Journal of Computational and Applied Mathematics 35 (1991) 119–132
North-Holland

On the subdivision strategy in adaptive quadrature algorithms

Jarle Berntsen
Institute of Marine Research, Nordnesgaten 33, N-5000 Bergen, Norway

Terje O. Espelid and Tor Sørevik
Department of Informatics, University of Bergen, Thormøhlensgate 55, N-5008 Bergen, Norway

Received 20 October 1990

Abstract

Berntsen, J., T.O. Espelid and T. Sørevik, On the subdivision strategy in adaptive quadrature algorithms, Journal of Computational and Applied Mathematics 35 (1991) 119–132.

The subdivision procedure used in most available adaptive quadrature codes is a simple bisection of the chosen interval. Thus the interval is divided in *two equally sized parts*. In this paper we present a subdivision strategy which gives *three nonequally sized parts*. The subdivision points are found using only available information. The strategy has been implemented in the QUADPACK code DQAG and tested using the "performance profile" testing technique. We present test results showing a significant reduction in the number of function evaluations compared to the standard bisection procedure on most test families of integrands.

Keywords: Adaptive quadrature, subdivision.

1. Introduction

Automatic algorithms are now used widely for the numerical calculation of integrals. Since the first such algorithm was given by McKeeman [13,14] in 1962, many new and sophisticated algorithms, both adaptive and nonadaptive, have been developed, among these [4,15,16]. For a more complete reference see [3, pp. 425–434].

Adaptive quadrature algorithms have a general structure consisting of the following four steps:

(1) Choose an interval from a set of subintervals.

(2) Subdivide the chosen interval.

(3) Compute local approximations to the integrals and estimate their errors for each new subinterval.

(4) Update the interval collection, the global integral approximation and the estimate of the global error. Check for convergence and, if necessary, repeat from step (1).

A lot of work has been done, since McKeeman's routine was published, to improve automatic quadrature software. The designer of such software has to make a number of decisions [17]. First of all he has to choose the basic quadrature rule to be used; McKeeman picked Simpson's rule, e.g. in QUADPACK [16] Gauss–Kronrod rules are used, while quite recently Berntsen and Espelid gave some arguments for choosing Gauss rules [2].

An aspect that has attracted some attention is the question of whether a local or global convergence criteria is to be preferred [3,12,18].

Observe furthermore that the estimate of the true error in the approximation of the integral governs the decision on whether to return the current approximation and terminate or to continue. Both the efficiency and the reliability therefore depend heavily on the error estimating procedure, a problem that has got a lot of attention; e.g., [2,4,5,8,9,16].

To our knowledge only a few papers [6,10,19] discuss *the subdivision strategy* used in adaptive quadrature software. Almost every published routine of this type uses a simple bisection strategy, that is, pick the interval to be processed next and *divide it in two equally sized parts*. One exception to this is McKeeman's original algorithm which is based on trisection, that is uniform 3-division. Lyness [10] discusses the use of bisection versus trisection in this adaptive code based on the Simpson rule and concludes that bisection is superior to trisection since the function values can be reused. Both Hanke [6] and Sørevik [19] discuss the more general strategy: divide in $p > 1$ equally sized parts with no reuse of function values. They conclude, based on theory (model problems) and experiments, that $p = 2$ is not the optimal choice. The "best" value is of course problem-dependent, but $p = 3$ (trisection) seems generally to be slightly better than bisection.

We will, in this paper, focus on the subdivision strategy used in adaptive quadrature routines. We will allow the interval to be divided in more than two *nonequal* parts. It turns out that on problems where adaptability is important such a simple idea can reduce the work by 25–60%. In spite of the huge literature on one-dimensional numerical integration, we are not aware of any paper which addresses this particular idea. The only article which has some relevance to this topic is the paper by Hazlewood [7]. However, his idea is based on Newton–Cote rules and is applied in a locally adaptive subdivision strategy.

The outline of this paper is as follows. In the next section we present the basic adaptive p-division quadrature algorithm. In Section 3 we discuss, in some detail, alternative ways of implementing a nonuniform 3-division algorithm. In Section 4 we present some test results and finally in Section 5 we give some concluding remarks.

2. A globally adaptive quadrature algorithm.

We define the integral to be computed by

$$If = \int_a^b f(x)\,\mathrm{d}x.$$

An adaptive quadrature algorithm has input a, b, f and an error tolerance ϵ. The output will normally be an estimate of the integral $\hat{Q}$ and an error estimate $\hat{E}$. If possible, the algorithm computes $\hat{Q}$ with $\hat{E} \leqslant \epsilon$. This does not ensure that $|If - \hat{Q}| \leqslant \epsilon$, but it does suggest that it is likely.

At a certain stage in an adaptive quadrature algorithm we have M intervals, say H_k, $k = 1,\dots,M$ (*an interval collection*). Furthermore we have, for each interval in the collection, an approximation $\hat{Q}_k$ to the integral and an error estimate $\hat{E}_k$ to this approximation. The basic structure in such an algorithm is then the following.

A globally adaptive quadrature algorithm.

```
Initialize:          Initialize the interval collection and put $M = 1$; Produce $\hat{Q}_1$ and $\hat{E}_1$; Put
                     $\hat{Q} = \hat{Q}_1$ and $\hat{E} = \hat{E}_1$;
Control:             while $\hat{E} > \epsilon$ do
                     begin
                     Pick an interval from the collection; say interval $H_k$;
                     Divide this interval in $p$ parts;
Process intervals:   Compute: $\hat{Q}_k^{(i)}$, $\hat{E}_k^{(i)}$, $i = 1,\dots, p$;
Update:              $\hat{Q} = \hat{Q} + (\Sigma_{i=1}^{p}\hat{Q}_k^{(i)}) - \hat{Q}_k$;
                     $\hat{E} = \hat{E} + (\Sigma_{i=1}^{p}\hat{E}_k^{(i)}) - \hat{E}_k$;
                     Let these $p$ new intervals replace interval $H_k$ in the collection and put
                     $M = M + (p - 1)$;
                     end
```

Note that when we pick an interval, say H_k, from the interval collection, we need information associated with that interval both in the step *Process intervals* and the step *Update*, e.g., $\hat{Q}_k$, $\hat{E}_k$ and interval bounds $[a_k, b_k]$.

If $p > 1$ is kept fixed and the subdivision is uniform, then the new interval bounds are produced easily. However, if we allow p to vary, but still use a uniform subdivision, then the actual p-value to be used on H_k has to be saved too (we assume that which p-value to use may depend on the function values used to produce $\hat{Q}_k$).

A nonuniform subdivision of H_k is slightly more complicated: we may save the future subdivision points along with the rest of the information on H_k (including p in case its value is allowed to vary). This choice is based on the fact that p is much smaller than the number of

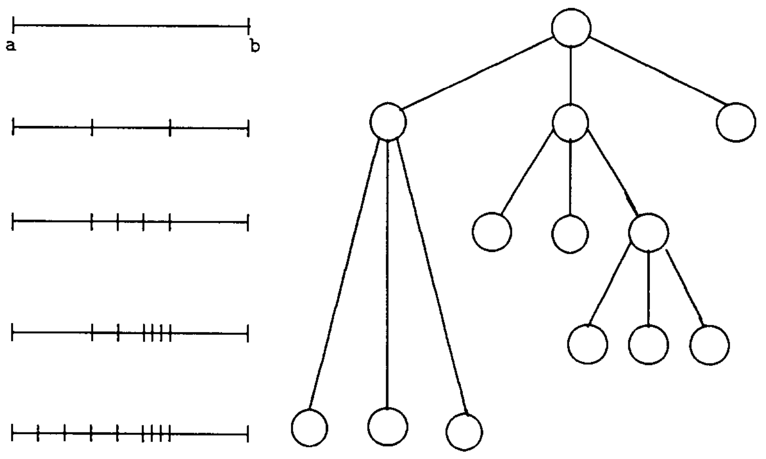

Fig. 1. A subdivision of an interval and the associated subdivision tree: $s = 4$ and $p = 3$.

function values n used by the basic quadrature rule Q and thus the subdivision points are cheaper to store than the n function values. Thus, we take the trouble to compute future subdivision points for all intervals without knowing whether this interval ever will be picked from the collection.

In order to illustrate how such an algorithm works, let us stop the algorithm after s subdivision steps and illustrate what has happened by a subdivision tree. In such a tree each node represents an interval. A node is either a leave (an interval that has not been subdivided so far in the process) or an interior node with exactly p branches. The root represents the original interval $[a, b]$, while the leaves represent the interval collection at any time. Thus the leaves represent the final partition of the original interval when the algorithm stops. In Fig. 1 we illustrate such a subdivision tree.

The statements in the following theorem are easy to prove.

Theorem 1 (p-division). *In a p-division algorithm, with p fixed, the total number of intervals that have been processed after s subdivision steps is* $N = 1 + ps$. *This number is equal to the total number of nodes in the subdivision tree. The number of interior nodes in the subdivision tree is* $N_{\text{interior}} = s$ *and the number of leaves is* $N_{\text{leaves}} = 1 + (p - 1)s$.

Note that when the algorithm stops, then only the leaves in the tree have information that contribute to the final estimates $\hat{E}$ and $\hat{Q}$. The internal nodes have been used to find the final subdivision, but are, with respect to the final estimates, in some sense "waisted". Assume that the total work W_p done by a p-division algorithm is essentially proportional to the number of processed intervals and thus the number of nodes in the subdivision tree

$$W_p \approx C(1 + ps).$$

Observe that approximately a $(p - 1)/p$ part of the total work contributes to the final estimate in a p-division algorithm. This part is smallest for $p = 2$ and increases with p. Let us formulate this fact in the next observation.

Observation. Assume that the final partitions of the original interval have approximately the same distribution (and thus the same number of points) for both the 2- and 3-division strategy; then

$$\tfrac{1}{2} W_2 \approx \tfrac{2}{3} W_3 \quad \text{or} \quad W_3 \approx 0.75 W_2.$$

In such a case the 3-division strategy will achieve the same result as the 2-division strategy using 75% of the effort.

This observation rests heavily on the assumption. At least two objections are easy to spot. (1) The *uniform* 3-division strategy is *less adaptive* than the 2-division strategy. (2) The 3-division strategy will occasionally divide a subinterval in three parts when only a division in two is necessary.

As a remark to the second objection the opposite is of course true too: the bisection algorithm sometimes processes 5 or 7 intervals when it is only necessary to use trisection and process 4 intervals. Which value of p that finds the best balance between adaptability and low "waist" has been discussed in the literature and we summarize the experience.

Rule of Thumb (Hanke [6] and Sørevik [19]). When adaptability is important, then uniform 3-division is slightly better than bisection:

$$W_3 \approx 0.95\, W_2.$$

This result is theoretically justified and confirmed by numerical experiments. We should note that when adaptability is not important, e.g., a strongly oscillating function over the whole interval, then we can expect an even greater benefit from choosing trisection instead of bisection. We see two immediate remedies to the disappointing 5% improvement (compared to the indicated 25%).

(1) Choose a *nonuniform* 3-division based on available information. Such a strategy may even be *more adaptive* than bisection.

(2) Avoid dividing in three parts when only two is necessary by designing an *a priori error estimate* for a potential subdivision.

We could of course generalize this idea to any value of $p > 2$, however we prefer to simplify the discussion by concentrating on 3-division. From a practical point of view p should not be chosen too big. Given that we want to compute the subdivision points using only the available n function values, then we expect that, on the average, at least three function values per new subinterval is necessary information, giving $n/p \geqslant 3$. For $n \approx 20$ then values of p equal to 2, 3, 4 and 5 seem reasonable.

3. Nonuniform 3-division

Suppose that we have an interval H: $[a, b]$ (let us drop the subscript k) and given n function values $f_1, f_2, \ldots, f_n$. (All points $\in [a, b]$, e.g., Gauss–Kronrod points.) The problem we want to address is: how to use this information to find two subdivision points c and d such that $a < c < d < b$ giving new subintervals $[a, c]$, $[c, d]$, $[d, b]$.

There exist a large number of possible ways to compute c and d. We will, in this paper, discuss three, essentially different, approaches to this problem.

- Focus on *the difficulty*. Choose c and d such that the difficulty is *isolated*.
- Model, a priori, the local error of any potential subdivision and choose c and d in order to *minimize the sum of these three local error estimates*.
- Model, a priori, the future work associated with a potential subdivision and choose c and d in order to *minimize the total estimated work for this subdivision*.

The first of these three approaches is the simplest and is easy to implement. Furthermore, the extra cost associated with this strategy is negligible. One basic algorithm for this strategy looks as follows.

Algorithm focus on the difficulty.

```
• Find the location of the difficulty.
• Estimate the width of the difficulty.
• if the width > $(b - a)/3$ then
  Choose a uniform 3-division
  else
```

Use the c and d values suggested by the *location* and *width* with one exception: if $c = a$ or $d = b$, then accept either $[a, d]$ or $[c, b]$ as the difficult interval and *bisect the rest of the interval.*
endif

Note that we keep $p = 3$ always in this algorithm making no attempt to discover when a 2-division (possibly nonuniform) would have been sufficient or a value of $p > 3$ preferable. In order to do such a judgment we will need to have tools as indicated by the two alternative approaches to this problem. The implementation of this first strategy is discussed in the next section. Here we just mention that the location of a difficulty is spotted by a great (in absolute value) fourth-order divided difference. The width is then estimated by comparing the sizes of neighbor divided differences.

We will now briefly discuss the two alternative approaches to the problem to choose the division points c and d. Suppose that we have two trial values c and d. We need to model the error in absolute value if the integral of f over $[c, d]$ is approximated by the quadrature rule Q

$$E_{[c,d]} \approx K(d-c)^q \max_{[c,d]} |\mathrm{dd}|. \tag{1}$$

Here K is a constant, $q \geqslant 1$ is an integer and dd is a divided difference computed using available information. Note: this is meant to be an a priori estimate, before the *n new* function values are available, and will be based on function values used in connection with the interval H: $[a, b]$ itself. Formula (1) is a natural estimate of the error if the function is sufficiently smooth and the interval sufficiently small with proper choices of q and the order of the divided differences. However, due to lack of information we have to use a low-order divided difference, take the maximum in absolute value of all possible divided differences over the actual interval, choose (or compute) a value of q to be used, hoping that this model will give good choices of c and d. Suppose that trial values of c and d are available; then, based on (1), we get for the three new intervals, with the notation $D_{[c,d]} = \max_{[c,d]} |\mathrm{dd}|$ that

$$E_{[a,c]} \approx K(c-a)^q D_{[a,c]}, \tag{2}$$

$$E_{[c,d]} \approx K(d-c)^q D_{[c,d]}, \tag{3}$$

$$E_{[d,b]} \approx K(b-d)^q D_{[d,b]}. \tag{4}$$

The total new error for this trial subdivision will then be

$$E_{[a,b]}^{(\text{new})} \approx K\left\{(c-a)^q D_{[a,c]} + (d-c)^q D_{[c,d]} + (b-d)^q D_{[d,b]}\right\}. \tag{5}$$

Now, choose c and d in such a way that (5) is minimized. This may at first look like a simple task, but note that the maximum (e.g., fourth-order) divided difference in each of the three intervals may vary with c and d. We have tried several simplified implementations of this model without being able to compete with the simple focus on the difficulty strategy, taking into account the smaller effort in the latter.

The third approach, minimize the future work, is based upon the model for the error over each subinterval given in (1). In order to judge how much future work is needed, given a trial subdivision, we assume that this can be measured as follows. Given a tolerance e_{tol} and some trial interval (any of the intervals $[a, b]$, $[c, d]$ or $[d, b]$). A crude measure of the work left to be done in connection with this trial interval is how many equal parts p_{trial} we have to subdivide our trial interval into in order to get the error in each part E_{sub} less than $e_{\text{tol}}/p_{\text{trial}}$:

$$E_{\text{sub}} \approx \frac{E_{\text{trial}}}{p_{\text{trial}}^q} \leqslant \frac{e_{\text{tol}}}{p_{\text{trial}}},$$

giving

$$p_{\text{trial}} \geqslant \left(\frac{E_{\text{trial}}}{e_{\text{tol}}}\right)^{1/(q-1)}. \tag{6}$$

Thus we expect to have to process the trial interval itself in addition to at least p_{trial} (smallest possible integer satisfying (6)) subintervals giving us the work expressed in the number of intervals to be processed (l is here the length of the trial interval)

$$W_{\text{trial}} = \begin{cases} 0, & \text{if } l = 0, \\ 1, & \text{if } l > 0 \quad \text{and} \quad p_{\text{trial}} = 1, \\ 1 + p_{\text{trial}}, & \text{if } l > 0 \quad \text{and} \quad p_{\text{trial}} > 1. \end{cases}$$

Note that we are using this model only to determine the points c and d and do not intend to trust p_{trial} to guide us in the number of subintervals we are going to choose. Finally we compute the total work associated with a given subdivision

$$W_{\text{total}} = W_{[a,c]} + W_{[c,d]} + W_{[d,b]},$$

where each of these subintervals are treated as a trial interval and the work computed as indicated. Minimizing W_{total} by varying c and d will now give us the optimal subdivision. Unfortunately, this is not a simple optimization problem due to the same objection as mentioned for the second strategy. Several simplified versions of this strategy have been implemented too with the overall impression that it is hard to beat the simple focus on the difficulty strategy. It is reasonable to let the input value of ϵ influence e_{tol}, implying that we get different subdivision points if we change the value of the requested error.

4. Focus on the difficulty

This section is divided in two parts. In the first part we give a more detailed description on how we have implemented the simple focus on the difficulty strategy. Next we present the different test problems and we conclude this section with the results of the numerical experiments.

4.1. Implementation

One of the best adaptive, general purpose, codes available today is the QUADPACK's code DQAG. This code is based on Gauss–Kronrod rules with several options for the choice of n, the number of points. We have picked the 21-point version based on our experience that on problems where adaptability is important this is to be preferred [1].

As mentioned we have chosen a fourth-order divided difference to locate the difficulty. Thus we need to compute 17 inner products, each one consisting of 5 elements. The weights used in each inner product are precomputed and stored along with the Gauss–Kronrod weights. Thus the work amounts to 4 extra multiplications per function evaluation in the rule. This comes in addition to the work done in DQAG in connection with rule evaluation and error estimation.

Next we search for the index to the greatest divided difference (in absolute value). By comparing the sizes of neighbor divided differences to the maximum using heuristic values of the reduction factor, we compute the width of the difficulty. Based on this information we pick c and d in order to isolate the difficulty in the interval.

We do not believe that the choice of the fourth-order divided difference is crucial. In order to discover problem spots, e.g. in cubic spline functions, we need at least a third-order divided difference. The quadrature rule and error evaluation in DQAG takes place in the subroutine DQK21. We have modified this particular routine to evaluate, in addition to the fourth-order divided differences, the maximum of these and the index to this maximum, and finally the points c and d giving the *width* $= d - c$. After deciding if this actually is an isolation of a trouble spot with a possible modification of c and d, we return the values of c and d in addition to the rest of the parameters computed by DQK21.

These values are stored with the rest of the information connected to this interval. When the interval eventually is picked to be subdivided, then the subdivision points are used. In order to complete the documentation of these ideas we include, in the Appendix, the modified part of the DQK21 code.

4.2. Testing

The testing technique used is a performance profile testing technique suggested in [11]. These tests are based on a selection of six test families of integrands. Each test family has a special

Table 1

Test families	Attributes
1 $\int_0^1 (\lvert x-\lambda \rvert)^{\alpha_1}\,dx$	Singularity
2 $\int_0^1 f_2(x)\,dx$, where $f_2(x)=\begin{cases}0, & \text{if } x\leqslant\lambda\\ \exp(\alpha_2 x), & \text{otherwise}\end{cases}$	Discontinuous
3 $\int_0^1 \exp(-\alpha_3\lvert x-\lambda\rvert)\,dx$	C_0 function
4 $\int_1^2 10^{\alpha_4}/((x-\lambda)^2+10^{2\alpha_4})\,dx$	One peak
5 $\int_1^2 \sum_{i=1}^{4} 10^{\alpha_5}/((x-\lambda_i)^2+10^{2\alpha_5})\,dx$	Four peaks
6 $\int_0^1 2B(x-\lambda)\cos(B(x-\lambda)^2)\,dx$, where $B=10^{\alpha_6}/\max(\lambda^2,(1-\lambda)^2)$	Nonlinear oscillation

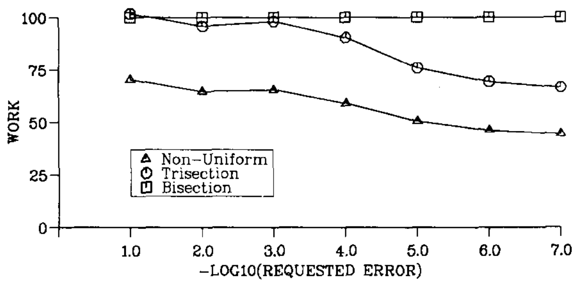

Fig. 2. Test family 1 (Singularity, $\alpha_1 = -0.5$ and $\lambda \in [0, 1]$).

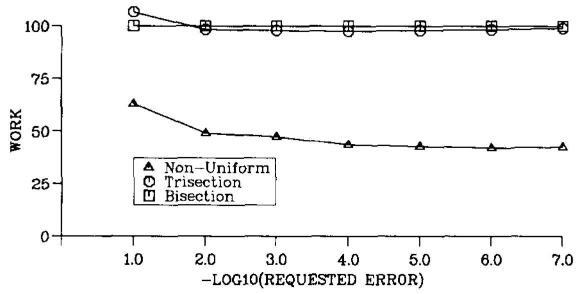

Fig. 3. Test family 2 (Discontinuous, $\alpha_2 = 0.5$ and $\lambda \in [0, 1]$).

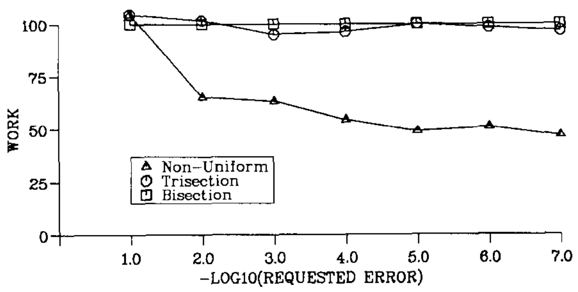

Fig. 4. Test family 3 (C_0 function, $\alpha_3 = -2$ and $\lambda \in [0, 1]$).

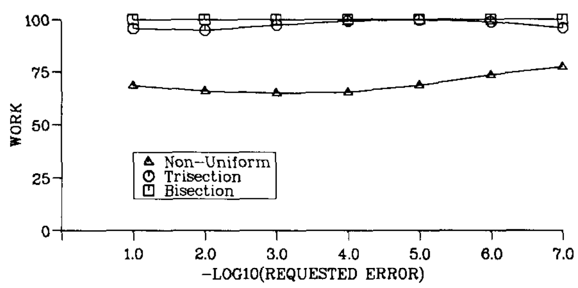

Fig. 5. Test family 4 (One peak, $\alpha_4 = -4$ and $\lambda \in [1, 2]$).

attribute, a difficulty parameter α and one or more random parameters λ (distributing the position of the difficulty uniformly over the integration interval) (see Table 1 and Figs. 2–7).

We have made one choice of value for each difficulty parameter and in addition we ran the codes on 50 sampled problems from each test family. We tested three versions of the DQAG code: (1) the original code DQAG which is based on bisection; (2) we replaced bisection with a simple trisection; (3) we replaced bisection with nonuniform 3-division based on the focus on the difficulty strategy.

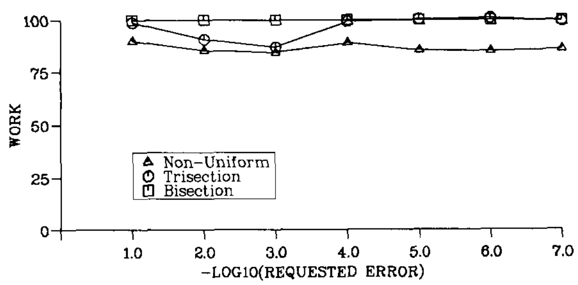

Fig. 6. Test family 5 (Four peaks, $\alpha_5 = -2$, $\lambda_i \in [1, 2]$, $i = 1, \ldots, 4$).

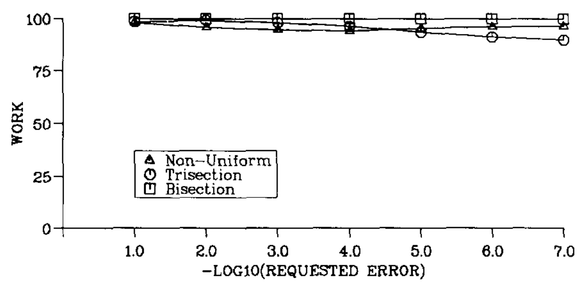

Fig. 7. Test family 6 (Nonlinear oscillation, $\alpha_6 = 3$ and $\lambda \in [0, 1]$).

We ran all codes on 7 different error requests: 10^{-j}, $j = 1,\ldots,7$. For each routine and error request we computed the average number of function evaluations based on 50 samples. We define the average number of function values for the DQAG code as a basic value for each error request, defined in each case to be 100%. Thus the results for the trisection version and the nonuniform version are given relative to this.

Note that the Rule of Thumb for uniform 3-division is confirmed by these experiments for all problems where adaptability is important with one exception: for singular problems it behaves far better than expected. We have at the moment no explanation to this behavior.

Furthermore, we see that nonuniform 3-division has results in the range 40–75% for all problems where adaptability is important (singularity, discontinuous, C_0 function and one peak). For the one peak problem family adaptability seems to become less important with smaller error requests. This is reasonable since the function is smooth and once the peak is isolated in a sufficiently small interval, it will not be more difficult than the rest of the intervals in the collection. This phenomenon is not observed in the three nonsmooth families since here the major job at any time will be to isolate the difficult area.

If we make the four peak family more difficult, we will improve the behavior of the nonuniform 3-division. Making this problem easier will move the results in direction of the results for the oscillating test family.

This last family is included to demonstrate the effect of using these codes to a problem with difficulties spread out over the whole integration area. Still we observe an improvement compared to DQAG, but in this case it is more moderate.

Only one place is DQAG better than its modifications: error request 10^{-1} for the C_0 function. This problem is very easy and in many of the sampled cases a division in 2 is sufficient. This fact makes it impossible for codes based on 3-division to compete with bisection.

We conclude this section by giving Table 2, which illustrates that the accuracy we have achieved with DQAG itself and the nonuniform 3-division do not differ much in spite of the reduction in the number of function evaluations we have achieved.

Table 2
The average number of correct digits with nonuniform 3-division minus the average number of correct digits with DQAG

Error request	Singularity	Discontinuous	C_0	One peak	Oscillatory	Four peaks
10^{-1}	0.02	0.37	1.21	2.11	0.33	1.16
10^{-2}	0.15	0.31	0.74	1.25	0.18	0.90
10^{-3}	0.08	0.50	1.40	0.43	0.06	0.40
10^{-4}	−0.18	0.34	0.86	−0.11	−0.01	0.49
10^{-5}	−0.44	0.21	0.66	−0.21	−0.02	0.65
10^{-6}	−0.57	−0.87	1.46	−0.32	−0.12	0.40
10^{-7}	−0.19	−2.84	1.02	−0.64	−0.20	0.26

5. Conclusions and remarks

We summarize our experience in the following remarks.

- A subdivision strategy based on a nonuniform p-division of intervals with $p \geqslant 2$ makes adaptive quadrature more efficient on problems where adaptability is important. We feel that the reduction in work is so great that future adaptive codes should include such a technique.
- A simple strategy of the type *focus on the difficulty* can be implemented with low cost and the tests show significant improvements in efficiency, without loss in reliability.
- Possible improvements of the simple strategy would be to (1) handle oscillating problems even better by trying to detect this situation and then increase p and (2) detect when p can be reduced in order to save work when this is possible.
- More sophisticated strategies, e.g., *minimize the error or minimize the future work* of a potential subdivision have a higher implementation cost than the simple strategy. We have, so far, been unable to design these strategies such that they become generally better than the simple strategy. The reason for this is probably that we have too little information available to implement these strategies in a proper manner.

Appendix

We give here the FORTRAN code for the tail of the modified QUADPACK routine DQK21. This is an implementation of the subdivision algorithm "Algorithm focus on the difficulty". c and d are new output parameters of the modified routine (to be saved along with the local interval $[a, b]$).

```
c     Non-uniform 3-division: Based on the 21 Gauss-Kronrod function
c     values we have computed the absolute value of the 17 divided
c     differences and stored these values in dd(i), i = 1, 2, ..., 17.
c     x(i), i = 1, 2, ..., 21, are the 21 Gauss-Kronrod points
c     on this local interval [a,b].
c
c     The problem is now to decide where to choose the two
c     division points: c and d.
c
c     Locate the maximum dd[1:17]; Save the address in index
c     and the value in divmax.
c
      divmax = dd(1)
      index = 1
      do 510 i = 2,17
        if (dd(i).gt.divmax) then
          divmax = dd(i)
          index  = i
        endif
510   continue
c
c     Find the size of the difficult region. This search is
c     based on the choice of the fourth order divided difference.
```

```
c     The region of influence is determined: it will be bounded by
c     the indices low and up. The value factor = 0.02 has been used
c     in the tests reported in this paper.
c
      low = 1
      compar = divmax*factor
      do 600 i = index-1,1,-1
        if (dd(i).le.compar) then
          low = i + 4
          go to 699
        endif
600    continue

699    continue
       up = 21
       do 700 i = index+1,17,1
         if (dd(i).le.compar) then
           up = i
           go to 799
         endif
700    continue
799    continue
c
c     Four, mutually exclusive, possibilities: 1 < low < up < 21;
c     up <= low; low = 1 < up <= 21; 1 < low < up = 21;
c     Now low=1 or up=21 suggests difficulty close to endpoint.
c     If both low>1 and up<21 then we expect an interior difficulty.
c
c     Identify the situation and choose the two division points.
c
      if ((low.gt.1).and.(up.lt.21).and.(low.lt.up)) then
c
c     In this case low and up give the division points c and d,
c     unless the width is too large: uniform 3-division.
c
          width = x(up) - x(low)
          if (width.le.((b-a)/3)) then
            c = x(low)
            d = x(up)
          else
            c = a + (b-a)/3
            d = b - (b-a)/3
          endif

      elseif (up.le.low) then
c
c     This is unlikely to happen. If it does, the "confusion"
c     is solved by expanding the region. Check for the ends
c     of the interval: Bisect the rest of the interval.
```

```
c
          up = up-1
          low =low+1
          if (up.eq.1) then
            c = x(low)
            d = b - (b-c)/2
          elseif(low.eq.21) then
            d = x(up)
            c = a + (d-a)/2
          else
            c = x(up)
            d = x(low)
          endif

      elseif(low.eq.1) then
c
c     Left endpoint difficulty: Check width and divide in three
c     either uniformly or by bisecting the rest of the interval.
c
          width = x(up) - a
          if (width.le.((b-a)/3)) then
            c = x(up)
            d = b -(b-c)/2
          else
            c = a + (b-a)/3
            d = b - (b-a)/3
          endif

      else
c
c     In this cases up = 21 and low > 1. Same procedure as
c     in the previous case, now with the right endpoint.
c
          width = b - x(low)
          if (width.le.((b-a)/3)) then
            d = x(low)
            c = a+(d-a)/2
          else
            c = a + (b-a)/3
            d = b - (b-a)/3
          endif
      endif
      return
      end
```

References

[1] J. Berntsen, A test of some well known quadrature routines, Reports in Informatics 20, Dept. Inf., Univ. Bergen, 1986.

[2] J. Berntsen and T.O. Espelid, Error estimation in automatic quadrature routines, *ACM Trans. Math. Software*, to appear.

[3] P.J. Davis and P. Rabinowitz, *Methods of Numerical Integration* (Academic Press, New York, 1984).

[4] C. de Boor, On writing an automatic integration algorithm, in: J.R. Rice, Ed., *Mathematical Software* (Academic Press, New York, 1971) 201-209.

[5] T.O. Espelid and T. Sørevik, A discussion of a new error estimate for adaptive quadrature, *BIT* **29** (1989) 283-294.

[6] W. Hanke, Die optimaler Sektion bei adaptiven Integrationsverfahren mit globaler Strategie, *Z. Angew. Math. Mech.* **62** (1982) T327-329.

[7] L.J. Hazlewood, An alternative strategy for cautious adaptive quadrature, *J. Inst. Math. Appl.* **20** (1977) 505-518.

[8] D.P. Laurie, Sharper error estimate in adaptive quadrature, *BIT* **23** (1983) 258-261.

[9] D.P. Laurie, Practical error estimation in numerical integration, *J. Comput. Appl. Math.* **12&13** (1985) 425-431.

[10] J.N. Lyness, Notes on the adaptive Simpson quadrature routine, *J. Assoc. Comput. Mach.* **16** (3) (1969) 483-495.

[11] J.N. Lyness and J.J. Kaganove, A technique for comparing automatic quadrature routines, *Comput. J.* **20** (1977) 170-177.

[12] M.A. Malcolm and R.B. Simpson, Local versus global strategies for adaptive quadrature, *ACM Trans. Math. Software* **1** (2) (1975) 129-146.

[13] W.M. McKeeman, Algorithm 145, adaptive numerical integration by Simpson's rule, *Comm. ACM* **5** (12) (1962) 604.

[14] W.M. McKeeman, Certification of algorithm 145, adaptive numerical integration by Simpson's rule, *Comm. ACM* **6** (4) (1963) 167-168.

[15] T.N.L. Patterson, The optimum addition of points to quadrature formulae, *Math. Comp.* **22** (1968) 847-856.

[16] R. Piessens, E. de Doncker-Kapenga, C.W. Überhuber and D.K. Kahaner, *QUADPACK, A Subroutine Package for Automatic Integration*, Ser. Comput. Math. **1** (Springer, Berlin, 1983).

[17] J.R. Rice, A metalgorithm for adaptive quadrature, *J. Assoc. Comput. Mach.* **22** (1975) 61-82.

[18] H.D. Shapiro, Increasing robustness, in global adaptive quadrature through interval selection heuristics, *ACM Trans. Math. Software* **10** (2) (1984) 117-139.

[19] T. Sørevik, Reliable and efficient algorithms for adaptive quadrature, Technical Report, D.Sc. Thesis, Dept. Inf., Univ. Bergen, 1988.