TITLE: AN ORGANIZATION OF THE EXTRAPOLATION METHOD OF MULTI-DIMENSIONAL QUADRATURE FOR VECTOR PROCESSING

AUTHOR: R.B. Simpson
A. YAZICI

NAME:	DATE TAKEN	DATE RETURNED	SIGNATURE
Virginia Simpson	4 June 79	4 June 79	

To Edith

From Virginia

Date 9 Aug 79.

memo

University of Waterloo

Revised Report.

AN ORGANIZATION OF THE EXTRAPOLATION METHOD OF MULTI-DIMENSIONAL QUADRATURE FOR VECTOR PROCESSING

by

R.B. Simpson and A. Yazici

Research Report CS-78-37

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

August 1978

ABSTRACT

A form of algorithm for the extrapolation method of quadrature for triangulated domains of the plane is presented as being suitable to vector processing computer architectures. Tests of its performance on a CDC STAR-100 are discussed.

PREFACE

This report was substantially revised and submitted for consideration for publication. In the revisions, the notation for the extrapolation table of §2 was changed, and §4.3 was extended and made into a separate §5 on conclusions and speculations. Also, some of the detailed data reported in §4 was reduced in the revision, which references this report for further details, and for the source listing contained in the appendix.

§1 Introduction

Numerical methods for estimating multi-dimensional integrals typically involve formulae of the form

$$(1.1) \qquad \int\int\ldots\int_{D_N} f dV \cong \sum_{i=1}^{M} w_i f(P_i)$$

where D_N is a specific N-dimensional region, and w_i and P_i are the weights and nodes of an N-dimensional quadrature formula. The major factor in the time needed to carry out the computation can be expected to be the time required to evaluate f at the nodes P_i. It is well known that this effect can grow dramatically with the dimension N, since M typically increases exponentially with N, and the complexity of f can be expected to increase with N as well.

A traditional objective of the design of multi-dimensional quadrature formulae has been to reduce the number of function evaluations needed to gain an acceptable estimate of the integral through producing rules of as high order and low M as the flexibility in the choice of w_i and P_i, and the stability of the computation will allow. Recently, an interesting study was made of the design and implementation of algorithms for reducing M by adaptively distributing the nodes of (1.1) by D.K. Kahaner and M.B. Wells [2]. In this study, we take the viewpoint of investigating the possible reduction in time requirements for (1.1) by reducing the time required to evaluate f through vector processing. It is a question, then, of how to organize a numerical method into an algorithm to take advantage of the potential offered by a vector processor (see [8] for a discussion of the influence of computer architecture on algorithm organization).

A considerably more specific computation than (1.1) is studied, i.e. the extrapolation method for triangulated domains of the plane. The basic rule used for the extrapolation is the simple generalization of the trapezoidal rule to triangles. An extensive discussion of extrapolation on simplices for general classes of quadrature rules has been given by J. Lyness in [4]. Our focus here is on organizing this simple case of extrapolation in a manner suitable to vector processing. The ideas have been tested on a CDC STAR-100 as reported below, however the algorithm specifically avoids machine dependent features other than vector processing capability. A comprehensive description of this capability can be found in the survey article on pipeline processors in general by C.V. Ramamoorthy and H.F. Li, [6]. While the choices of algorithm and geometry are simple, the ideas are believed to be of more general interest because, on the one hand they are related to the important particular case of computing local stiffness matrices for the finite element method, and on the other hand, no special properties of the plane were used, so direct extensions to higher dimensions seem plausible.

In the following section a discussion of extrapolation as a numerical method, and a conventional algorithm for its implementation are given. In §3, an alternative, buffered, algorithm is discussed, and in §4, results of testing these algorithms on a CDC STAR-100 (as an example of a vector processor,) and on a HW 66/60 (as an example of a scalar processor,) are presented. Briefly summarized, these tests indicate that for multiple triangle integrations on the STAR, the buffered algorithm organization reduces the execution time per triangle by factors of down to 1/20 of the time taken by the conventional algorithm. In contrast, both organizations of the algorithm run in effectively the same time on the HW 66/60.

§2 Extrapolation for a Triangular Domain

The phrase 'the m-fold bisection of a triangle' will be used to describe the subdivision of a triangle determined by introducing 2^m-1 equally spaced nodes on each side and then joining these nodes to the corresponding ones on other edges by parallel lines as in Figure 2.1.

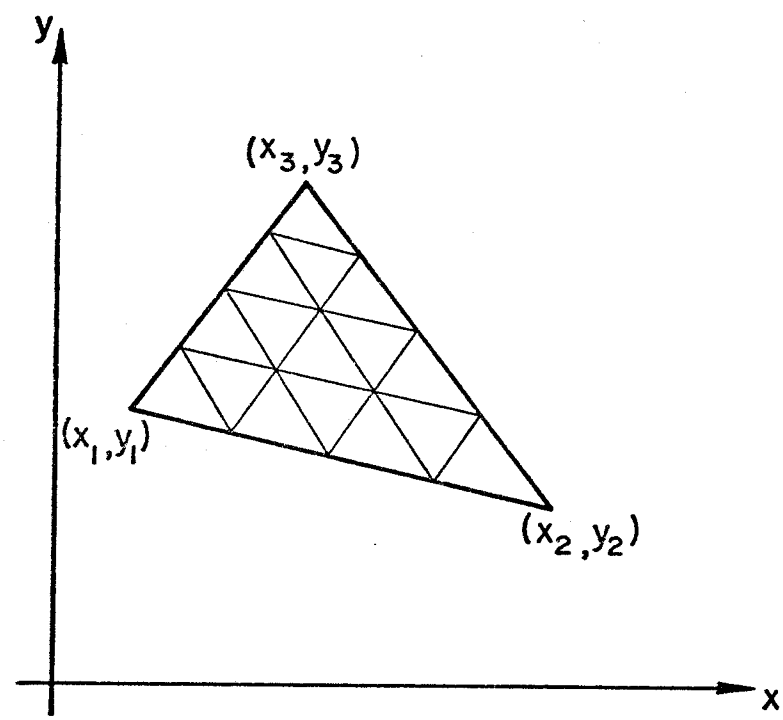


Figure 2.1

The vertices of the 4^m similar subtriangles introduced by this subdivision form the nodes of the m-fold copy (or composite form) of the trapezoidal quadrature rule over the triangle. Let D_2 denote the triangle, and for the m-fold bisection of D_2, let -

(2.1) $V \equiv$ the set of 3 corner vertices of D_2

$E_m \equiv$ the set of nodes interior to the edges of D_2

$I_m \equiv$ the set of nodes interior to D_2

Using A to denote the area of D_2, the m-fold copy of the trapezoidal rule can be written

$$(2.2)\qquad T_m^{(0)} = A\Big(\sum_{P\in V} f(P) + 3\sum_{P\in E_m} f(P) + 6\sum_{P\in I_m} f(P)\Big)/(3.4^m).$$

However, all the nodes of the $(m-1)$-fold bisection of D_2 are included in those of the m-fold bisection; so if the sequence $T_0^{(0)}, T_1^{(0)}, T_2^{(0)}, \ldots$ is to be computed, it can be done without duplicating function evaluations conveniently by the relation

$$(2.3)\qquad T_m^{(0)} = T_{m-1}^{(0)}/4 + A\Big(\sum_{P\in E_m - E_{m-1}} f(P) + 2\sum_{P\in I_m - I_{m-1}} f(P)\Big)/4^m$$

If $f(P)$ is 2k times continuously differentiable, it follows as a simple case of [4], that

$$(2.4)\qquad T_m^{(0)} = \iint_{D_2} f dA + c_1/2^{2m} + c_2/2^{4m} + \ldots$$

$$+ c_k/2^{2km} + O(2^{-(2k+2)m})$$

This error behaviour justifies the construction of the well known extrapolation table

$$(2.5)\qquad \begin{array}{lllll} T_0^{(0)} & & & & \\ T_1^{(0)} & T_0^{(1)} & & & \\ T_2^{(0)} & T_1^{(1)} & T_0^{(2)} & & \\ \vdots & \vdots & \vdots & & \\ T_k^{(0)} & T_{k-1}^{(1)} & T_{k-2}^{(2)} & \cdots & T_0^{(k)} \end{array}$$

by

$$(2.6) \qquad T_m^{(k)} = T_{m+1}^{(k-1)} + (T_{m+1}^{(k-1)} - T_m^{(k-1)}) / (4^k - 1)$$

and the entries of the k^{th} column, $T_m^{(k)}$ are an approximation of polynomial order $2k+1$ to $\iint_{D_2} f dA$. The bulk of the computational effort goes into evaluating the integrand to obtain the entries of the first column of (2.5) and we shall concentrate on this first column in the sequel. Specifically, the following table indicates the rate at which new function values are required as the level of extrapolation (= the length of the first column of (2.5)) increases.

Extrapolation level m	new corner nodes	new edge nodes	new interior nodes	attainable order 2m+1
0	3	-	-	1
1	-	3	-	3
2	-	6	3	5
3	-	12	18	7
4	-	24	84	9
5	-	48	360	11
6	-	96	1488	13
7	-	192	6048	15
8	-	384	24384	17
Total	3	765	32385	-

Table 2.1 Increase of quadrature nodes with extrapolation level.

A 'conventional' implementation of (2.2) and (2.6) could be patterned after the following algorithm organization for K levels of extrapolation.

(2.7) A CONVENTIONAL EXTRAPOLATION SCHEME

1. INITIALIZE $T_m^{(0)}$ FROM NODES OF V

2. FOR $m = 1$ TO K (m = LEVEL)

    (SUM OVER NEW EDGE NODES)

        FOR NODES OF $E_m - E_{m-1}$

            GENERATE NODE P

            EVALUATE $f(P)$

            UPDATE $T_m^{(0)}$

    (SUM OVER NEW INTERIOR NODES)

        FOR NODES OF $I_m - I_{m-1}$

            GENERATE NODE P

            EVALUATE $f(P)$

            UPDATE $T_m^{(0)}$

3. EXTRAPOLATE $T_0^{(0)}, T_1^{(0)}, T_2^{(0)}, \ldots, T_k^{(0)}$

The statement GENERATE NODE P indicates simply the computation of the (x-y) coordinates of P, conveniently done, e.g. from the area coordinate description of the m-fold bisection (See [9] page 116 for a description of area coordinates). The statements FOR THE NODES OF $E_m - E_{m-1}$ and FOR THE NODES OF $I_m - I_{m-1}$ denote loops which would be indexed by a parametrization of the points of $E_m - E_{m-1}$ and $I_m - I_{m-1}$ respectively. For example, expressed in FORTRAN, the second of these scans would appear as a nested pair of DO loops with the range of the inner DO loop parameter depending on the outer

DO loop parameter in order to cover the interior of the triangle (See §3.1 for more details). Since the scan occurs inside the loop structure for the successive levels of extrapolation, the core of the algorithm is a triple nested DO loop.

While this algorithm organization is reasonable for scalar processors, it does not lend itself naturally to vector processing because of the fairly complex loop structure and the presence of different 'macro' operations in the core of the loops. The complexity of the loops is due to the need to generate nodal coordinates, but the time consuming operation of the body of the loops is the evaluation of the integrand. This observation suggests that an alternative algorithm organization better suited to vector processing ought to 'desynchronize' node generation from integrand evaluation.

§3 A Buffered Algorithm Organization

In this alternative approach to an extrapolation algorithm, node generation, integrand evaluation, and updating of the partial sums for the first column of the Romberg table (2.5), are considered to be three separate subprocesses. The node generation process generates a vector, or buffer, of nodes which it passes to the function evaluation process to turn into a vector of function values. This latter process then passes the vector of function values to the updating process which updates the relevant components of $T_i^{(0)}$ $i = 1$ to K.

The interprocess communication uses buffers whose length is a parameter that is independent of the extrapolation algorithm and may be varied to suit the computer being used. In particular, one of the objectives of the experimental computations is to examine the effect of buffer length on performance. A single buffer may contain information (nodal coordinates, or function values) for several levels of extrapolation at lower levels, or only part of a level at the higher levels of extrapolation. The contents of each buffer are described in a buffer head containing pointers to level interfaces within the buffer, if any.

This organization has apparent implications for multi-processor architectures, but here we are concerned only with vector processors. In this case, the relationship of subprocesses is indicated in Figure 3.1.

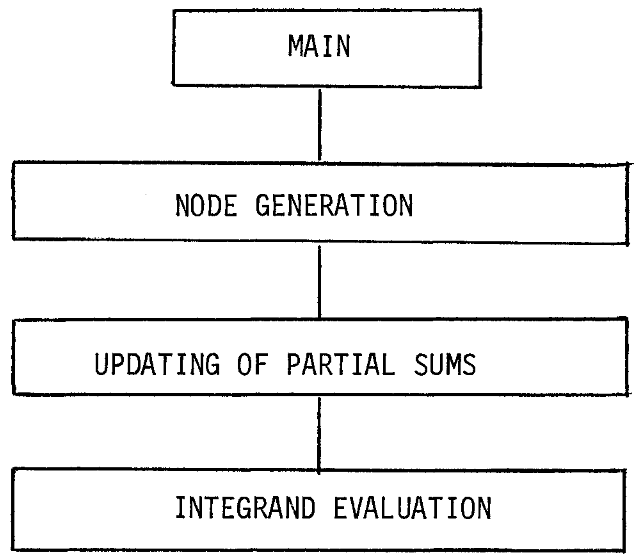


Figure 3.1 Relation of subprocesses of alternative algorithm organization

The algorithm operates basically under the control of the node generation process. The potential for vector operations in evaluating the integrand is obvious, and is the primary motivation. However, as indicated below, significant benefits can also be derived from updating the quadrature partial sums using vectorized addition, and using vector operations within the node generation process. A more detailed description of the node generation process is given in the following subsection, which ends with an algorithmic description of the alternative organization.

§3.1 Node Generation

The looping structure of the conventional organization (2.7) is retained in the node generation process i.e.

(3.1) <u>NODE GENERATION</u> (FOR K LEVELS OF EXTRAPOLATION)

(GENERATE NEW EDGE NODES)

1. FOR $m = 1$ TO K (m = LEVEL)

    FOR NODES OF $E_m - E_{m-1}$

        GENERATE NODE P

(GENERATE NEW INTERIOR NODES)

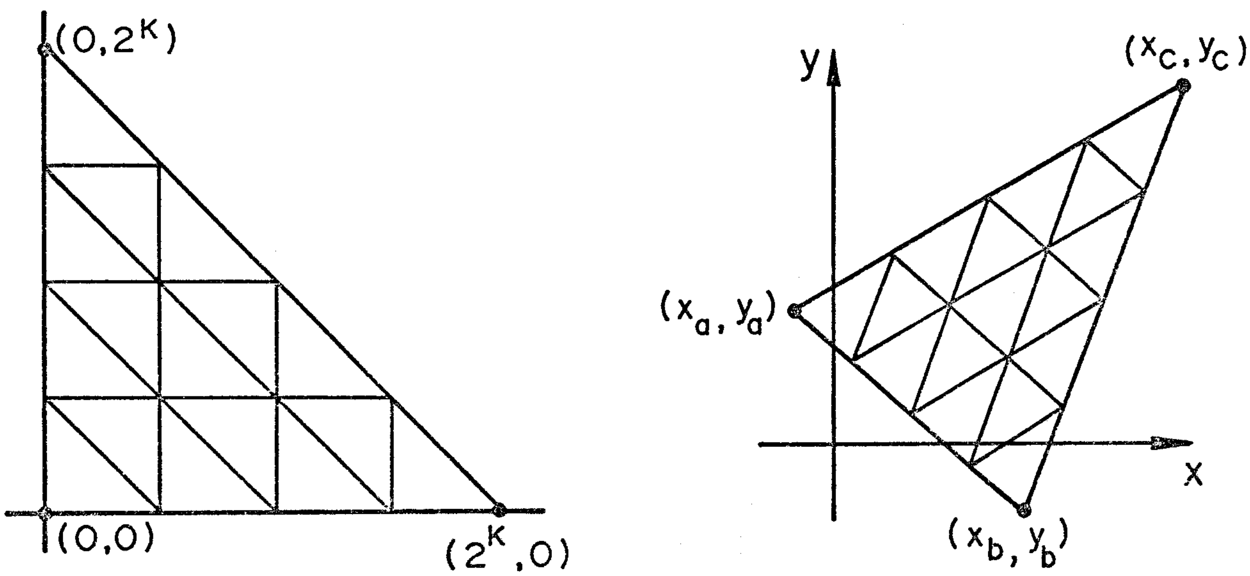
2. FOR $m = 1$ TO K

    FOR NODES OF $I_m - I_{m-1}$

        GENERATE NODE P

Under this alternative scheme, however, the generated nodal coordinates are stored in a node buffer. When this buffer is full, the looping process is interrupted and the full buffer delivered to the function evaluation process. Upon return from the function evaluation process, the node generation resumes, filling a new buffer. The process is separated into a loop over the extrapolation levels generating boundary nodes, and then one generating interior nodes to simplify the updating of the partial sums for the quadrature rule. If a buffer contains exclusively boundary nodes, or interior nodes, then the corresponding function values all contribute to the partial sums with the same weight.

A further refinement of the node generation process can be made by generating the nodes in a standard triangle chosen to minimize the amount of arithmetic occurring inside the looping structure of (3.1) and then transforming the standardized nodal coordinates to the domain of integration, D_2, using vectorized arithmetic. The standard triangle used is shown in Figure 3.2A, with vertices $(0,0)$, $(2^K,0)$, $(0,2^K)$ in the (u,v) plane, (K = maximum level of extrapolation.)



(A) STANDARD TRIANGLE (B) DOMAIN OF INTEGRATION

Figure (3.2)

If D_2 has vertices (x_a,y_a), (x_b,y_b), (x_c,y_c) as shown in Figure 3.2B, then the transformation mapping the standard triangle onto D_2 is

(3.2) $$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} x_c \\ y_c \end{pmatrix}$$

where $A = (x_a-x_c)/2^K$

$B = (x_b-x_c)/2^K$

$C = (y_a-y_c)/2^K$

$D = (y_b-y_c)/2^K$.

If the coordinates of the standardized nodes are denoted (u_i,v_i), a vector of n of these can be transformed to nodes (x_i,y_i) of D_2 by

$$\begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} = A \begin{bmatrix} u_1 \\ u_2 \\ \cdot \\ \cdot \\ \cdot \\ u_n \end{bmatrix} + B \begin{bmatrix} v_1 \\ v_2 \\ \cdot \\ \cdot \\ \cdot \\ v_n \end{bmatrix} + \begin{bmatrix} x_c \\ x_c \\ \cdot \\ \cdot \\ \cdot \\ x_c \end{bmatrix} \tag{3.3}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix} = C \begin{bmatrix} u_1 \\ u_2 \\ \cdot \\ \cdot \\ \cdot \\ u_n \end{bmatrix} + D \begin{bmatrix} v_1 \\ v_2 \\ \cdot \\ \cdot \\ \cdot \\ v_n \end{bmatrix} + \begin{bmatrix} y_c \\ y_c \\ \cdot \\ \cdot \\ \cdot \\ y_c \end{bmatrix}$$

involving 4 vector multiplications and additions.

When generating the nodes of the sets E_m-E_{m-1} (new edge nodes) or I_m-I_{m-1} (new interior nodes) in the standard triangle, it is efficient to generate three at a time as indicated in Figure 3.3 for the 3-fold bisection of the triangle.

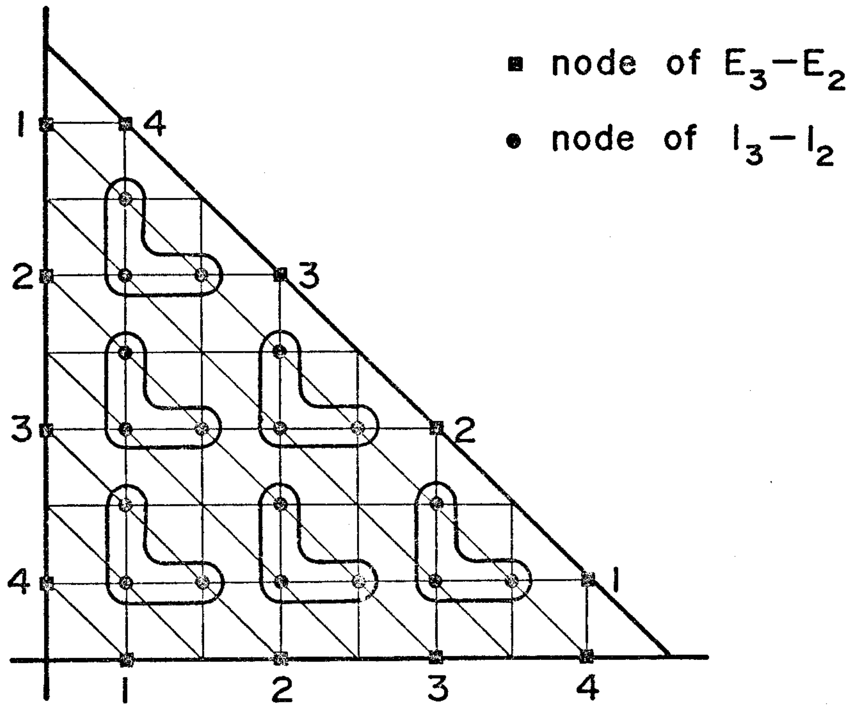


Figure 3.3 - Groupings of Nodes

of E_3-E_2, I_3-I_2

The 12 nodes of E_3-E_2 are generated in a 4 pass loop, the first pass generated the three edge nodes marked "1" and so on. The 18 nodes of I_3-I_2 are generated in a double loop, the outer loop determining the column being scanned, the inner loop determining the row. The first column corresponds to $x = 1$, the second to $x = 3$, and the third to $x = 5$. For the rows corresponding to $y = 1$, 3 or 5 within each column, the cluster of three nodes indicated in Figure 3.3 is generated. This is a form of the loop efficiency improvement technique referred to as loop unravelling [3]; however, its generalization to N dimensions is technically complicated.

These details of the node generation process can be summarized in the expanded form of (3.1), given below. When a buffer has been filled with nodal coordinates from the standard triangle, the vectorized transformations of (3.3) must be applied prior to passing the buffer to the function evaluation process. However, this transformation is algorithmically simply some minor vector preprocessing which is similar to the function evaluation process itself. Hence in this description of node generation, when a buffer is filled, control is passed to a subprocess which will be referred to as BUFFER PROCESSING to perform the transformation, call the vectorized integrand function to obtain integrand values at the quadrature nodes, and update the partial sums of the first column of the Romberg table.

(3.4) NODE GENERATION (FOR K LEVELS OF EXTRAPOLATION)

1. INITIALIZE BUFFER HEADER

   (GENERATE NEW EDGE NODES)

2. FOR m = 1 TO K (m = LEVEL)

   FOR NODES OF $E_m - E_{m-1}$

   IF BUFFER IS FULL

   THEN

   INVOKE BUFFER PROCESSING AND RESET BUFFER HEADER

   ELSE

   GENERATE NEXT THREE NODES AND STORE IN BUFFER

3. INVOKE BUFFER PROCESSING AND RESET BUFFER HEADER

   (GENERATE INTERIOR NODES)

4. FOR m = 1 TO K (m = LEVEL)

   FOR NODES OF $I_m - I_{m-1}$

   IF BUFFER IS FULL

   THEN

   INVOKE BUFFER PROCESSING AND RESET BUFFER HEADER

   ELSE

   GENERATE NEXT THREE NODES AND STORE IN BUFFER

5. INVOKE BUFFER PROCESSING

The extension of this algorithm to a region made up of a group of triangles is straightforward, i.e. when a buffer of standard nodes has been generated, a copy is passed to the BUFFER PROCESSING process for each triangle in the domain. In this way, the unvectorized part of the node generation process is done once for all the triangles and we can expect to obtain an algorithm which, for multiple triangles, runs at essentially vector instruction speed.

§4 Tests on the CDC STAR-100

The buffered algorithm of the preceding section was implemented for the CDC STAR-100 using vector operations provided through STAR FORTRAN [7]* The algorithm was programmed into a main program and three subprograms performing node generation, buffer processing and function evaluation as in §3. The main program consists of initializing the quadrature sums using triangle vertex values (V), calling the node generation subprogram, carrying out the extrapolation, ((2.6)) and printing out. Its execution times were essentially independent of buffer size and integrand and were about .032 seconds for level 6 extrapolation and .052 seconds for level 8. These times are omitted from subsequent reported time figures.

Two integrand functions were used, a simple one

$$f(x,y) = \exp(x+y) \tag{4.1}$$

and a somewhat more complex one

$$f(x,y) = e^{-x}\sin(16\pi(x-y))\sin(16\pi(x+y)). \tag{4.2}$$

The STAR FORTRAN library provides vectorized exponential and sine functions, as well as a vectorized summation routine used to update the partial sums of the quadrature formulae. Extrapolation tables for these functions were built to levels 6 and 8.

In §4.1, the basic results for the buffered algorithm applied to a single triangle are reported. In §4.2, several comparisons are made concerning the interaction between algorithm organization and computer architecture.

---

* We are grateful to R.A. Pimblett and staff of Control Data Canada for their assistance in arranging remote job entry to CDC STAR service from Minneapolis, Minnesota.

In §4.3, tests involving domains triangulated into multiple triangles are reported and in §4.4, some concluding observations are made.

§4.1 A Single Triangle

For these basic tests, the integrands (4.1) and (4.2) are integrated over the triangle with vertices at (0,0), (1,0) and (0,1) to extrapolation levels 6 and 8. In each of Tables 4.1-4.4, the lines show the times and percentage of time used by the three major subprocesses of the buffered algorithm to carry out the same computation using the buffer size indicated for that line. The actual array space used by the buffers was slightly greater than the quoted buffer size to allow for the buffer header (see §3). The times for the very short buffer size of 6 shows anomalous behaviour and is included for interest. The data for level 6 extrapolation from Tables 4.1 and 4.2, excluding the buffer size of 6, is also shown in Figures 4.1 and 4.2.

As the node generation and the buffer processing processes carry out the same sequence of computations for either integrand, the variations of the data in the corresponding columns of Tables 4.1 and 4.2 (or 4.3 and 4.4), indicate the level of variability in the timing of these processes. (For a general discussion of factors influencing timing see [1]). The trends of this data for level 6 and level 8 seem very similar, so a number of subsequent discussions will be carried on for level 6 extrapolation only.

The effect of vectorizing on BUFFER PROCESSING and INTEGRAND EVALUATION seems clear in the reductions of CPU time required with increasing buffer size. A more modest decrease in CPU time with increasing buffer size can be seen in NODE GENERATION; however, it is believed that this is primarily due

LEVEL 6	NODE GENERATION		BUFFER PROCESSING		INTEGRAND EVALUATION		TOTAL*
BUFFER SIZE (words)	SEC.	%	SEC.	%	SEC.	%	SEC.
6	.0943	21	.1660	37	.1915	42	.456
60	.0174	30	.0180	31	.0232	39	.059
120	.0131	36	.0098	27	.0136	37	.037
240	.0110	43	.0056	22	.0089	35	.026
480	.0100	49	.0036	18	.0068	33	.020
960	.0095	53	.0028	16	.0057	31	.018
1920	.0094	55	.0025	15	.0053	30	.017
3840	.0094	54	.0028	16	.0051	30	.017

Table 4.1
Buffered algorithm - Level 6 extrapolation
integrand $f(x,y) = \exp(x+y)$

LEVEL 6	NODE GENERATION		BUFFER PROCESSING		INTEGRAND EVALUATION		TOTAL*
BUFFER SIZE (words)	SEC.	%	SEC.	%	SEC.	%	SEC.
6	.0834	11	.1563	21	.5120	68	.752
60	.0167	17	.0170	17	.0655	66	.099
120	.0128	20	.0093	15	.0414	65	.064
240	.0109	24	.0054	12	.0293	64	.047
480	.0101	28	.0035	10	.0231	62	.037
960	.0096	30	.0027	8	.0206	62	.033
1920	.0095	29	.0026	8	.0202	63	.032
3840	.0095	28	.0025	8	.0214	64	.033

Table 4.2
Buffered algorithm - Level 6 extrapolation
integrand $f(x,y) = e^{-x}\sin(16\pi(x+y))\sin(16\pi(x-y))$

* Exclusive of main program.

LEVEL 8	NODE GENERATION		BUFFER PROCESSING		INTEGRAND EVALUATION		TOTAL*
BUFFER SIZE (words)	SEC.	%	SEC.	%	SEC.	%	SEC.
6	1.450	21	2.565	37	2.980	42	6.995
60	.261	29	.269	30	.357	40	.887
120	.195	35	.145	26	.211	38	.551
240	.163	43	.080	21	.138	36	.381
480	.147	49	.049	16	.104	35	.300
960	.139	53	.034	13	.087	34	.260
1920	.135	56	.027	11	.078	33	.240
3840	.133	57	.028	12	.074	31	.235

Table 4.3
Buffered algorithm - Level 8 extrapolation
integrand $f(x,y) = \exp(x+y)$

LEVEL 8	NODE GENERATION		BUFFER PROCESSING		INTEGRAND EVALUATION		TOTAL*
BUFFER SIZE (words)	SEC.	%	SEC.	%	SEC.	%	SEC.
6	1.284	11	2.424	21	7.987	68	11.70
60	.248	16	.253	17	1.020	67	1.521
120	.189	19	.136	14	.654	67	.979
240	.161	23	.077	11	.467	66	.705
480	.146	26	.048	8	.372	66	.566
960	.140	28	.033	7	.324	65	.497
1920	.137	29	.026	6	.306	65	.469
3840	.135	28	.028	6	.311	66	.474

Table 4.4
Buffered algorithm - Level 8 extrapolation
integrand $f(x,y) = e^{-x}\sin(16\pi(x+y))\sin(16\pi(x-y))$

* Exclusive of the main program.

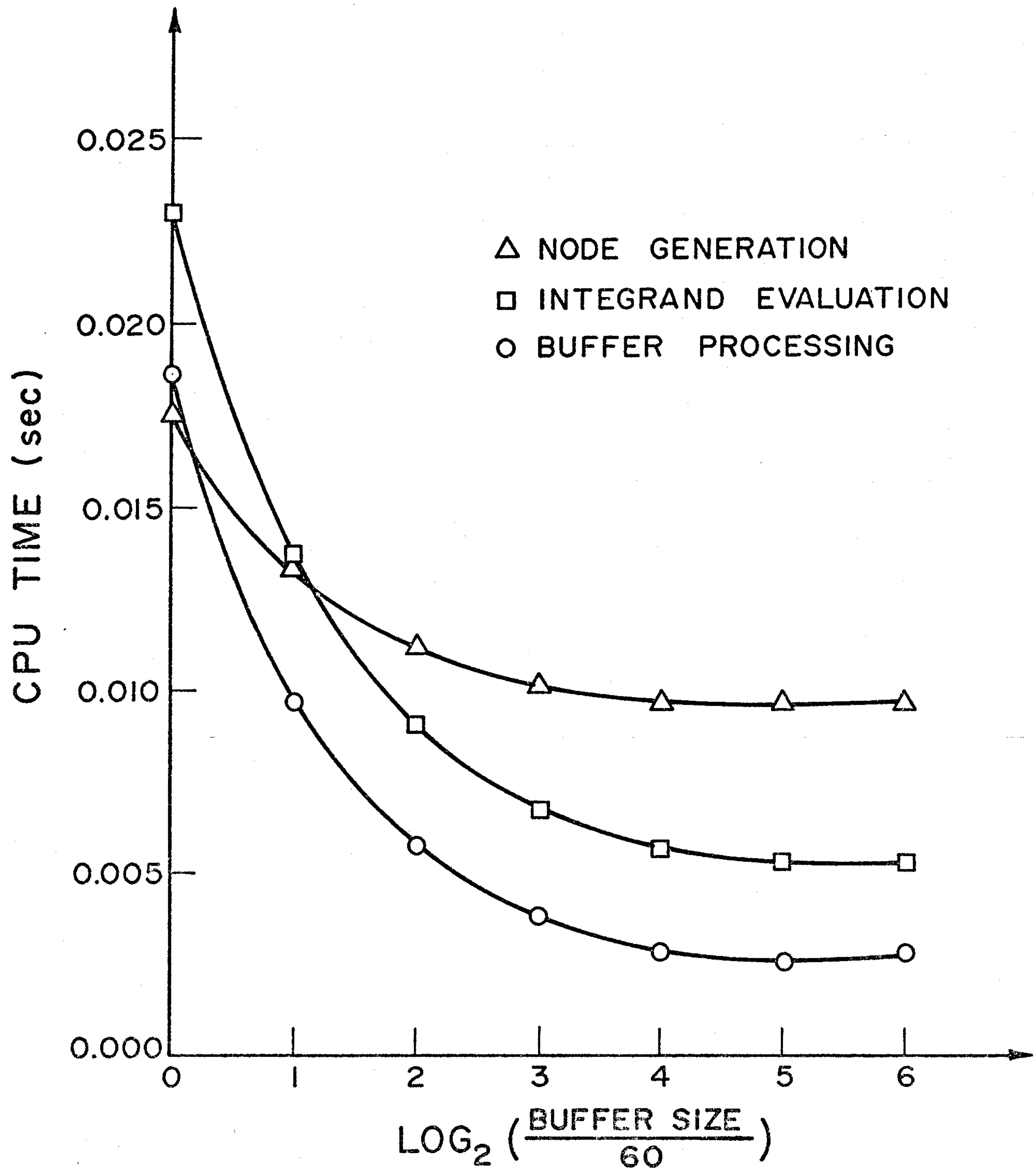


Figure 4.1

Variation of Process times (sec) with buffer size (words)

Integrand (4.1), f(x,y) = exp(x+y)

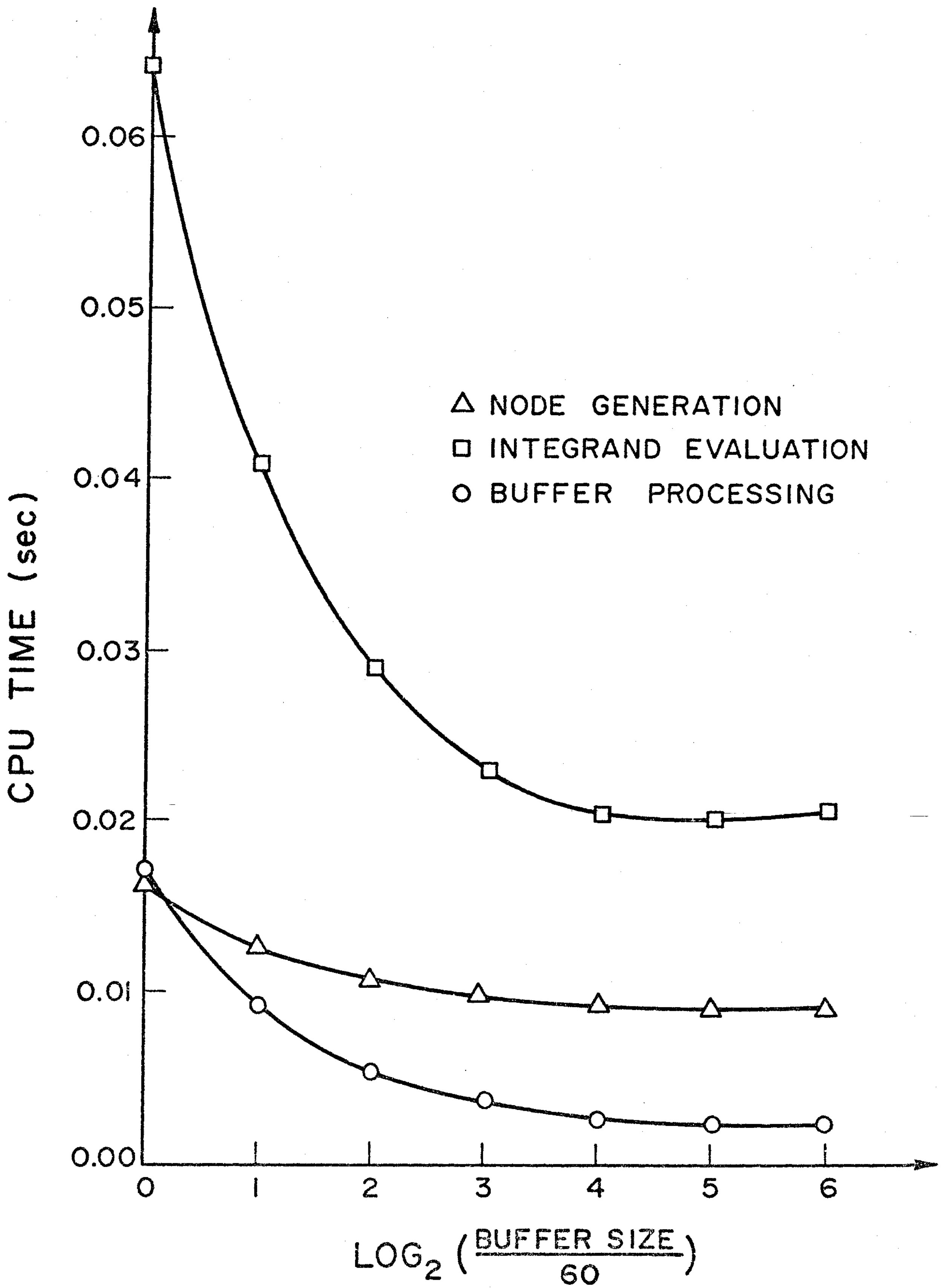


Figure 4.2

Variation of Process times (sec) with buffer size (words)

Integrand (4.2), $f(x,y) = \exp(-x)\sin(16\pi(x+y))\sin(16\pi(x-y))$

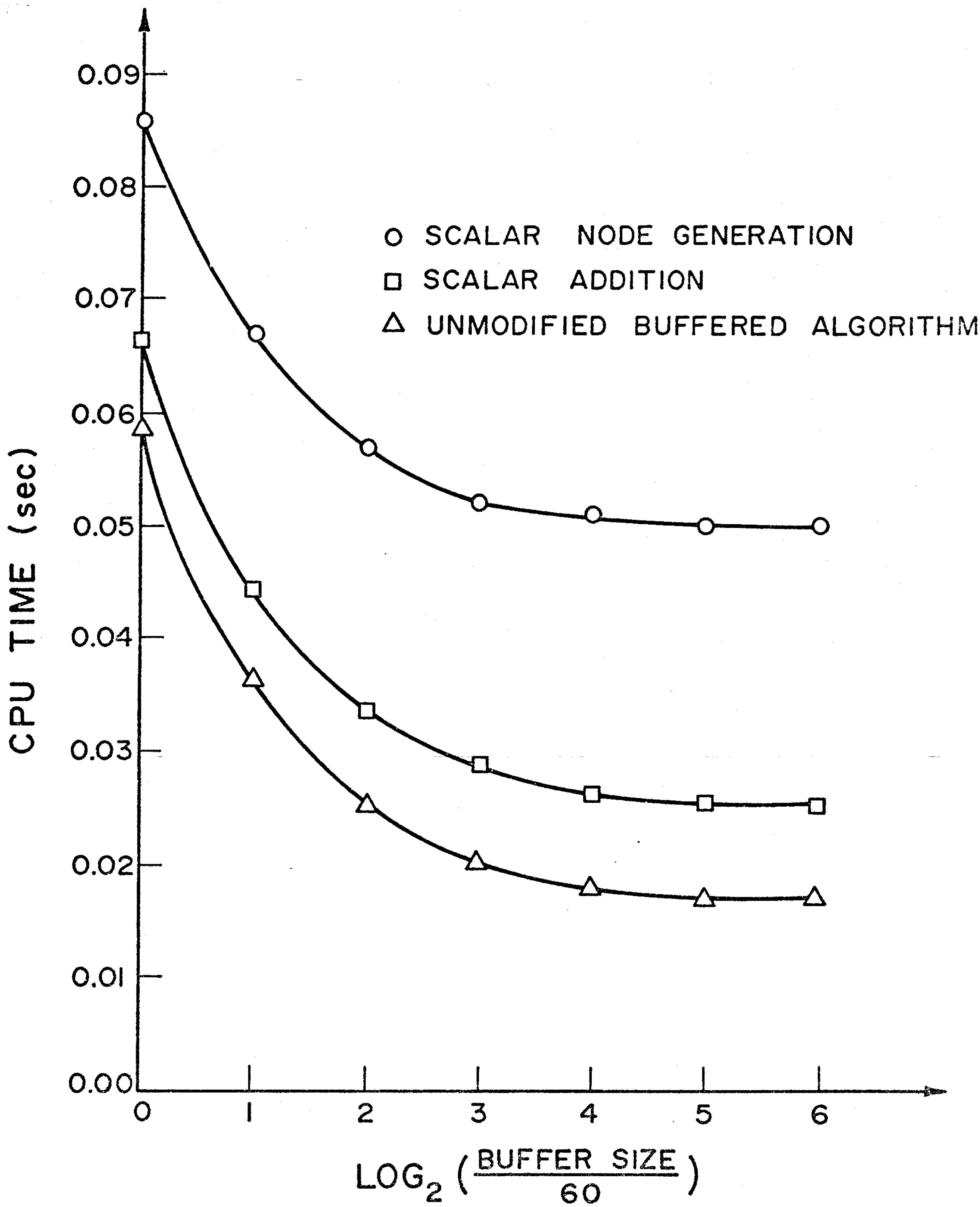


Figure 4.3 Influence of Vector

Instructions in Node Generation and Summation to the decreased overhead associated with managing fewer, longer, buffers. The two parts of the computation affected by vector instructions other than evaluation of the integrand are the transformation of nodal coordinates from the standard triangle to the domain of integration, and the summations of the quadrature sums. To illustrate the influence of vectorizing these two computations, the buffered algorithm was run with the simpler integrand ((4.1)) at level 6 extrapolation with two independent modifications. The first modification involved generating nodes with scalar arithmetic directly in the domain of integration. The second modification involved replacing the vectorized summation process with scalar summations. The influence of these two modifications can be seen in Figure 4.3.

§4.2 Some Comparisons

In this subsection comparisons are made between the running of the conventional and buffered algorithms on the STAR-100 as representative of a vector processor and on the Honeywell 66/60 as representative of a scalar processor. Table 4.5 summarizes the total CPU times used by the STAR in running the alternative algorithms applied to the integrands (4.1) and (4.2) integrated over the same triangle as in §4.1.

	Integrand (4.1)		Integrand (4.2)	
	Level 6	Level 8	Level 6	Level 8
CONVENTIONAL ALGORITHM	.136	2.422	.551	8.845
BUFFERED ALGORITHM BUFFER SIZE = 60 words	.059	.887	.099	1.521
BUFFERED ALGORITHM BUFFER SIZE = 1920 words	.017	.240	.032	.469

Table 4.5
Comparison of Total CPU times (sec.)
for alternative algorithms on CDC STAR-100

Table 4.6 compares total CPU time programs implementing the buffered algorithm and the conventional algorithm as run on the HW 66/60 of the University of Waterloo. As might be expected, there is relatively little dependence of the running time on the buffer size. It is somewhat surprising however that there is so little difference between the running times for the two algorithm organizations.

It is, of course, questionable to attribute time performance measurements directly to algorithms, since the coding of algorithms in a programming language plays a significant role in determining the execution time of the resultant program. Our coding objective was to turn the algorithms of (2.7), (3.1), and (3.4) into FORTRAN programs in as straightforward a way as possible. No effort was made to enhance either implementation through coding techniques, beyond what is explicitly mentioned in §3. The programs used to obtain the times of Table 4.5 and 4.6 only differ in their references to STAR FORTRAN features. The source listing for the STAR FORTRAN implementation and details about the buffer head are given in an Appendix. Hence we feel that the fact that Table 4.6 shows so little effect of varying buffer length and Table 4.5 shows a very significant effect can indeed be attributed to the interaction between architecture and algorithm, rather than details of coding.

	Integrand (4.1)		**Integrand (4.2)**	
	Level 6	Level 8	Level 6	Level 8
CONVENTIONAL ALGORITHM	.515	7.62	.980	13.8
BUFFERED ALGORITHM BUFFER SIZE = 60 words	.561	7.20	.884	14.3
BUFFERED ALGORITHM BUFFER SIZE = 120	.446	7.05	.884	14.3
BUFFERED ALGORITHM BUFFER SIZE = 240	.441	7.00	.898	14.1
BUFFERED ALGORITHM BUFFER SIZE = 480	.435	6.49	.872	14.7
BUFFERED ALGORITHM BUFFER SIZE = 960	.440	6.55	.905	13.7
BUFFERED ALGORITHM BUFFER SIZE = 1920	.447	6.62	.891	13.9

Table 4.6

Comparison of Total CPU times (sec.)
for alternative algorithms on HW 66/60

§4.3 Multiple Triangles

As described in §3, the intention of the buffered algorithm organization is to separate the loop structure of the node generation process, which does not appear well suited to vectorizing, from function evaluation and some other computations which can be vectorized conveniently. The effect of this have been demonstrated in §4.1, in which it can be seen that as the time spent in the vectorized parts of the algorithm drops, node generation takes up an increasingly large fraction of the total time. One might then consider ways to vectorize node generation as a process. However, if the underlying computational problem is really one of estimating an integral over a domain triangulated into multiple triangles, then a simpler avenue is available. As mentioned in §3.1, the nodes of the standard triangle need only be generated once, and then the BUFFER PROCESSING process can be invoked for each triangle of the domain to transform the standard nodes to nodes in that triangle via transformation (3.3). In this way the scalar operation dominated NODE GENERATION process can be amortized over multiple triangles and a computation which proceeds at essentially vector instruction speed results. The buffered algorithm was run for integrating integrands (4.1) and (4.2) over the unit square triangulated into 4, 8, and 16 triangles as shown in Figure 4.4.

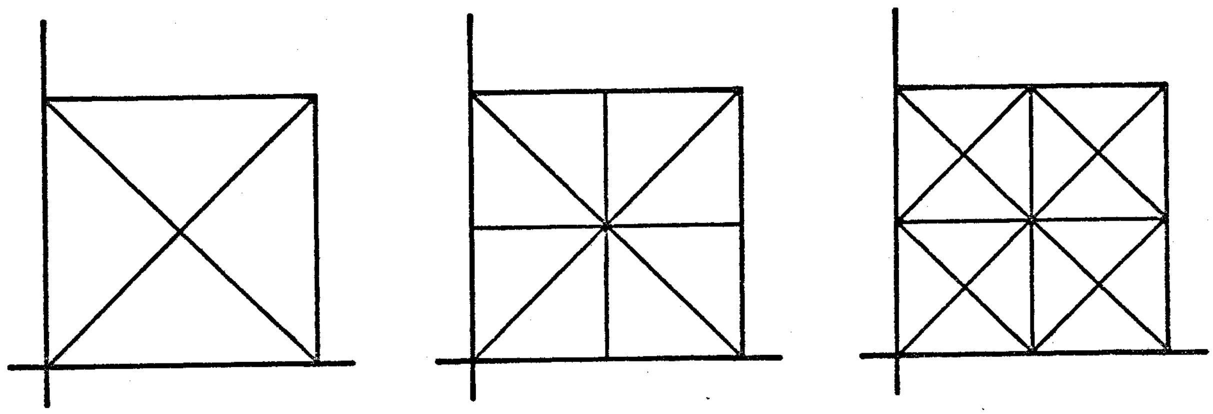


Figure 4.4

The running times per triangle are shown in Tables 4.7, 4.8 and the data of Table 4.8 for 1, 4 and 8 triangles are shown in graphical form in Figure 4.5. As can be seen in the tables, the times per triangle for 8 and 16 triangles are essentially the same.

To estimate the speed up gained by this technique, we may use .008 sec. and .023 sec. as the execution times per triangle for the buffered algorithm applied to 8 to 16 triangles at level 6 extrapolation and to integrands (4.1) and (4.2) respectively. The times for the corresponding calculations done on the STAR with the conventional algorithm, as taken from Table 4.5, are .136 sec. and .551 sec. Using these figures then, we can estimate that the running times for the calculation have been reduced by factors $.008/.136 \approx 1/17$ and $.023/.551 \approx 1/24$ respectively, which are the basis for our remarks of the introductory section.

BUFFER SIZE	SINGLE TRIANGLE	4 TRIANGLES	8 TRIANGLES	16 TRIANGLES
60	.0587	.0403	.0379	.0370
120	.0365	.0242	.0225	.0217
240	.0256	.0160	.0147	.0141
480	.0203	.0124	.0108	.0103
960	.0181	.0104	.0092	.0087
1920	.0172	.0097	.0085	.0080
3840	.0173	.0097	.0083	.0078

Table (4.7)
Total CPU Time/Triangle at Level 6
for Integrand $f(x,y) = \exp(x+y)$

BUFFER SIZE	SINGLE TRIANGLE	4 TRIANGLES	8 TRIANGLES	16 TRIANGLES
60	.0993	.0831	.0800	.0796
120	.0636	.0527	.0501	.0500
240	.0456	.0369	.0347	.0344
480	.0367	.0289	.0270	.0271
960	.0329	.0255	.0237	.0239
1920	.0324	.0241	.0224	.0227
3840	.0334	.0246	.0228	.0229

Table (4.8)
Total CPU Time/Triangle at Level 6
for Integrand $f(x,y) = \exp(-x)\sin(16\pi(x+y))\sin(16\pi(x-y))$

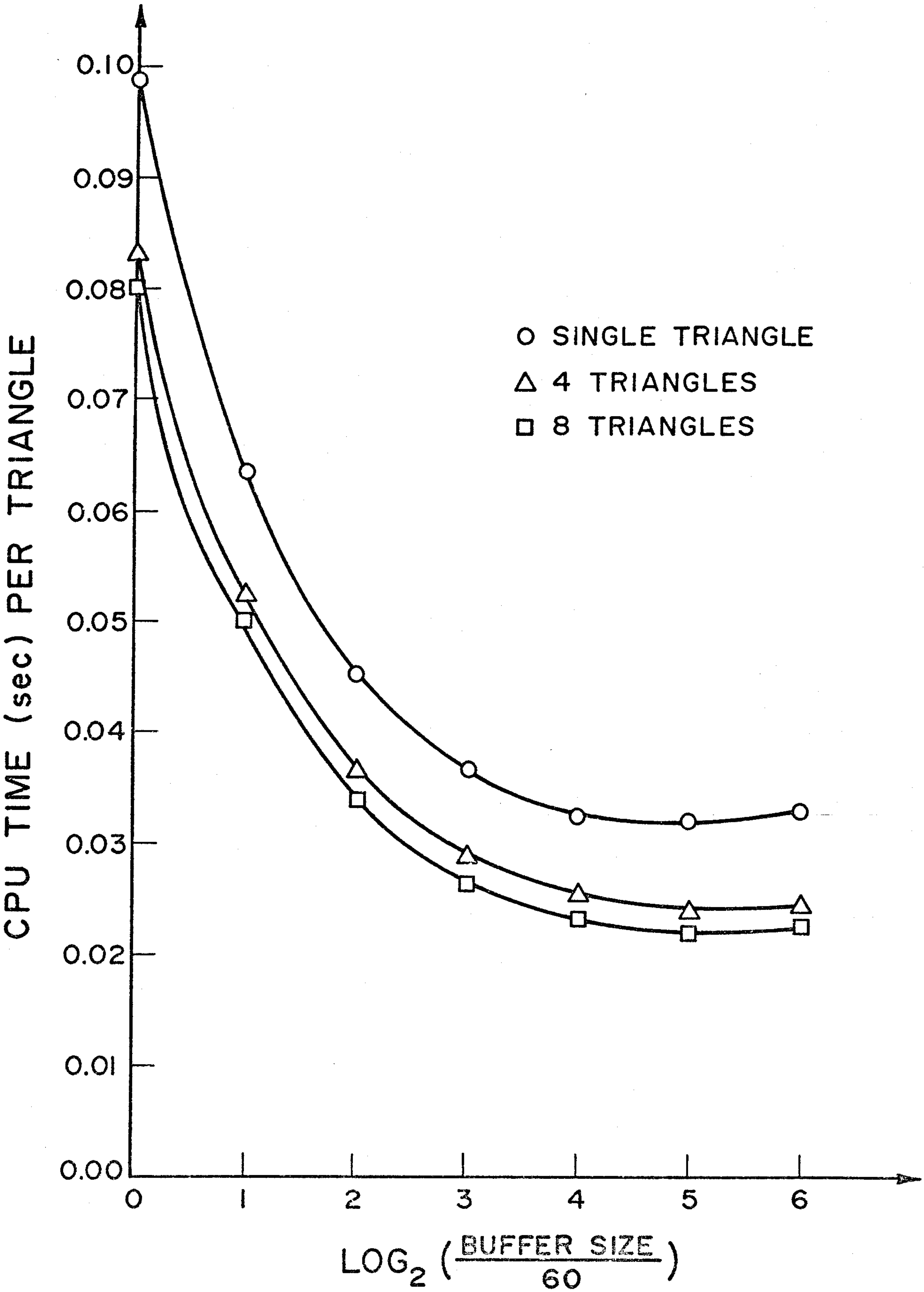


Figure 4.5 Total CPU Time/Triangle at Level 6

for Integrand $f(x,y) = \exp(-x)\sin(16\pi(x+y))\sin(16\pi(x-y))$

§4.4 Some Observations

These experimental computations indicate that for multi-dimensional quadrature calculations done with vector processors, the buffered algorithm organization can offer significant running time improvements, (reductions by factors down to 1/20,) over algorithm organizations which are appropriate for scalar processor computer architectures. Tables 4.1-4.4 and Figures 4.1 and 4.2 show that nearly all of the benefits of the buffered algorithm have been reached with buffers of size about 1000 words. The increase in program complexity is not substantial so the additional memory requirements for the buffered algorithm are reasonably modest.

We will speculate further about the appropriate buffer size for the algorithm. Let us hypothesize that the time spent by the algorithm on one buffer of length ℓ (i.e. time to generate nodes for it, to process them into function values, and to add its contribution to the partial sums of the quadrature rules) can be characterized by two parameters, a 'fixed' time, t_f and a 'variable' time, t_v as

$$(4.3) \qquad t = t_f + t_v \ell$$

The number of buffers used in any one computation depends on ℓ and will be denoted $N(\ell)$, with

$$(4.4) \qquad \ell N(\ell) = C.$$

C is a constant determined by the total number of nodes required for the computation. The total time to do the computation will be

$$\begin{aligned} T(\ell) &= N(\ell)t \\ &= N(\ell)(t_f + t_v\ell) \\ &= C(t_f/\ell + t_v) \end{aligned}$$

If the computation is done twice, with buffer lengths ℓ_1 and $\ell_2 > \ell_1$, then, under (4.3) the relative improvement using the longer buffer is

$$(4.5) \qquad \begin{aligned} &(T(\ell_1) - T(\ell_2))/T(\ell_2) \\ &= t_f(1/\ell_1 - 1/\ell_2)/(t_f/\ell_2 + t_v) \\ &= (\ell_2/\ell_1 - 1)/(1 + \ell_2(t_v/t_f)) \end{aligned}$$

Notice that this relative improvement is independent of C of (4.4), the size of the computation. This effect is suggested by the similarity in the shapes of the curves in Figures 4.1 and 4.2.

In our computations, we have repeatedly doubled the buffer size, i.e. $\ell_2 = 2\ell_1$; (4.5) suggests that the relative benefit of doubling the buffer size is

$$(4.6) \qquad r(\ell_2) = 1/(1 + \ell_2(t_v/t_f))$$

or

$$(4.7) \qquad t_v/t_f = (1/r(\ell_2) - 1)/\ell_2$$

The right hand side of (4.7) is immediately computable from successive lines of Tables 4.1-4.4, and for any cases where $r(\ell_2) > .1$, this expression lies between 5.0×10^{-3} and 7.0×10^{-3}. The relative constancy of this expression lends credence to (4.3) as a hypothesis. But more striking is the fact that this ratio $t_v/t_f \cong 5 \times 10^{-3}$ is quite close to the ratio i_v/i_f of the parameters i_f and i_v used to describe the execution time of vector instructions in terms of vector length L. The parameter i_f, is referred to as the set up time, and the time to add two vectors of length L on the CDC STAR-100 is given as ([5])

$$t_{add} = i_f + i_v L \quad \text{for} \quad i_v/i_f \cong 7.2 \times 10^{-3}$$

and for multiplication

$$t_{mult} = i_f + i_v L \quad \text{for} \quad i_v/i_f \cong 6.3 \times 10^{-3}$$

This suggests the conjecture that the relative improvement returned by increasing the buffer length from ℓ_1 to ℓ_2 could be estimated, a priori, from (4.6) with ratio t_v/t_f replaced by a typical ratio of vector instruction timing parameters i_v/i_f. This would enable one to determine reasonable buffer lengths for the buffered algorithm from essentially machine constants of a particular vector processor.

APPENDIX

STAR-100 FORTRAN PROGRAM

```
C    THIS PROGRAM IS WRITTEN IN STAR FORTRAN LANGUAGE.
C    IT PERFORMS EXTRAPOLATION ON A GENERAL TRIANGLE
C    USING A LINEAR RULE AT THE VERTICES.
C    IN THE MAIN PROGRAM (EXTRAP) THE INITIAL APPROXIMATION
C    TO THE INTEGRAL IS COMPUTED AND NODEGEN SUBPROGRAM
C    IS CALLED FOR THE COMPUTATIONS ON HIGHER LEVELS.
C    UPON RETURN FROM NODEGEN, EXTRAPOLATION IS PERFORMED
C    ON THE ROMBERG TABLE PRODUCED BY THE
C    OTHER SUBPROGRAMS (FEVAL).
C    THE FOLLOWING VARIABLES ARE USED IN THE PROGRAM:
C       XC(I)-          X-COORDINATES OF THE VERTICES
C                       (INPUT)
C       YC(I)-          Y-COORDINATES OF THE VERTICES
C                       (INPUT)
C       AREA  -         AREA OF THE TRIANGLE
C       TROMB(I)-       ROMBERG TABLE
C       XD1,YD1 -       TRANSFORMATION ELEMENTS
C       XD2,YD2 -       TRANSFORMATION ELEMENTS
C       BUF(I)  -       BUFFER USED FOR STORING
C                       NODAL INFORMATION
C                       AND FUNCTION VALUES. (VECTOR)
C       LMAX    -       MAXIMUM LEVEL OF EXTRAPOLATION
C                       (INPUT)
C       IBUFSIZ -       BUFFER LENGTH (INPUT)
C
      PROGRAM EXTRAP(UNIT6=OUTPUT,UNIT5=INPUT)
      DIMENSION XC(3),YC(3)
      COMMON BUF(5000),TROMB(20),XD1,XD2,YD1,YD2,X3,Y3
     1,TNODE,TFEVAL,TFUN
      TROMB(1;20)=0.
C
C    INPUT THE X-COORD AND Y-COORD OF THE TRIANGLE
C
      READ(5,100)(XC(I),I=1,3)
      READ(5,100)(YC(I),I=1,3)
  100 FORMAT(3F10.6)
  200 T1=SECOND(CPU)
      TFEVAL=0.
      TFUN=0.
      READ(5,101,END=999)LMAX,KSIZ
  101 FORMAT(2I10)
      IBUFSIZ=6*KSIZ+2*LMAX+1
      WRITE(6,112)LMAX,IBUFSIZ
  112 FORMAT(10X,I10,5X,I10)
C
C    SET UP THE ENTRIES OF TRANSFORMATION MATRIX
C
      LEV2=2**LMAX
```

```
      X3=XC(3)
      Y3=YC(3)
      XD1=(XC(1)-XC(3))/LEV2
      XD2=(XC(2)-XC(3))/LEV2
      YD1=(YC(1)-YC(3))/LEV2
      YD2=(YC(2)-YC(3))/LEV2
C
C   COMPUTE THE AREA AND FIRST APPROXIMATION
C
      AREA=XC(2)*YC(3)-XC(3)*YC(2)-XC(1)*(YC(3)-YC(2))+YC(1)*
     3(XC(3)-XC(2))
      AREA=0.5*ABS(AREA)
      TROMB(1)=AREA/3.*(EXP(XC(1)+YC(1))+EXP(XC(2)+YC(2))+
     4EXP(XC(3)+YC(3)))
      WRITE(6,108)TROMB(1)
  108 FORMAT(10X,E21.14)
C
      CALL NODEGEN(LMAX,IBUFSIZ)
C
C UPDATE THE FIRST ROW OF THE ROMBERG TABLE AND EXTRAPOLATE
C
      DO 2 I=1,LMAX
        TROMB(I+1)=0.25*TROMB(I)+AREA*TROMB(I+1)/(3.*4.**I)
    2 CONTINUE
      DO 3 I=1,LMAX
        WRITE(6,108)TROMB(I+1)
    3 CONTINUE
      DO 5 J=1,LMAX
        JN=LMAX-J+1
        DO 4 K=1,JN
          TROMB(K)=TROMB(K+1)+(TROMB(K+1)-TROMB(K))/(4.**J-1.)
          WRITE(6,110)J,TROMB(K)
    4   CONTINUE
  110 FORMAT(5X,I10,5X,E21.14)
    5 CONTINUE
      TIME=SECOND(CPU)-T1
      WRITE(6,111)TIME
      TNODE=TNODE-TFEVAL
      WRITE(6,111)TNODE
      TFEVAL=TFEVAL-TFUN
      WRITE(6,111)TFUN
  111 FORMAT(15X,E14.7)
      GO TO 200
  999 STOP
      END
```

```
C     THIS SUBROUTINE GENERATES A NODE BUFFER AND PASSES IT
C     TO FEVAL SUBPROGRAM FOR PROCESSING UNTIL ALL THE
C     BUFFERS ARE GENERATED TO LEVEL=LMAX
C     VARIABLES USED IN THIS ROUTINE:
C       LP-            POINTER TO BUFFER
C       LMAX-          MAXIMUM LEVEL OF EXTRAPOLATION
C       IBUFSIZ-       BUFFER LENGTH
C       BUF(I)-        NODE BUFFER (VECTOR)
C       IHEAD-         A POINTER TO BUFFER HEADER
C
C
        SUBROUTINE NODEGEN(LMAX,IBUFSIZ)
        DIMENSION X(3),Y(3)
        COMMON BUF(5000),TROMB(20),XD1,XD2,YD1,YD2,X3,Y3
       4,TNODE,TFEVAL,TFUN
        TNODE=SECOND(H)
        LEVMAX=LMAX
        LEV2=2**LEVMAX
        LP=2*(LMAX+1)
        K3=(IBUFSIZ-2*LMAX-1)/2
        IHEAD=2
        BUF(1)=1.
C
C     GENERATE EDGE NODES   E
C
        DO 2 LEVEL=1,LEVMAX
          JFAC=LEV2/(2**LEVEL)
          INC=2*JFAC
          JBOU=LEV2-JFAC
          DO 3 J=JFAC,JBOU,INC
            IF(LP.LT.(IBUFSIZ-K3)) GO TO 4
            BUF(IHEAD)=-1.
            BUF(IHEAD+LMAX)=LEVEL
            CALL FEVAL(LMAX,LP-1,IBUFSIZ)
            IHEAD=2
            LP=2*(LMAX+1)
  4         BUF(LP)=0.0
            BUF(LP+1)=J
            BUF(LP+2)=J
            BUF(LP+K3)=0.0
            BUF(LP+K3+1)=J
            BUF(LP+K3+2)=LEV2-J
            LP=LP+3
  3       CONTINUE
          BUF(IHEAD)=LP-1
          BUF(IHEAD+LMAX)=LEVEL
          IHEAD=IHEAD+1
  2     CONTINUE
        IF(LP.GT.2*(LMAX+1)) CALL FEVAL(LMAX,LP-1,IBUFSIZ)
        IHEAD=2
        LP=2*(LMAX+1)
```

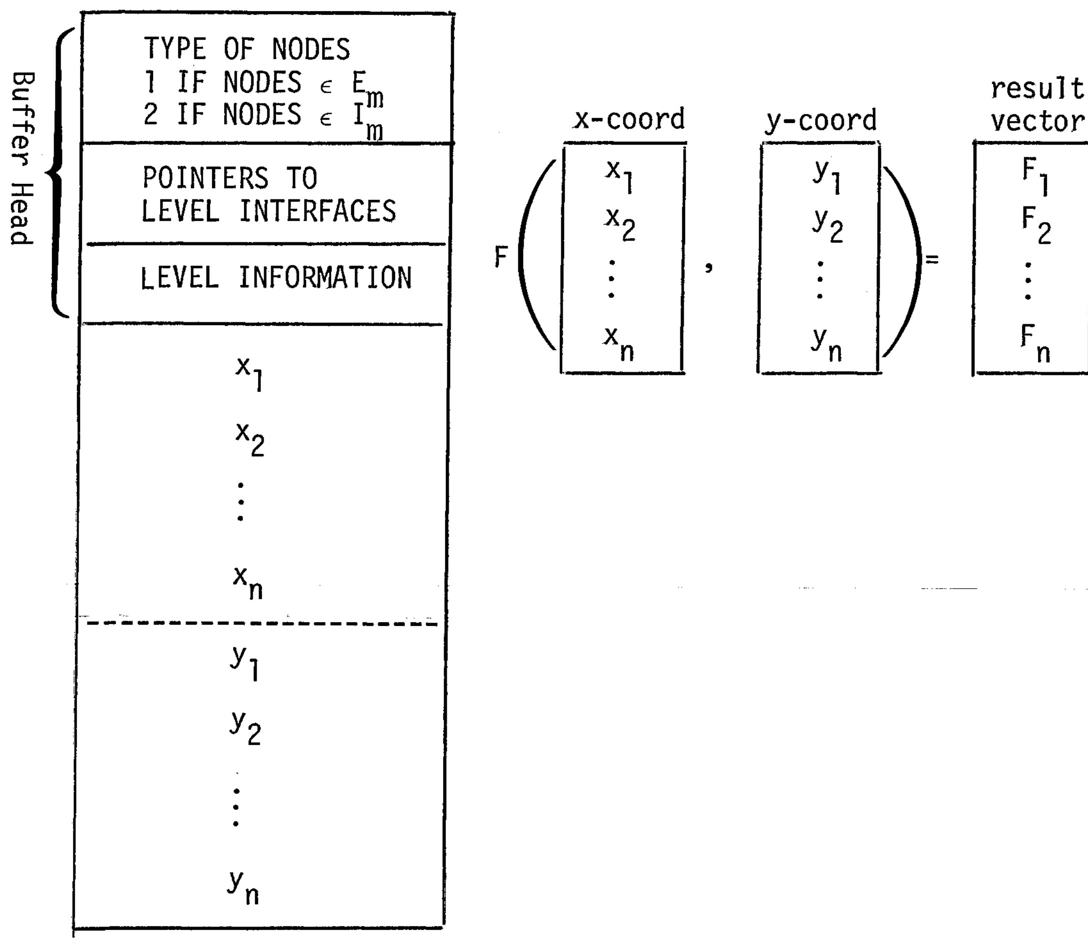
```
      BUF(1)=2.
C
C     GENERATE INTERIOR NODES   I
C
      DO 5 LEVEL=2,LEVMAX
        IFAC=LEV2/2**LEVEL
        IFAC2=2*IFAC
        IEND=LEV2-3*IFAC
        DO 6 I=IFAC,IEND,IFAC2
          JEND=LEV2-I-IFAC2
          DO 7 J=IFAC,JEND,IFAC2
            IF(LP.LT.(IBUFSIZ-K3)) GO TO 8
            BUF(IHEAD)=-1.
            BUF(IHEAD+LMAX)=LEVEL
            CALL FEVAL(LMAX,LP-1,IBUFSIZ)
            IHEAD=2
            LP=2*(LMAX+1)
 8          BUF(LP)=I
            BUF(LP+1)=I+IFAC
            BUF(LP+2)=I
            BUF(LP+K3)=J
            BUF(LP+K3+1)=J
            BUF(LP+K3+2)=J+IFAC
            LP=LP+3
 7        CONTINUE
 6      CONTINUE
        BUF(IHEAD)=LP-1
        BUF(IHEAD+LMAX)=LEVEL
        IHEAD=IHEAD+1
 5    CONTINUE
      IF(LP.GT.2*(LMAX+1)) CALL FEVAL(LMAX,LP-1,IBUFSIZ)
      TNODE=SECOND(H)-TNODE
      RETURN
      END
```

```
C     THIS ROUTINE EVALUATES FUNCTION AT THE NODES OF
C     THE OBJECT TRIANGLE USING VECTOR OPERATIONS.
C
      SUBROUTINE FEVAL(LMAX,ISIZE,IBUFSIZ)
      COMMON BUF(5000),TROMB(20),XD1,XD2,YD1,YD2,X3,Y3
     6,TNODE,TFEVAL,TFUN
      T1=SECOND(H)
      IS=2*(LMAX+1)
      K3=(ISIZE-2*LMAX-1)
      K4=(IBUFSIZ-2*LMAX-1)/2
      ASSIGN PXVAL,BUF(IS;K3)
      ASSIGN PYVAL,BUF(IS+K4;K3)
      ASSIGN TEMPX,.DYN.K3
C
C     PERFORM TRANSFORMATION TO ORIGINAL TRIANGLE
C     USING VECTOR OPERATIONS
C
      TEMPX=XD1*PXVAL+XD2*PYVAL+X3
      PYVAL=YD1*PXVAL+YD2*PYVAL+Y3
      PXVAL=TEMPX
      T2=SECOND(E)
C
C     INVOKE VECTOR FUNCTION SUBPROGRAM F
C
      BUF(IS;K3)=F(PXVAL,PYVAL;BUF(IS;K3))
      TFUN=TFUN+SECOND(E)-T2
      IPOINT=2
      LEV=BUF(IPOINT+LMAX)
      ISTART=IS
      WEIGHT=6.
      IF(BUF(1).EQ.1.)WEIGHT=3.
      BUF(IS;K3)=WEIGHT*BUF(IS;K3)
    1 IFINIS=ISIZE
      IF(BUF(IPOINT).NE.-1.)IFINIS=BUF(IPOINT)
      LENFUN=IFINIS-ISTART+1
C
C     UPDATE PARTIAL SUMS OF ROMBERG TABLE
C
      TROMB(LEV+1)=TROMB(LEV+1)+Q8SSUM(BUF(ISTART;LENFUN))
      ISTART=BUF(IPOINT)+1
      IPOINT=IPOINT+1
      LEV=BUF(IPOINT+LMAX)
      IF(ISTART.LT.ISIZE.AND.LEV.GT.0.AND.IPOINT.LE.(LMAX+1))GO TO 1
      BUF(2;2*LMAX)=0.
      TFEVAL=TFEVAL+SECOND(H)-T1
      FREE
      RETURN
      END
```

```
C     VECTOR FUNCTION SUBPROGRAM TO COMPUTE
C     THE FUNCTION AT THE NODES OF THE BUFFER
C     IT RETURNS A VECTOR OF FUNCTION VALUES
C
      FUNCTION F(PXVAL,PYVAL;*)
      DESCRIPTOR F,PXVAL,PYVAL
      PXVAL=PXVAL+PYVAL
      F=VEXP(PXVAL;F)
      RETURN
      END
```

Appendix

STRUCTURE OF A BUFFER



References

1. Gentleman, W.M. and Wichman B., "Timing on Computers", SIGARCH, Computer Architecture News, Vol. 2, No. 3, 1973, pp. 20-23.

2. Kahaner, D.K. and Wells M.B., "An Algorithm for N-Dimensional Adaptive Quadrature Using Advanced Programming Techniques", Los Alamos Scientific Labratories Report, Nov. 1976, to appear.

3. Knuth, D.E., "Structured Programming with go to Statements", Computing Surveys, Vol. 6, No. 4, 1974, pp. 262-301.

4. Lyness, J.N., "Quadrature over a Simplex: Part I and II", submitted to SIAM J. Num. Anal. (1976).

5. Trivedi, K.S., "Prepaging and Applications to the STAR-100 Computer", High Speed Computer and Algorithm Organization, Proc. of Symposium of U. of Illinois, 1977, edit D.J. Kuck, D.H. Lawrie, A.H. Sameh, Academic Press, (See references).

6. Ramamoorthy, C.V. and Li, H.F., "Pipeline Architecture", Computing Surveys, Vol. 9, No. 1, 1977, pp. 61-102.

7. STAR FORTRAN LANGUAGE, Version 2, Reference Manual, Control Data Corporation, 1977.

8. Voigt, R.G., "The Influence of Vector Computer Architecture on Numerical Algorithms", High Speed Computer and Algorithm Organization, Proc. of Symposium at U. of Illinois, 1977, edit D.J. Kuck, D.H. Lawrie, A.H. Sameh, Academic Press.

9. Zienkiewicz, O., "The Finite Element Method in Engineering Science", McGraw-Hill, 1971.