Vectorized Adaptive Quadrature in Matlab

L.F. Shampine

Mathematics Department, Southern Methodist University, Dallas, TX 75275, U.S.A.

Abstract

Adaptive quadrature codes process a collection of subintervals one at a time. We show how to process them all simultaneously and so exploit vectorization and the use of fast built-in functions and array operations that are so important to efficient computation in MATLAB. Using algebraic transformations we have made it just as easy for users to solve problems on infinite intervals and with moderate end point singularities as problems with finite intervals and smooth integrands. Piecewise-smooth integrands are handled effectively with breakpoints.

Key words: global adaptive quadrature, infinite interval, end point singularity, piecewise-smooth integrand
1991 MSC: 65D30

1 Introduction

We discuss here the algorithms of `quadva`, an effective program for approximating

$$I = \int_a^b f(x)\,dx \tag{1}$$

in the MATLAB [12] problem solving environment (PSE). Because MATLAB is an interpreted language, certain programming practices can reduce run times very substantially in favorable circumstances. One is to vectorize the evaluation of functions. Accordingly, the quadrature programs of MATLAB, including in particular the recommended program `quadl`, require users to do this. Like other adaptive codes, `quadl` processes a collection of subintervals of (a, b) one

Email address: `lshampin@smu.edu` (L.F. Shampine).

at a time so that vectorization helps only in evaluating $f(x)$ for the arguments that lie in a single subinterval. The principal novelty of `quadva` is that all subintervals are processed at the same time. We shall see that this reduces greatly the number of times that the function for evaluating $f(x)$ is called. With the number of vector function evaluations as our measure of work, we are led to reconsider basic algorithms and make decisions different from those of `quadl`. The number of values computed in each call affects the run time, but generally not very much. Besides, the overhead of the computation may be comparable to, or even bigger than, the cost of evaluating typical integrands. This observation leads us to another programming practice of utmost importance for efficient computation in the PSE: Using the built-in functions and array operations provides the efficiency of compiled code within a larger computation that is interpreted. We show how to exploit these facilities to code vectorized adaptive quadrature in a clear and efficient manner. Vectorization of adaptive quadrature and the use of array operations would also be helpful with a compiled language like Fortran 90, but we give our attention to MATLAB where the case is very strong.

The end points must satisfy $a < b$, but `quadva` allows $a = -\infty$ and/or $b = +\infty$. It uses an algebraic change of variable to reduce a problem on an infinite interval to a problem on a finite interval. The resulting integrand is generally singular at one or both end points. End point singularities occur as a result of these transformations, but they also occur directly. `quadva` deals with moderate end point singularities by complementing adaptive quadrature with algebraic transformations to weaken singularities. The techniques we use to deal with infinite intervals and end point singularities are not novel, but we have exploited them to provide a useful capability. Indeed, computing integrals involving infinite intervals and/or end point singularities with `quadva` is *exactly* like computing integrals with smooth integrands and finite intervals. `quadva` allows users to specify breakpoints in a convenient way so as to deal effectively with integrands that are only piecewise smooth and integrands with sharp peaks that occur at locations roughly known in advance. The transformations that are so convenient in dealing with infinite intervals and end point singularities complicate significantly our treatment of finite precision arithmetic and breakpoints.

2 User Interface

For the discussion that follows it will be helpful to state the form of a call to `quadva`:

```
[Ifx,errbnd] = quadva(f,interval,reltol,abstol);
```

Here `Ifx` is an approximation to the integral (1) and `errbnd` is an optional quantity that is an approximate bound on the error $|I - Ifx|$. `f` is the handle of a function that accepts a row vector `x` and returns a row vector `fx` of corresponding values of the integrand $f(x)$. `interval` is a row vector with entries that increase from $a =$ `interval(1)` to $b =$ `interval(end)`. Infinity is a predefined quantity in MATLAB called `Inf`. The end point a can be `-Inf` and b can be `Inf`. Ordinarily `interval` is just `[a,b]`. Additional entries are breakpoints that are discussed in §5.

`quadl` has only a pure absolute error tolerance with default value 10^{-6}. `quadva` has both relative and absolute error tolerances, `reltol` and `abstol`, with default values 10^{-5} and 10^{-10}, respectively. The code actually works with values `rtol` and `atol`. They are used to deal with improper specifications. For example, if `abstol` is negative, the code takes the internal absolute error tolerance `atol` to be 0. A computational distinction is that if `reltol` is positive, then `rtol = max(reltol,100*eps)`, where `eps` is the unit roundoff in MATLAB. This prevents the code from attempting an accuracy unreasonable in the precision available.

3 Vectorization

There are important quadrature methods that treat the whole interval of integration in a uniform way. Trefethen [17] implements high-order Gaussian formulas and Clenshaw-Curtis formulas in MATLAB and compares the methods both theoretically and numerically. Mori [13] surveys quadrature formulas obtained by variable transformations followed by application of the composite trapezoidal rule. These methods are especially effective when the integrands are smooth and the last kind copes well with end point singularities. Adaptive methods attempt to recognize portions of the interval where the integrand is difficult and work hard only where this appears to be necessary. Adaptive methods are in wide use, but whether they will perform better than the uniform schemes cited will depend on the problem. In this paper we consider how to improve the efficiency of the adaptive approach in MATLAB by exploiting vectorization.

We use some lines of code and a few details about data structures from `quadva` to explain in a concrete way how we vectorize global adaptive quadrature and use fast operations in the PSE to accelerate the computation. At a typical stage of the computation we have an accurate approximation `IfxOK` to the integral over a portion of $[a, b]$ and an estimate `errOK` of its error. The remaining portion of $[a, b]$ is a set of subintervals that we describe by an array `subs`. Specifically, each column of `subs` has two entries with the first being the left end point of a subinterval and the second being the right end point. The

quadrature formula is to be applied to each subinterval. The number of samples taken by the formula in each subinterval is called `samples` and in `quadva` it is 15. We form the arguments for each subinterval and assemble them as a row vector which is then passed to the function for evaluating the integrand. In this way, all samples are formed in a single call. Hitherto adaptive schemes have processed at most one subinterval at a time. How that subinterval is chosen distinguishes local and global schemes.

The values of the integrand are returned as a row vector `fx` that we reshape as an array with each column containing the samples for a single subinterval using a built-in function, `fx = reshape(fx,samples,[])`. The weights for the formula are held as a diagonal matrix `wt` and a row vector `halfh` holds half the lengths of the subintervals of `subs`. Approximate integrals are formed simultaneously for all the subintervals by `Ifxsubs = sum(wt*fx).*halfh` and estimates, `errsubs`, of the errors of these approximations are formed similarly with one line of code. In this we use fast array operations and the fact that when the built-in function `sum` is applied to a two-dimensional array, it adds the entries in columns to produce a row vector. When applied to a one-dimensional array, `sum` adds the entries, which is used to form an approximation to the integral over all of $[a, b]$, namely `Ifx = sum(Ifxsubs) + IfxOK`. An approximation to the error over the whole interval is formed similarly using `errOK`.

If the approximate integral computed in this way passes the error test, we are done. If it does not, we identify the intervals for which the approximations have an acceptable error. If we want to achieve an absolute error of τ in an integral over the interval $[a, b]$, it will suffice to achieve an absolute error of $\tau\,(\beta - \alpha)/(b - a)$ in all integrals over subintervals $[\alpha, \beta]$. This is more stringent than necessary [16, p. 187], but like de Boor [4], we prefer this conservative approach to the control of error because it enhances the reliability of the program. An important consequence is that as soon as we obtain a sufficiently accurate approximation, we can drop the subinterval from further consideration. This contrasts with global schemes like that of QAG [15] which consider throughout the computation all subintervals arising at any stage. The approach is convenient in the present circumstances because we can determine simultaneously which subintervals are acceptable. After defining `tol = max(atol,rtol*abs(Ifx))`, this is done efficiently by using the built-in function `find` to determine the indices `ndx` of `Ifxsubs` for which the errors are acceptable:

```
ndx = find( abs(errsubs) <= (2/(b-a))*halfh*tol );
```

The cumulative approximation `IfxOK` is then updated and these subintervals removed from the array `subs` of active subintervals with

```
IfxOK = IfxOK + sum(Ifxsubs(ndx));
subs(:,ndx) = [];
```

Evidently the dynamic storage and array operations of the PSE make it easy to maintain a list of active subintervals.

If a subinterval remains in the active list, we follow the usual practice of bisecting the subinterval. `quadl` proceeds in a different way. It is based on a 7 point Lobatto-Kronrod pair which samples at both ends of a subinterval. The code splits a subinterval into 6 pieces so that each sample is an end point of a new subinterval. Its recursive implementation then makes it convenient to reuse these samples when computing approximations on smaller subintervals. This is not generally thought to be worthwhile in scalar computation when using formulas of moderately high order, and we prefer higher order than the pair of `quadl`. Furthermore, in our approach of processing all active subintervals simultaneously, reuse of samples does not appear to be worth the overhead.

At all stages of the computation we have an approximate integral and an estimate of its error. Like `quadl`, we interrupt the computation when a subinterval is so short that the effects of finite precision arithmetic intrude. We also quit when halving the active subintervals would result in more than 650 subintervals. This is analogous to the error return from `quadl` when it has reached 10,000 evaluations of $f(x)$. In these situations we return from `quadva` with the current approximation and an estimate of its error and warn the user that the approximation does not satisfy the error test. As illustrated in §7, an approximation can be useful even when the desired error was not achieved provided that the code returns an assessment of its error.

4 Difficulties

In this section we discuss difficulties common to adaptive quadrature codes and devices for enhancing their performance.

4.1 Formulas

We compute two independent approximations $Q1$, $Q2$ to the integral over a subinterval and estimate the error in $Q1$ by $Q2 - Q1$. This works only if $Q2$ is more accurate than $Q1$ and it is then natural to accept $Q2$ and regard the estimate of the error in Q1 as being a conservative estimate of the error in Q2. If the integrand is smooth and $Q2$ is of much higher order than $Q1$, the estimate is very conservative, so it might be described better

as an approximate bound on the error in Q2. As we discuss more fully in §5, any scheme based on a finite set of samples and an error estimate of this kind can be fooled when the behavior of the integrand at these samples is not representative. Certain precautions are usual. `quadl` uses a 4-point Lobatto formula as the basic formula and a Kronrod extension for a total of 7 samples in the subinterval. Because vectorization makes samples relatively inexpensive, we prefer the accuracy and robustness of a pair developed for QUADPACK [15], namely a 7-point Gauss formula and a Kronrod extension for a total of 15 samples. These formulas have much higher degree of precision than those of `quadl`. The 15-point Kronrod extension samples the integrand at 8 additional points that interlace those of the 7-point Gauss formula. These independent samples add to the robustness of the error estimate. The samples are located at irrational points, lessening the chance of an unfortunate correlation of sample and integrand. If a formula with positive coefficients, like the ones we consider, has degree of precision d, it is easy to argue [16, p. 172] that its error is bounded by $2(b-a)$ times the error of the best possible approximation of the integrand by a polynomial of degree d. Even if the integrand is not very smooth, this argument and the much higher degree of precision of the Kronrod extension gives us reason to think that it will provide a more accurate result than the Gauss formula, hence a reliable assessment of the error. We compared `quadl` and `quadva` on the battery of test problems in [7, p. 334] and did parameter studies as in [5] and as in [7] for one family. The results we report here and others show that `quadva` is a reliable program, especially when compared to `quadl`. A number of authors have tried to enhance the efficiency and reliability of this simple approach to estimating error. We found that modifying `quadva` to use a scheme discussed in [3,15] was not straightforward, but it is quite possible that further research and development along these lines could improve the reliability of the program.

All the programs tested in [7] behaved in a special way on problem #21, so we take it out of the battery of test problems and discuss it in §5. Each of the remaining 22 problems was solved for pure absolute error tolerances $10^{-1}, 10^{-2}, \ldots, 10^{-12}$. A computation is said to have *failed* if the result is in error by more than the tolerance and to have failed *badly* if it is in error by more than ten times the tolerance. Table 1 reports the performance of `quadl`. The `quadva` program had no failures. It might be remarked that `quadl` gave repeated warnings of divide by 0, logarithm of 0, minimum step size, and maximum function count exceeded. An indication of the relative efficiency of the two programs is provided by a count of the total number of vector function evaluations made in solving all the problems at all the tolerances: `quadva` made 652 and `quadl`, 26,076.

De Boor [5] explores the behavior of CADRE when solving three families of problems involving a parameter. In these tests the pure absolute error tolerance is held fixed at 10^{-6} and the problem is solved for a range of parameter

Problem #	2	3	7	13	17	19	23
Failed (badly)	5 (0)	2 (0)	5 (3)	3 (1)	3 (1)	2 (0)	3 (1)

Table 1
For each problem of [7] (less #21), number of times error bigger than τ (10τ) when using `quadl` and $\tau = 10^{-1}, 10^{-2}, \ldots, 10^{-12}$. Number is 0 for problems not listed.

values. The first family has a peaked integrand

$$\int_{-1}^{1} \frac{2^{-\alpha}}{4^{-\alpha} + x^2}\, dx, \tag{2}$$

which we solved for 50 values of α equally spaced between 0 and 30. The second family has an algebraic end point singularity,

$$\int_{0}^{1} x^{\alpha}\, dx, \tag{3}$$

which we solved for 50 values of α equally spaced between -0.6 and $+1.6$. The third family has an oscillatory integrand,

$$\int_{0}^{1} 1 + \cos(\alpha\pi x)\, dx, \tag{4}$$

which we solved for 50 values of α equally spaced between $1/3$ and $83 + 1/3$. Measuring reliability as with the battery test, we found that `quadl` had no failures for the first family and none for the third family. There were 7 failures for the second family, but no bad failures. `quadva` had no failures for any of the families. In approximating three families of integrands, each for 50 parameter values, `quadva` made a total of 921 vector function evaluations and `quadl` made 15,284. Other aspects of computing these integrals are discussed in §7.

Espelid [7] does parameter testing in a somewhat different way that we illustrate with one of his six families of integrands, namely family #5, the integral of $\sum_{i=1}^{4} 10^{-2}/((x-\lambda_i)^2+10^{-4})$ from 1 to 2. What is distinctive about this kind of parameter testing is that the parameters λ_i are taken to be random points in the interval of integration. As with the battery of problems, for each set of randomly chosen parameters, the integral is computed with pure absolute error tolerances $10^{-1}, 10^{-2}, \ldots, 10^{-12}$. We did this for 1000 sets of parameter values using `quadl` and `quadva`. Out of the 12,000 integrals approximated, `quadl` failed 427 times and failed badly 215 times. `quadva` failed 41 times and had no bad failures. The total number of vector function evaluations made by

`quadl` was 3,931,248 and the number made by `quadva` was 53,544. In §7 we provide cpu times for this kind of comparison.

4.2 Singularities

General–purpose software can be quite useful when integrands have isolated singularities. Standard procedure is to split $[a, b]$ into subintervals with the singular points being included among the end points. A closely related task is computing integrals over infinite intervals. In the large collection of quadrature programs that form QUADPACK [15] are a code QAWS for integrands with algebraic and logarithmic singularities at end points and a code QAGI for infinite intervals. The former asks the user to identify the nature of the singularity and provide some details if algebraic singularities are present. Our goal was a capable quadrature code that is very easy to use, so we were unwilling to ask for information of this kind. `quadva` cannot solve as large a class of problems as codes that ask for more information from the user, but it can solve many problems on infinite intervals and deal with integrands that have moderate end point singularities.

Singular points provide a good argument in favor of open formulas. The Lobatto formula of `quadl` evaluates the integrand at both end points, but its predessor `adaptlob` makes no provision for end point singularities. That is why the examples of [8,9] give integrands an artificial value of zero at singular points. Via the error estimator, such a value can have the effect of directing the attention of the program to a difficult part of the interval. `quadl` traps values that are infinite and replaces them with values computed at points close to the end points. The approach may not avoid error messages arising from the evaluation of integrands at singular points, but it typically allows the program to compute an approximation to the integral. When they can be applied, adaptive quadrature codes are surprisingly effective at dealing with end point singularities.

We change variables to weaken the effects of singularities at finite end points. Specifically, if both end points are finite, we use an algebraic change of variables presented in [6, p. 441] that we discuss below in more detail. With this transformation we can solve a large class of problems with end point singularities without troubling the user in any way. If the interval is infinite, we use an algebraic transformation to obtain an equivalent integral on a finite interval. The resulting integrand may be singular at one or both end points. To illustrate what is done, suppose that the interval is $[0, +\infty)$. First we change variable to weaken any singularity that might be present at the finite end point: $\int_0^\infty f(x)\,dx = \int_0^\infty f(t^2)\,2t\,dt = \int_0^\infty g(t)\,dt$. If, say, $f(x) \sim c\,x^\alpha$ as $x \to 0$, then $g(t) \sim 2c\,t^{2\alpha+1}$ as $t \to 0$. In particular, if $\alpha \geq -1/2$, the new integrand

is finite at the left end point. This is useful for positive α as well as negative: The formulas are based on approximating the integrand by a polynomial. For this reason a function that behaves like $\sqrt{x}$ is difficult because its derivative is infinite at the origin, but the change of variables results in a function that behaves more like a polynomial. Naturally we should ask if improving the situation at the left end of the interval makes it worse at the right. For the integral to exist it is necessary that the integrand decay to zero as $x \to +\infty$ and it must decay pretty rapidly if typical numerical schemes are to succeed. Often the integrand behaves like

$$f(x) \sim c\, e^{-x} x^k \tag{5}$$

for some $k > 0$. For such an integrand, the change of variables actually increases the rate of decay. On the other hand, oscillatory integrands can be difficult and this is acute on infinite intervals. The change of variables might well make the difficulty worse by causing the integrand to oscillate more rapidly as $t \to +\infty$. The net effect of faster decay and more rapid oscillation will depend on the problem, but we advise users with oscillatory integrands and infinite intervals to use methods specific to the task rather than a general–purpose code like `quadva`. We follow weakening a potential singularity at the finite end point by reducing the infinite interval to a finite one with an algebraic transformation: $\int_0^\infty f(x)\,dx = \int_0^1 f(\phi(t))\,\phi'(t)\,dt = \int_0^1 g(t)\,dt$, where $\phi(t) = t/(1-t)$. In this we follow Kahaner [11, p. 140] who points out that if $f(x)$ behaves like (5) as $x \to \infty$, then $g(t)$ goes to zero rapidly as $t \to 1$. The singularity at the finite end point is only apparent for such integrands, but the computation is simpler and more robust when open formulas are used.

The QUADPACK program QAGI uses similar transformations to reduce a problem on an infinite interval to one on a finite interval, but the authors chose to make it distinct from the underlying program for finite intervals. Perhaps that was because the code was written in FORTRAN 66, but the same decision was made in the NAG library [14] Fortran 90 program `nag_quad_1d_inf_gen`, which is based on QAGI. We think it a great convenience to have just one code, so we allow the user to specify `-Inf` and `Inf` as end points. There is a built-in function for recognizing `Inf` that we use on entry to recognize infinite intervals and identify an appropriate change of variable. Similarly, QUADPACK uses transformations much like the ones we use to deal with singularities at finite points. However, they are implemented in a program for the task, QAWS, and require the user to provide some details about the nature of the singularities. The NAG library has an equivalent code based on QAWS that retains this design. Our goal was to provide a modest capability of integrating functions with singularities without putting users to any extra trouble. To assess the capabilities of `quadva` and `quadl`, we applied them to several examples from the NAG documentation. As in the documentation, all the examples were

solved with the default absolute error tolerance of `sqrt(eps)` and a relative error tolerance of `1e-4`. In order to apply `quadl`, we used analytical values for the integrals to convert this mixed test to equivalent absolute error tolerances.

The logarithmic singularities allowed in QAWS and the NAG equivalent are illustrated by $\int_0^1 \log(x)\cos(10\pi x)\,dx$. The default tolerances correspond to an absolute error tolerance of about 4.9×10^{-6}. Neither `quadva` nor `quadl` asks users for information about singularities. Nevertheless, `quadva` used only 2 vector function evaluations to compute a result in error by 4×10^{-8}. Although `quadl` warns of a "Log of 0", it returns a result in error by 4.0×10^{-5} at a cost of 51 vector function evaluations. We comment that this approximation is likely to be perfectly satisfactory in practice, but according to the criteria of §4.1, `quadl` failed, and it was not far from failing badly. QAWS also allows algebraic singularities, which is illustrated by $\int_0^1 \sin(10x)/\sqrt{x(1-x)}\,dx$. With an equivalent absolute error tolerance of 5.4×10^{-5}, `quadva` used 52 function evaluations to compute an approximation accurate to 10^{-13}. `quadl` used 70 function evaluations to return a warning about a division by 0 and a result accurate to 5.2×10^{-5}. The problem used to illustrate the NAG equivalent of QAGI is $\int_0^\infty (\sqrt{x}\,(x+1))^{-1}\,dx$. This problem is difficult because the integrand has an algebraic singularity at the origin and decays at only a modest rate as x tends to infinity. `quadl` cannot be applied to a problem on $[0, \infty)$. With the equivalent absolute error tolerance of 3×10^{-4}, `quadva` used 70 vector function evaluations to compute an approximation accurate to 4×10^{-16}.

5 Breakpoints

It is well-known that any adaptive quadrature scheme can be fooled if its samples of the integrand are not representative, c.f. [16, p. 186]. This issue is of particular concern when the integrand oscillates rapidly or has sharp peaks. In the case of highly oscillatory integrands, especially on infinite intervals, special methods should be used. Because vectorization makes samples relatively inexpensive, we make the program more robust by taking a relatively large number of samples in the initial approximation to the integral. Accordingly, the minimum number of vector function evaluations in `quadva` is one, but the program uses a minimum of 10 subintervals, hence takes a minimum of 150 samples.

A standard device for getting samples in regions of activity is to split $[a, b]$ into appropriate subintervals. We make this convenient by allowing `interval` to be an array with entries that increase from a to b. Piecewise-smooth integrands pose special difficulties for automatic quadrature routines. It is best to split the interval into subintervals on which the integrand is smooth, approximate the integrals over the subintervals, and sum the results. If this is

not done, open and closed formulas behave rather differently. An adaptive quadrature code will, in effect, locate the discontinuity. The difficulty with an open formula is that the discontinuity will eventually lie somewhere between nodes in adjacent subintervals. The error estimate on each subinterval is small because the samples are taken where the integrand is smooth, but the estimate is misleading because the location of the discontinuity is uncertain. A closed formula may have to resort to a small subinterval to deal with a discontinuity, but it will recognize a discontinuity and attempt to deal with it. On the other hand, if a user supplies the locations of discontinuities, he will also have to code evaluation of the integrand so that proper one-sided limits are computed at discontinuities else the advantages of the approach are lost. This is inconvenient and it complicates vectorization. With an open formula no special care is necessary in coding the integrand because the formula will not evaluate at the discontinuity. We favor open formulas and facilitate the treatment of piecewise-smooth integrands by making them easy to specify by means of an array `interval`. An example is found in §7. An example from [8,9], namely $f(x) = x/(\exp(x) - 1)$ on $[0, 1]$, shows that coding the integrand at a discontinuity should be done carefully when using a closed formula. To deal with the indeterminate form at $x = 0$, the authors give the function the value 0 there. With this value the integrand is discontinuous and using the default tolerance, `quadl` takes 26 vector evaluations to approximate the integral. If the integrand is given the proper limiting value of 1, `quadl` needs only 2 vector evaluations.

There are some complications in implementing breakpoints that arise in the changes of variable that deal with infinite intervals and weakening singularities at finite end points. The user provides a function for evaluating $f(x)$ and the solver works with a variable t in a standard finite interval. Whenever the integrand is to be evaluated for an array of arguments t, an algebraic transformation is made to get the corresponding array of arguments x and the user's function is invoked. In the case of both end points being finite, we have $-1 < t < 1$ and $x = 0.25(b-a)t(3-t^2)+0.5(b+a)$. The complication is that to start the integration, we must invert this relationship to get the breakpoints in the t variable. Some care is needed with each of the four cases, but this one is the most difficult. We can write the equation as $t^3 - 3t + \alpha = 0$ where $\alpha = 4(x - 0.5(b + a))/(b - a)$. Using the fact that $|\alpha| < 2$, a little analysis shows that Newton's method converges not only quadratically fast, but even monotonically from an initial guess of zero. With this guess the first iterate is $\alpha/3$, which we use to start the computation. It is convenient to compute all the breakpoints simultaneously, so inverting the transformation in this way is fast and of course, it is done only once. Once the interval has been mapped to a finite interval, we split the resulting subintervals in half until we have at least 10 subintervals for the initial approximation.

Espelid [7] applied a number of quadrature programs to a battery of problems

and found that they *all* had failures when solving problem #21, the integral of $\sum_{i=1}^{3} 1/\cosh(20^i(x-0.2i))$ from 0 to 1. We removed it from the results reported in §4.1 for this reason and discuss it here. For this problem `quadl` had 6 failures, 4 of which were bad, and `quadva` had 10 failures, 8 of which were bad. The integrand has moderate peaks at 0.2 and 0.4, and a very sharp peak at 0.6. In particular, the integrand has a local maximum of 1 at 0.6 and decreases to 0.5 at 0.6 ± 0.000165. Whether a program will "see" this peak depends very much on the details of its algorithm. Breakpoints provide a way to draw attention to a region of interest. Good practice is illustrated by the paper [2] on the evaluation of some cumulative distribution functions by numerical quadrature. Appreciating that the integrands that arise in this application typically have one (very) sharp peak, Amos first locates the peak numerically and estimates its width. With this information he can then approximate the integral reliably. If the peak occurs at c and is mostly contained in an interval of width w, the integral could be computed reliably with `quadva` by specifying breakpoints at $c - w/2$ and $c + w/2$. If `quadva` is given the interval $[0, 1]$ and asked to compute the integral of problem #21 to an absolute error tolerance of 10^{-12}, it returns a result in error by 4×10^{-4} after 7 vector function evaluations. This error corresponds approximately to the area under the sharp peak, so it appears that the program simply did not "notice" the peak. If we help the program by specifying `interval` as `[0,0.59,0.61,1]`, it again uses 7 function evaluations, but the result is accurate to 6×10^{-17}. Indeed, with this `interval`, `quadva` had *no* failures when approximating the integral of problem #21 in the battery test of §4.1. In this it made only 61 vector function evaluations in approximating the integral at all 12 tolerances.

6 Limiting Precision

We break off the computation if arguments become closer in a relative sense than 100 times the unit roundoff `eps`. This is not entirely straightforward because of the transformations: The test must be done after the variable t used in the formulas is transformed to the variable x given to the function $f(x)$ and there are four sets of transformations. Still, by using a built-in function and array operations, the check is easy and fast. A more troublesome matter is approach to singular points.

Authors using exponential transformations to weaken singularities like [13,18] recognize difficulties due to a finite number of digits and a limited exponent range and attempt to deal with them in FORTRAN. The difficulties we encounter are less severe because the transformations are algebraic and we have convenient tools in the PSE for handling these difficulties. We begin by considering what is done in `quadl` at the end points. The predecessor `adaptlob` of this code expects the user to define a value at end points where the integrand

is indeterminate or infinite. In the examples of [8,9] this value is taken to be zero and the same is true of #19 of [7]. `quadl` takes advantage of the way MATLAB handles the evaluation in these circumstances: The PSE displays messages like "Divide by zero" or "Log of zero" as warnings and continues the computation. A built-in function, `isfinite`, is used to test whether the value computed is infinite (`Inf`) or not-a-number (`NaN`). If, say, $f(a)$ is not finite, the code forms $f(a+\texttt{eps}*(b-a))$ instead. If this argument is far enough away from the singular point $x = a$ that the integrand can be evaluated, the user receives an unwelcome message, but the code can carry on to approximate the integral. However, these precautions may not be sufficient, as we illustrate with a family of integrals,

$$F(\alpha) = \int_{-1}^{1} \frac{x^3}{\sqrt{1-x^2}} \sin(\alpha x)\, dx \tag{6}$$

studied by Abramowitz [1]. When `quadl` was asked to approximate the integral with $\alpha = 1$ to an absolute error tolerance of 10^{-12}, it displayed a variety of warning messages and quit after 1,034 evaluations with an answer of `Inf`. (`quadva` required only *one* evaluation to produce a result with an error of 4×10^{-13}.)

The messages from `quadl` are annoying, so in `quadva` we allow messages only on the first evaluation. In this way the user is informed of mistakes in coding $f(x)$, but messages that might arise from approaching a singular point are suppressed. After every evaluation we test whether any of the function values is `Inf` or `NaN`. If so, we break off the computation. If this is the first evaluation, we do not have an approximate integral, so the code returns with an error message. On subsequent evaluations, we have an approximate integral and approximate error bound which are returned along with a warning. An example is provided in §7.

7 Illustrative Examples

I. Gladwell [10] asked students in a numerical methods class to study the evaluation of three integrals with the MATLAB programs `quad` and `quadl`. They provide a small, but illustrative, set of test problems that represent our experience with a good many test problems. Examples presented in other sections and supplementary examples presented here expand on aspects of the implementations. Unless stated otherwise, all the examples of this section were solved with a pure absolute error tolerance of 10^{-12}. ***quadva*** *and a program that uses it and* ***quadl*** *to compute the three integrals studied by Gladwell are available on the author's home page.* `quadva` makes use of nested functions.

Nested and anonymous functions were added to `Matlab` at version 7.0 (May 2004).

Before discussing performance, we illustrate the user interface by approximating $\int_0^\infty \exp(-x^2)(\log(x))^2\,dx$. Defining the integrand with an anonymous function and using default tolerances, the integral is computed and displayed by

```
f = @(x) exp(-x.^2).*log(x).^2;
Ifx = quadva(f,[0,Inf])
```

It is seen that a semi-infinite interval and a singularity at the finite end point do not complicate the use of `quadva` in any way. The integral is about 1.947522, so the default relative error tolerance of 10^{-5} and absolute error tolerance of 10^{-10} correspond to a relative error control. `quadva` used 3 vector evaluations of the integrand to get a result with a relative error of 1.5×10^{-7}.

The first of Gladwell's problems is

$$\int_0^8 e^{-3x} - \cos(5\pi x)\,dx \tag{7}$$

`quadva` used 3 vector evaluations of the integrand to compute an approximation with an error of 3×10^{-16}. `quadl` computed an approximation with an error of 2×10^{-16}, but it required a surprising 1,712 vector evaluations. To be clear about this, `quadl` returns the number of samples, so we put a counter in the function that evaluates the integrand to measure the number of vector function evaluations that we report throughout this paper.

We cannot explain the difference in performance of the two programs on (7), but we have seen it with all our test problems that involve oscillatory integrands. In particular, we have seen it with all five of the problems of this kind in the battery of test problems [7] that we discussed in §4.1. The most dramatic difference was seen with problem #13, which is $\int_{0.1}^1 \sin(100\pi x)/(\pi x)\,dx$. `quadva` used 4 evaluations to compute an answer with error 5×10^{-17}. `quadl` quit with the warning

```
Maximum function count exceeded; singularity likely.
```

after 2,012 evaluations and an approximate integral in error by 2×10^{-2}. In §4.1 we also discussed the performance of the programs on three families of problems involving a parameter. De Boor [5] presented results for these problems in the form of a plot of cost versus parameter. We follow him by showing in Figure 1 how the programs compare when solving the oscillatory family (4) with a pure absolute error tolerance of 10^{-6}. Both codes achieved

the specified accuracy over the range of α considered. Indeed, the *worst* error with `quadva` was 10^{-14} and with `quadl`, it was 4×10^{-9}. We computed the integrals for all 50 values of α ten times with each program and divided to find that on average, `quadva` required about $0.106s$ to compute all the integrals and `quadl`, about $1.00s$. The run times differ on other computers and when using other versions of `Matlab`, so the times we report can be considered only a very rough indication of relative cost.

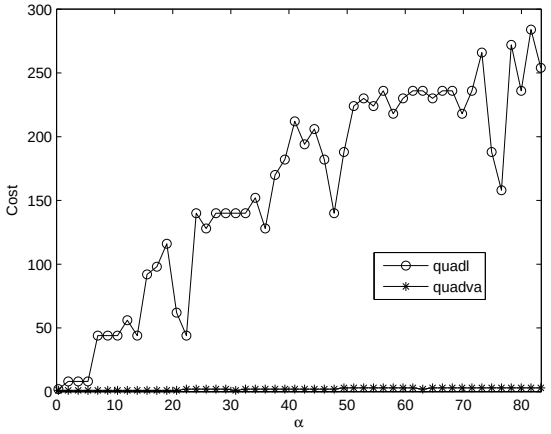

Fig. 1. Cost in vector function evaluations to compute (4) for a range of α.

Gladwell's set does not include a problem with a peaked integrand, so we again provide more details of computations reported in §4.1. The family (3) was used by de Boor to study peaked integrands. On average `quadva` took $0.349s$ to compute all 50 integrals and the worst error was 7×10^{-12}. `quadl` took on average $0.794s$ and the worst error was 5×10^{-9}. Family #5 of Espelid [7] was also discussed in §4.1. The computation involved approximating integrals for 1000 sets of randomly chosen parameter values. In order to compare the programs fairly, the random values were stored so that both programs could be applied to the same problems. For each set of parameter values, the integral was computed for 12 tolerances. The 12,000 integrations were accomplished by `quadva` in 19.46s. `quadl` took many more vector function evaluations, leading to a run time of 274.2s.

Gladwell's second problem is $\int_{-1}^{2} \left(|x - 1/\sqrt{3}\,| + |x - 1/\sqrt{2}\,| \right) dx$. This function is integrated exactly by `quadva` in 1 vector evaluation if it is told where the breakpoints are by calling it with `interval` equal to `[-1, 1/sqrt(3), 1/sqrt(2), 2]`. When called with `[-1, 2]`, it used 15 vector evaluations to compute an approximation with error 5.5×10^{-13}. `quadl` used 92 vector evaluations to compute an approximation with error 1.5×10^{-13}.

The last of Gladwell's problems is $\int_0^1 x^{-2/3}\, dx$. After 789 vector evaluations, `quadl` returns with the message

```
Warning: Minimum step size reached; singularity possible.
```

The approximate integral returned has an error of 10^{-6}, but no assessment of the accuracy is provided. `quadva` uses a transformation that weakens the singularity, but the resulting integrand is still singular. As a consequence, `quadva` returns an approximation after 16 vector evaluations and displays the warning

```
***Ifx does not satisfy error test.
   Approximate bound on error is 1.6e-005.
```

We describe this as a *soft failure* because the user is provided a useful approximation and assessment of its accuracy. We consider this assessment to be satisfactory because the true error of 6×10^{-7} is less than the approximate bound.

To illustrate a *hard failure* we tried to compute $\int_0^\infty \sin(x)/x\,dx$ with default tolerances. The decay of this integrand as x increases is too slow to compensate for its oscillations. With default tolerances, the program returns after 7 vector evaluations with a warning that the error test was not passed and an approximate error bound of 2.9. The true error is about 5.8, so the approximate bound on the error is not very good, but it is more than good enough to make clear that the approximation returned for the integral is useless. We remark that for this particular integral, integration by parts can be used to get equivalent integrals with integrands that decay fast enough to get good approximations with `quadva`.

8 Acknowledgements

This project has benefitted greatly from conversations with Professor I. Gladwell of Southern Methodist University. The author acknowledges with gratitude comments from anonymous referees that led to improvements in both the paper and program.

References

[1] M. Abramowitz, On the Practical Evaluation of Integrals, *J. SIAM* **2** (1954) 20–35.

[2] D. Amos, Evaluation of Some Cumulative Distribution Functions by Numerical Quadrature, *SIAM Review* **20** (1978) 778–800.

[3] J. Berntsen and T.O. Espelid, Error estimation in automatic quadrature routines, *ACM Trans. Math. Softw.* **17** (1991) 233–252.

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

[5] C. de Boor, CADRE: an Algorithm for Numerical Quadrature, in: J.R. Rice (Ed.), *Mathematical Software*, Academic, New York, 1971, pp. 417–449.

[6] P.J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, 2nd ed., Academic, Orlando, 1984.

[7] T. Espelid, Doubly Adaptive Quadrature Routines Based on Newton-Cotes Rules, *BIT* **43** (2003) 319-337.

[8] W. Gander and W. Gautschi, Adaptive quadrature–revisited, Research Rept. #83-03, Seminar für Angewandte Mathematik, ETH, Zürich, 1993.

[9] W. Gander and W. Gautschi, Adaptive Quadrature–Revisited, *BIT* **40** (2000) 84–101.

[10] I. Gladwell, Mathematics Department, Southern Methodist University, Dallas, TX, private communication.

[11] D. Kahaner, Sources of Information on Quadrature Software, in: W.R. Cowell (Ed.), *Sources and Development of Mathematical Software*, Prentice-Hall, Englewood Cliffs, NJ, 1984, pp. 134–164.

[12] MATLAB, The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760.

[13] M. Mori, Quadrature Formulas Obtained by Variable Transformation and the DE-Rule, *J. Comp. Appl. Math.* **12** (1985) 119–130.

[14] NAG Fortran 90 Library, Release 4, Numerical Algorithms Group Inc., Oxford, U.K., 2000.

[15] R. Piessens, E. de Doncker-Kapenga, C. Überhuber, and D. Kahaner, *QUADPACK: Subroutine Package for Automatic Integration*, Springer-Verlag, New York, 1983.

[16] L.F. Shampine, R.C. Allen, Jr., and S. Pruess, *Fundamentals of Numerical Computing*, Wiley, New York, 1997.

[17] L.N. Trefethen, Is Gauss Quadrature better than Clenshaw-Curtis? *SIAM Review*, to appear.

[18] H. Yserentant, A Remark on the Numerical Computation of Improper Integrals, *Computing* **30** (1983) 179–183.