, so that $D_{i_s,j_s} \geq D_{j_s,l_s} \geq D_{i_s,l_s}$. If $D_{i_s,j_s}/8 \geq D_{j_s,l_s}$ (the largest D value edge has D significantly larger than the other D 's), then the edge (i_s, j_s) is trisected, producing three new equal volume subregions of T_k . Otherwise, the edge (i_s, j_s) is cut at the point $\mathbf{v}_c = (2\mathbf{v}_{k,i_s} + \mathbf{v}_{k,j_s})/3$, and two subregions of T_k are produced. The subregion that has the original edge (j_s, l_s) is then divided into two pieces by cutting that edge at the midpoint. The final result is three new equal volume subregions of T_k . The following procedure summarizes our subdivision algorithm.

Procedure for Choice of Subdivision.

- Determine $D_{i_s,j_s} = \max_{i<j}(D_{i,j})$.
- **If** 2-division is recommended **then**
 - bisect edge (i_s, j_s) of T_k to produce T_k^1 and T_k^2 .
- else**
 1. Determine $D_{i_t,j_t} = \max_{i<j,(i,j)\neq(i_s,j_s)}(D_{i,j})$.
 2. **If** 4-division is recommended **and** $D_{i_t,j_t} > D_{i_s,j_s}/2$ **then**
 - i) bisect edge (i_s, j_s) of T_k to produce T_k^1 and T_k^2 ;
 - ii) bisect edge (i_t, j_t) of both T_k^1 and T_k^2 to produce four new subregions $T_k^{11}, T_k^{12}, T_k^{21}$ and T_k^{22} .
 - else**
 - (a) Determine $D_{i_s,l_s} + D_{l_s,j_s} = \max_{l,l\neq i_s,l\neq j_s}(D_{i_s,l} + D_{l,j_s})$.
 - (b) Reorder i_s, j_s , if necessary, so that $D_{i_s,j_s} \geq D_{j_s,l_s} \geq D_{i_s,l_s}$.
 - (c) **If** $D_{i_s,j_s}/8 \geq D_{j_s,l_s}$ **then**
 - trisect edge (i_s, j_s) of T_k to produce new subregions T_k^1, T_k^2 and T_k^3 .
 - else**
 - i) cut edge (i_s, j_s) of T_k at $(2\mathbf{v}_{k,i_s} + \mathbf{v}_{k,j_s})/3$ to produce new subregions T_k^1 and T_k^2 ;
 - ii) bisect edge (j_s, l_s) of T_k^2 replacing T_k^2 with new subregions T_k^{21} and T_k^{22} .
 - end if.**
- end if.**
- end if.**

The heuristic factors 1/2 and 1/8 used at steps 2 and 2c in our procedure were selected after extensive testing (see next section) of the algorithm. The cost of determining the subdivision of T_k is $2n(n+1) + 1\mathbf{f}$ values. The actual cost per new subregion is at most approximately $n(n+1)\mathbf{f}$ values, and this is small compared to the cost of computing the local rules when the local rule has degree at least seven.

4 Implementation in CUBPACK and Testing

The algorithm described in the previous sections has been implemented in Fortran 90 as a part of CUBPACK. A standard call to use the algorithm is a call to the main CUBPACK integration routine CUBATR in the form:

```
USE CUI
USE PRECISION
...
CALL CUBATR( n, f, l, V, RGTYPE, m, RESULT, ERROR, FUNVLS,
            IFAIL, EPSABS, EPSREL, RESTART, MINPTS, MAXPTS, KEY, JOB, TUNE
)
)
```

CUBATR computes approximations in the vector RESULT to the vector integral

$$I[\mathbf{f}] = \int_T \mathbf{f} \, dx_1 dx_2 \cdots dx_n$$

where \mathbf{f} is an l -vector of integrands and T is collection of m n -dimensional simplices with vertices given in a 3-dimensional array V. CUBATR attempts to compute RESULT with

$| I[f_k] - \text{RESULT}(k) | \leq \max(\text{EPSABS}, \text{EPSREL} | I[f_k] |)$, for $k = 1, \dots, l$, where EPSABS and EPSREL are absolute and relative tolerances. A complete specification for all of the CUBATR parameters is given in Appendix 1.

The module CUI is a CUBPACK user interface module that provides details of the CUBATR parameter declarations. The precision level of the real variables used in the calculations is determined by the Fortran 90 module PRECISION, which the user of CUBPACK may modify. We have followed the recommendations for precision level maintenance described by Buckley [5] (sections 6.2 and 6.3) in the design of this module and its use in CUBPACK. From the users point of view, a key feature of this interface is that the only additions to the user code for precision specifications are the line “USE PRECISION” at the beginning of each subprogram or module, and the use of “REAL(STND)” for declarations of all reals related to the use of CUBPACK.

The main integration subroutine and subprograms are organized according to the structure given in Figure 1. We discuss only CUBPACK subroutines that are used for simplex integration. In the following list we briefly describe the purpose of CUBATR and supporting subroutines.

- CUBATR is a driver subroutine for the algorithm controller Global_Adapt. Inside CUBATR the input is checked and the workspace is allocated.
- Check checks the validity of the input to CUBATR.
- Rule_Cost uses n and KEY to determine how many integrand values are needed for the local integration rule.
- Handle_Error determines output values for IFAIL and if necessary, prints error messages.
- Global_Adapt is the algorithm controller which at each subdivision step decides whether to stop or continue.
- Process_Region is the subregion processing subroutine. Other subroutines are called to maintain the subregion data structure and Divide is called to subdivide a chosen subregion.
- Divide subdivides a chosen simplex subregion.
- Rule_General uses n and KEY to select a local integration rule.
- Rule_T2 computes approximations to integrals over triangles, as in [3].

- Rule_T3 computes approximations to integrals over tetrahedra, as in [4].
- Rule_Tn computes approximations to the integrals and the errors over each subregion. When the first call to Rule_Tn with a new n or KEY value occurs, Rule_Tn uses RuleParms_Tn to compute the parameters for the rules, and orthogonalize the null rules.
- RuleParms_Tn computes the generators and weights for all of the rules.
- SymSmp_Sum computes the symmetric sums needed for the rules. The integrand values are computed using the user supplied integrand \mathbf{f} .

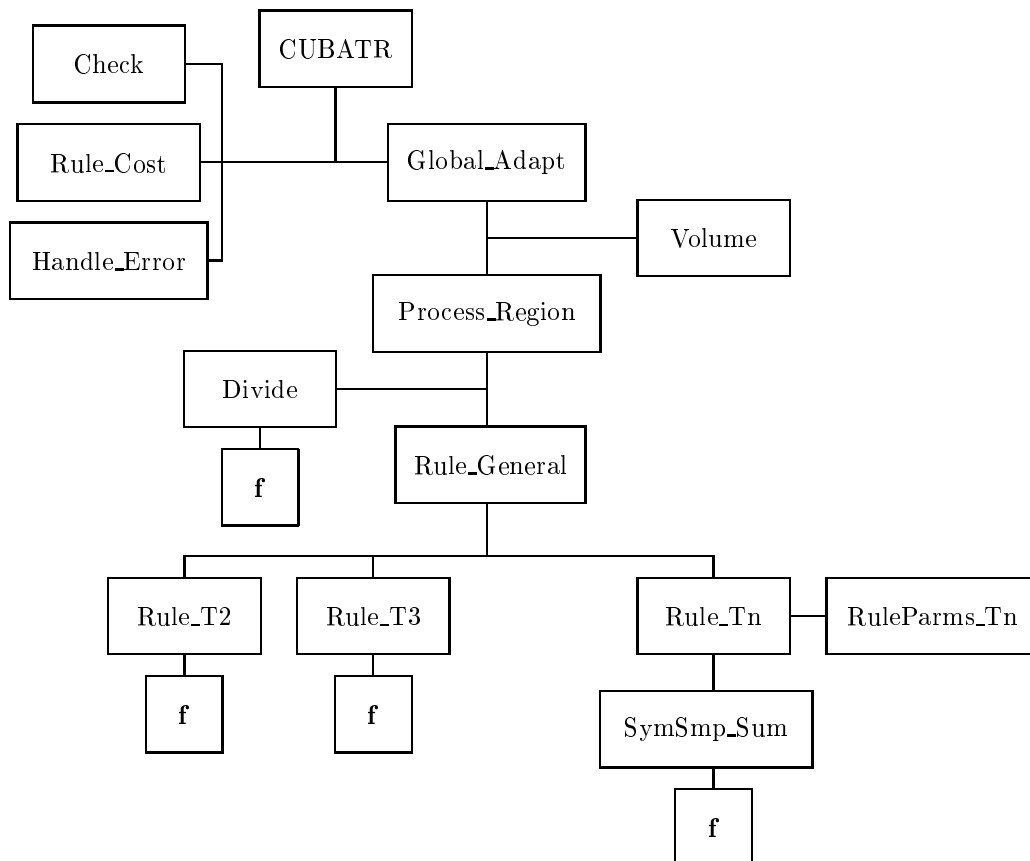


Figure 1. Subprogram organization

The simplex part of CUBPACK was tested with a combination of techniques. The primary testing used a modified version of the package developed by Genz [14]. This package was designed for testing cubature algorithms for hyper-rectangles. We use two applications of the package. In the first application, we used the same cubature problems generated by the package but divided the n -dimensional hyper-rectangular cubature region into $n!$ equal volume simplices in a standard way [24] and applied CUBATR to the resulting collection of pieces. We found that testing in this manner was feasible only for $n < 7$. In the second application of the package, we used the same cubature problems generated by the package but we transformed the standard unit simplex to the hyper-rectangle for each problem, using a constant Jacobian transformation described by Fang and Wang [13]. If the hyper-rectangle H for a particular test problem has dimensions $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n]$, then the transformation from a point $\mathbf{x} \in T$ to a point $\mathbf{y} \in H$

is defined by $y_i = a_i + (b_i - a_i)((1 - \sum_{j=i}^n x_j)/(1 - \sum_{j=i+1}^n x_j))^i$, for $i = 1, 2, \dots, n$, with the Jacobian given by $n! \prod_{i=1}^n (b_i - a_i)$. The test families that we used are given in the following table.

Table 3: Test Families for Hyper-Cubes

Test Family	Attribute
$f_1(x, y) = \cos(2\pi\beta_1 + \sum_{i=1}^n \alpha_i y_i)$	Oscillatory
$f_2(x, y) = \prod_{i=1}^n (\alpha_i^{-2} + (y_i - \beta_i)^2)^{-1}$	Internal Peak
$f_3(x, y) = (1 + \sum_{i=1}^n \alpha_i y_i)^{-(n+1)}$	Corner Peak
$f_4(x, y) = \exp(-\sum_{i=1}^n \alpha_i^2 (y_i - \beta_i)^2)$	Gaussian
$f_5(x, y) = \exp(-\sum_{i=1}^n \alpha_i y_i - \beta_i)$	C_0 Function

The integration region for each family is the unit n -cube $[0, 1]^n$. To determine a particular test problem, the β_i parameters are chosen uniformly random from $[0, 1]$. Then, an additional set of parameters α'_i are also chosen uniformly random from $[0, 1]$. These parameters are then all scaled by a constant c with $\alpha_i = c\alpha'_i$, where c is determined by the condition $n^{e_j} \sum_{i=1}^n \alpha_i = d_j$, with e_j and d_j fixed for each test family j . The tests that we carried out used $(e_1, e_2, e_3, e_4, e_5) = (1.5, 2, 2, 1, 2)$ and $(d_1, d_2, d_3, d_4, d_5) = (100, 500, 100, 100, 200)$. A series of tests were run using both applications of the package and results from the test package were used to determine what we believe are good values for error the estimation parameter C_e as a function of the tuning parameter C_t . We also used the tests to tune the subdivision procedure.

Table 4: CUBATR Test Results for $n = 7$, KEY = 3 and TUNE = 0

f Type	Estimated Digits	Actual Digits	Reliability	Wrong Digits
Oscillatory	(3.3, 3.6)	(3.2, 3.7)	0.48	(0.0,0.2)
Product peak	(2.5, 2.6)	(2.7, 3.0)	0.88	(0.0,0.0)
Corner peak	(3.1, 3.2)	(3.1, 3.3)	0.60	(0.0,0.0)
Gaussian	(2.2, 2.3)	(2.8, 3.2)	1.00	(0.0,0.0)
C_0 function	(1.7, 1.9)	(2.1, 2.4)	0.90	(0.0,0.0)

Tables 4 and 5 contain results from two sample test runs (after tuning) with the second application of the test package, with each run using 50 randomly chosen integrands from each of the five integrand types, for $n = 7$ and KEY = 3 (degree 7). For each sample, CUBATR was called, with a maximum of 343000 **f** values allowed, and a requested relative accuracy set at 10^{-10} to force the maximum number of **f** values to be used. The entries in the tables of the form (a,b) are 97% confidence intervals for the respective sample medians. In these tables, the number of digits is defined to be $-\log_{10}$ of the relative error (estimated or actual). Wrong digits are defined to the difference between estimated and actual digits, for those cases where the estimated number of correct digits is greater than the actual number of correct digits. The numbers in the reliability column give the fraction of the times when the number of estimated correct digits were less than the number of actual correct digits.

Table 5: CUBATR Test Results for $n = 7$, KEY = 3 and TUNE = 1

f Type	Estimated Digits	Actual Digits	Reliability	Wrong Digits
Oscillatory	(2.5, 2.7)	(3.2, 3.7)	1.00	(0.0,0.0)
Product peak	(1.6, 1.7)	(2.7, 2.9)	1.00	(0.0,0.0)
Corner peak	(2.3, 2.4)	(3.1, 3.3)	1.00	(0.0,0.0)
Gaussian	(1.2, 1.4)	(2.6, 3.1)	1.00	(0.0,0.0)
C_0 function	(0.7, 1.0)	(2.0, 2.4)	1.00	(0.0,0.0)

The results in Tables 4 and 5 are typical of results from the tests that we carried out for $1 < n \leq 8$. For high levels of reliability the user must be prepared to accept a more conservative error estimate. We also did extensive testing of the different rules with polynomials for all key values for $1 < n \leq 10$ to verify that we had a correct implementation of the rules. Test programs have been successfully run on IBM, SUN and DEC workstations.

References

- [1] J. Berntsen, T.O. Espelid, and A. Genz. An adaptive algorithm for the approximate calculation of multiple integrals. *ACM Trans. Math. Softw.*, 17(4). pp. 437-451, 1991.
- [2] J. Berntsen and T.O. Espelid. Error estimation in automatic quadrature routines. *ACM Trans. Math. Softw.*, 17(2), pp. 233-252, 1991.
- [3] J. Berntsen and T.O. Espelid. Algorithm 706: DCUTRI: An algorithm for adaptive cubature over a collection of triangles. *ACM Trans. Math. Softw.*, 19, pp. 329-342, 1993.
- [4] J. Berntsen, R. Cools and T.O. Espelid. Algorithm 720: An algorithm for adaptive cubature over a collection of 3-dimensional simplices. *ACM Trans. Math. Softw.*, 19, pp. 320-332, 1993.
- [5] A.G. Buckley. Conversion to Fortran 90: a case study. *ACM Trans. Math. Softw.*, 20(3), pp. 308-353, 1994.
- [6] R. Cools and A. Haegemans. CUBPACK: Progress report. in *Numerical Integration*, T.O. Espelid and A. Genz (Eds.), Kluwer Academic Publishers, pp. 305-315, 1992.
- [7] R. Cools and P. Rabinowitz. Monomial cubature rules since Stroud: a compilation. *J. Comput. Appl. Math.* 48, pp. 309-326, 1993.
- [8] R. Cools and A. Haegemans. CUBPACK: the framework. In preparation.
- [9] P.J. Davis and P. Rabinowitz. *Methods of Numerical Integration*, Academic Press, New York, 1984.
- [10] E. de Doncker and I. Robinson. An algorithm for automatic integration over a triangle using nonlinear extrapolation. *ACM Trans. Math. Softw.*, 10, pp. 1-16, 1984.
- [11] H. Engels. *Numerical Quadrature and Cubature*, Academic Press, New York, 1980.
- [12] T.O. Espelid and A. Genz. On the Subdivision Strategy for Adaptive Cubature Algorithms for Triangular Regions. University of Bergen Department of Informatics Technical Report No. 74, 1992.
- [13] K.-T. Fang and Y. Wang, Y. *Number Theoretic Methods in Statistics*, Chapman and Hall, London, 1994.
- [14] A. Genz. A package for testing multiple integration subroutines, in *Numerical Integration*, P. Keast and G. Fairweather (Eds.), D. Riedel, pp. 337-340, 1987.
- [15] A. Genz. An adaptive numerical integration algorithm for simplices. in *Computing in the 90s, Proceedings of the First Great Lakes Computer Science Conference*, N. A. Sherwani, E. de Doncker and J. A. Kapenga (Eds.), Lecture Notes in Computer Science Volume 507, Springer-Verlag, New York, pp. 279-292, 1991.
- [16] A.C. Genz and A.A. Malik. An adaptive algorithm for numerical integration over an n-dimensional rectangular region. *J. Comp. Appl. Math.*, 6, pp. 295-302, 1980.
- [17] A. Grundmann and H.M. Möller. Invariant integration formulas for the n-simplex by combinatorial methods. *SIAM J. Numer. Anal.* 15, pp. 282-290, 1978.
- [18] A. Haegemans. An algorithm for automatic integration over a triangle. *Computing*, 19, pp. 179-187, 1977.
- [19] D.K. Kahaner and O.W. Rechar. TWODQD: An adaptive routine for two-dimensional integration. *J. Comp. Appl. Math.* 17, pp. 215-234, 1987.

- [20] D.K. Kahaner and M.B. Wells. An experimental algorithm for N-dimensional adaptive quadrature. *ACM Trans. Math. Soft.* 5, pp. 86-96, 1979.
- [21] P. Keast. Cubature formulas for the sphere and simplex. *J. Inst. Maths. Applics.* 23, pp. 251-264, 1979.
- [22] D.P. Laurie. Algorithm 584: CUBTRI: Automatic cubature over a triangle. *ACM TOMS* 8, pp. 210-218, 1982.
- [23] J.N. Lyness and R. Cools. A survey of numerical cubature over triangles. *Proceedings of Symposia in Applied Mathematics* 48, pp. 127-150, 1994.
- [24] J.N. Lyness and A. Genz. On simplex trapezoidal rule families. *SIAM J. Numer. Anal.*, 17(1), pp. 126-147, 1980.
- [25] I.P. Mysovskikh. On a cubature formula for the simplex (Russian), *Vopros. Vycisl. i Prikl. Mat., Tashkent* 51, pp. 74-90, 1978.
- [26] D01PAF: An automatic integration subroutine for integration over an N-simplex, Numerical Algorithms Group Limited, Wilkinson House, Jordan Hill Road, Oxford, United Kingdom OX2 8DR.
- [27] P. Silvester. Symmetric quadrature formulas for simplices, *Math. Comp.*, 24, pp. 95-100, 1970.
- [28] A.H. Stroud. *Approximate Calculation of Multiple Integrals*, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.
- [29] A.H. Stroud. A Fifth Degree Integration Formula for the n-Simplex, *SIAM J Numer. Anal.* 6, pp. 90-98, 1969.

APPENDIX 1: The Simplex Parameters for CUBATR.

We describe the current version, which might change in the near future.

Input Parameters

n Integer number of variables in the integrand(s). We require $n > 1$.

f An externally declared real vector valued function for computing all components in the vector function for the given evaluation point. An interface of the following form must be given by the user in the subprogram which calls CUBATR:

```
INTERFACE
  FUNCTION F( L , Z )
    USE PRECISION
    INTEGER L
    REAL (STND) Z(:)
    REAL (STND) F(L)
  END FUNCTION F
END INTERFACE
```

l Integer number of components in the vector integrand function.

V A real array of dimension $(n, 0 : n, m)$. $V(i, j, k)$ must contain the i^{th} component of the j^{th} vertex of the k^{th} input simplex region, for $i = 1, \dots, n$, $j = 0, \dots, n$, $k = 1, \dots, m$.

RGTYPE An integer vector of length m .

For integration over simplices *all components of RGTYPE must be set to 1*.

m Integer number of subregions.

IFAIL Integer input/output information parameter.

This is an optional parameter.

If IFAIL is not set, then CUBPACK failures are soft noisy failures with error messages sent to standard output.

If IFAIL = -1, then CUBPACK failures are soft noisy failures with error messages sent to standard output, and the returned value for IFAIL specifies the type of failure.

If IFAIL = 0, then CUBPACK failures are hard noisy failures with error messages sent to standard output.

If IFAIL = 1, then CUBPACK failures are soft silent failures with no error messages sent to standard output, but the returned value for IFAIL specifies the type of failure.

EPSABS Real requested absolute error.

This is an optional parameter with default value 0.

EPSREL Real requested relative error.

This is an optional parameter with default value set to be the square root of the precision level epsilon, determined by a call to the Fortran 90 intrinsic function EPSILON.

RESTART A logical flag to signal continuation calls of CUBATR.

This is an optional parameter with default value .FALSE..

If RESTART = .FALSE., this is the first attempt to compute the integral(s).

If RESTART = .TRUE. , then a previous attempt is continued. In this case the only parameters for CUBATR that may be changed (with respect to the previous call of CUBATR) are IFAIL, MINPTS, MAXPTS, EPSABS, EPSREL and RESTART.

MINPTS The integer minimum number of **f** values the code has to use.

This is an optional parameter with default value 0.

MAXPTS The integer maximum number of **f** values the code is allowed to use. This is an optional parameter with default value set at $500 \times$ the number of **f** values needed for one local cubature rule application.

KEY An integer parameter that selects the integration rule. KEY must satisfy $0 \leq \text{KEY} \leq 4$. This is an optional parameter with default value 0. The following table gives the polynomial degrees of the different integration rules for the possible values of KEY and for $n = 2$, $n = 3$ and $n > 3$.

KEY \ n	2	3	> 3
0	13	8	7
1	3	3	3
2	5	5	5
3	7	7	7
4	9	9	9

JOB An integer memory allocation parameter. This is an optional parameter with default value 0.

For $\text{JOB} = 0$, the global adaptive integration is done as requested.

For $\text{JOB} = 1$, no integration is done and all previously allocated memory is freed.

TUNE A real error estimate tuning parameter. TUNE must satisfy $0 \leq \text{TUNE} \leq 1$.

This is an optional parameter with default value 1.

For $\text{TUNE} = 0$, a liberal error estimate is used.

For $\text{TUNE} = 1$, a conservative error estimate is used; this is the most reliable case.

Output Parameters

RESULT A real vector of length l containing approximations to all components of the vector integral.

ABSERR A real vector of length l of estimates for absolute errors for all components of the vector integral.

FUNVLS The number of **f** values used by CUBATR. In the case of a continuation call of CUBATR (with input $\text{RESTART} = \text{.TRUE.}$) FUNVLS is the number of additional **f** used by CUBATR for that call of CUBATR.

IFAIL Integer output error information parameter.

IFAIL = 0 for normal exit, with

$\text{ABSERR}(j) < \max(\text{EPSABS}, \text{EPSREL}|\text{RESULT}(j)|)$, for $j = 1, \dots, l$,

using MAXPTS or fewer **f** values.

IFAIL = 1 if CUBATR was called with input $|\text{IFAIL}| = 1$, and MAXPTS is too small for CUBATR to obtain the requested error. In this case CUBATR returns values of RESULT with estimated absolute errors ABSERR.

IFAIL = $k > 1$, if CUBATR was called with input $|\text{IFAIL}| = 1$, and an error occurred in the setting of one or more of the input parameters. Textual information about the meaning of a particular value of k is sent to standard output if CUBATR is called with IFAIL = -1 or IFAIL = 0. Detailed information about different values of k and their associated meanings is given in [8].

APPENDIX 2: An Example.

In this appendix we describe a simple 5-dimensional example. The integrands of interest are 5 expectations, $E\{x_i\}$, $i = 1, 2, \dots, 5$, with

$$E\{x_i\} = \frac{\int_0^1 \int_0^{1-x_1} \dots \int_0^{1-x_1-\dots-x_4} x_i \exp^{-\sum_{j=1}^5 (jx_j)^2} dx_5 dx_4 dx_3 dx_2 dx_1}{\int_0^1 \int_0^{1-x_1} \dots \int_0^{1-x_1-\dots-x_4} \exp^{-\sum_{j=1}^5 (jx_j)^2} dx_5 dx_4 dx_3 dx_2 dx_1}.$$

Here is a Fortran 90 program for this example.

```

PROGRAM Simplex_Example

USE Precision
USE CUI                               ! Cubpack User Interface

INTERFACE
  FUNCTION F( L, X )                  ! the integrand
    USE Precision
    INTEGER L
    REAL(stnd) X(:)
    REAL(stnd) F(L)
  END FUNCTION F
END INTERFACE

INTEGER, PARAMETER :: n = 5, & ! the dimension
                   m = 3    ! the number of simplex regions

INTEGER :: RGtype(m) = 1, NEval, i
REAL(stnd):: Vertices(1:n,0:n,1:m) = 0, Result(0:n), AbsErr(0:n)

DO i = 1, n
  Vertices(i,i,1:m) = 1
END DO
Vertices(1,0,1) = 0.5_stnd
Vertices(1,1,2) = 0.5_stnd

CALL CUBATR( n, F, n+1, Vertices(1:n,0:n,1:2), RGtype, 2,           &
            Result, AbsErr, NEval )

Print '( ''Expected values are '' / 5F12.8 )', Result(1:n)/Result(0)
Print '( ''with estimated errors < '' / 5F12.8 )', AbsErr(1:n)/Result(0)
Print *, 'The number of integrand evaluations used was', NEval

CALL CUBATR( n, F, n+1, Vertices(1:n,0:n,3), RGtype(1), 1,       &
            Result, AbsErr, NEval )

Print '( ''Expected values are '' / 5F12.8 )', Result(1:n)/Result(0)
Print '( ''with estimated errors < '' / 5F12.8 )', AbsErr(1:n)/Result(0)
Print *, 'The number of integrand evaluations used was', NEval

END

FUNCTION F( L, X )
USE Precision

```

```

INTEGER L, I
INTEGER, PARAMETER :: N = 5
REAL(stnd):: F(0:N), X(:), S
  S = 0
  DO I = 1, N
    S = S + ( I*X(I) )**2
  END DO
  F(0) = EXP(-S)
  DO I = 1, N
    F(I) = X(I)*F(0)
  END DO
END

```

In this test, CUBATR is called twice to illustrate the use of CUBATR with more than one subregion. On the first call, the original input simplex is split into two halves and the output gives the sum of results for both halves. For the second call, only the original simplex is used. Default values were used for all of the optional parameters. The following output results were produced.

```

-> Allowed number of function evaluations reached.
Expected values are
  0.22419489  0.17814155  0.14013797  0.11373497  0.09524075
with estimated errors <
  0.00044685  0.00021591  0.00030762  0.00021125  0.00026086
The number of integrand evaluations used was      62956
-> Allowed number of function evaluations reached.
Expected values are
  0.22419247  0.17813798  0.14013535  0.11373032  0.09523686
with estimated errors <
  0.00012068  0.00009732  0.00008745  0.00006266  0.00006824
The number of integrand evaluations used was      62757

```