Available online at www.sciencedirect.com

ScienceDirect

Applied Mathematics and Computation 202 (2008) 266–274

APPLIED MATHEMATICS AND COMPUTATION

www.elsevier.com/locate/amc

MATLAB program for quadrature in 2D

L.F. Shampine

Mathematics Department, Southern Methodist University, Dallas, TX 75275, United States

Abstract

We discuss here the algorithms of `TwoD`, a MATLAB program for approximating integrals over generalized rectangles and sectors. Capabilities of the language are exploited to make `TwoD` very easy to use. Vectorization is exploited by a novel algorithm to make the program efficient and reliable. `TwoD` provides for moderate singularities on boundaries of the region.

Keywords: Integral over plane region; MATLAB; Vectorization; Singularities; Adaptive quadrature

1. Introduction

We discuss here the algorithms of `TwoD`, an effective and capable program for approximating numerically integrals over regions in the plane. Some of the algorithms may be of interest in other computing environments, but the program was designed to exploit the MATLAB [1] problem solving environment (PSE). MATLAB itself has just one program for the task, `dblquad`, that approximates integrals over rectangles. `TwoD` approximates integrals over a much larger class of regions, specifically generalized rectangles

$$I = \int_a^b \int_{c(x)}^{d(x)} f(x,y)\,\mathrm{d}y\,\mathrm{d}x \tag{1}$$

and optionally, generalized sectors. We have taken advantage of capabilities in the MATLAB language to make `TwoD` very easy to use and in particular, to make it easy to specify regions. The program has other options not available in `dblquad`. By virtue of using open formulas, `TwoD` can be applied to integrands that are singular on a boundary. If it is known that an integrand is singular on a boundary, there is an option to use transformations that improve the performance of the solver in this situation. `TwoD` provides for both relative and absolute error control.

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 integrands. This is so important that all the quadrature programs of MATLAB *require* it. Recently some authors [2,3] have increased notably the

E-mail address: shampine@smu.edu

0096-3003/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.amc.2008.02.012

efficiency of quadrature in this PSE by exploiting vectorization more vigorously. The fact that `dblquad` requires vectorization with respect to x, but treats evaluation in y as a scalar computation suggests that further improvement in the use of vectorization is possible for quadrature in the plane. A large collection of numerical examples discussed in Section 4 confirms that `TwoD` approximates integrals with a much smaller number of vector function evaluations than `dblquad`.

A novel approach to error estimation and control is developed in Section 3. It exploits vectorization to obtain inexpensively information that supplements an estimate computed in the usual way with a pair of formulas. It is further used to estimate the error in the higher order formula of the pair, which is important to efficiency. Along with other algorithmic details, the approach has resulted in a reliable control of error. Indeed, results reported in Section 3 for two families of randomly generated problems show that `TwoD` is much more reliable than `dblquad`.

The `TwoD` program and a selection of numerical examples discussed in Section 4 are available at http://faculty.smu.edu/shampine/current.html.

2. The approach

With powerful programs readily available to approximate one-dimensional integrals, it is natural to write (1) as

$$\int_a^b g(x)\,\mathrm{d}x,$$

and each time the program needs a value $g(\xi)$, obtain it by approximating

$$g(\xi) = \int_{c(\xi)}^{d(\xi)} f(\xi,y)\,\mathrm{d}y.$$

There are a number of difficulties in this approach that are discussed in [4–7]. `dblquad` implements the approach with adaptive one-dimensional schemes. Lyness [6] argues that it is better to approximate $g(\xi)$ with a non-adaptive scheme. Perhaps that is why D01DAF [8] uses an iterative, non-adaptive scheme based on a collection of imbedded formulas due to Patterson. DITAMO [9] also uses an iterative, non-adaptive scheme, but it is based on the IMT-transformation followed by the trapezoidal rule on a sequence of meshes. This approach to approximating integrals of the form (1) is an effective one, but it is not clear how to exploit vectorization. Indeed, the `dblquad` program requires vectorization with respect to x, but treats evaluation in y as a scalar computation. We have taken a different approach to quadrature in the plane that allows us to exploit vectorization in a much more substantial way.

The other main approach to integration in the plane is to use a formula designed for a simple region such as a rectangle, triangle, or disk. Some programs provide for regions that are a collection of such regions and then refine as necessary to get a sufficiently accurate approximation. Others use a transformation to map a relatively general region to a standard one. An outstanding example [10] is Cubpack++, which allows a user to specify a region as a union of standard regions. Some of these standard regions are processed directly by the various quadrature schemes of the package and others are first reduced to primitive regions by transformation. For instance, a region like that of (1) is a generalized rectangle that is mapped internally to a rectangle. We have been less ambitious, but to explain what regions are allowed in `TwoD`, it will be convenient first to discuss some aspects of the software interface.

In developing `TwoD` we have drawn upon our experience [3] with a quadrature program for integrals in one-dimension. In collaboration with Hosea of The MathWorks, that program was improved, renamed `quadgk`, and added to MATLAB in the release R2007b.

2.1. Basic transformations

`TwoD` has a very simple user interface. If default values are used, an integral of the form (1) can be approximated with a call of the form

```
Q = TwoD(fun, a, b, c, d);
```

Here fun is a function that evaluates $f(x,y)$. It must accept arrays X and Y and return an array $Z = f(X,Y)$ of corresponding function values, i.e., for each (i,j), the entry $Z_{i,j} = f(X_{i,j}, Y_{i,j})$. All these arrays are 14×14. Generally it is not hard to code the evaluation in this way and certainly there is no difficulty with any of the large set of test problems that we consider in this paper. The efficiency of our approach to quadrature depends on vectorization of this function. Although we *strongly* discourage its use, we have provided an option 'Vectorized' that can be set false to tell the solver that fun has been coded only for scalar arguments. Internally TwoD applies this function to arrays by means of Z=arrayfun(fun,X,Y). TwoD is not intended for the integration of an $f(x,y)$ that is not vectorized, but if the integrand is not expensive to evaluate, the run time may be quite acceptable.

The default region is the generalized rectangle $a \leqslant x \leqslant b$, $c(x) \leqslant y \leqslant d(x)$ of (1). It is provided to the program in a natural way with the input parameters a,b,c,d. The parameters a and b must be constants. The parameters c and d can be either constants or functions. If a boundary is specified as a function of x, the function must be vectorized. Some regions are more naturally described in polar coordinates, so there is an option for a region that is a generalized sector. All options are specified as property–value pairs. When the 'Sector' option is set true, as in

```
Q = TwoD(fun, a, b, c, d, 'Sector', true);
```

the region is described using polar coordinates (r, θ). The input parameters then mean that the angular variable satisfies $0 \leqslant a \leqslant \theta \leqslant b$ and the radial variable, $c(\theta) \leqslant r \leqslant d(\theta)$. It is important to appreciate that polar coordinates are used only to describe the *region*—the integrand is $f(x,y)$ for both kinds of regions. It is convenient for the user to write f as a function of (x,y) no matter how the region is described, but, of course, polar coordinates are introduced inside the solver when the region is a sector. This is accomplished by means of the anonymous function

```
FUN = @(theta, r)fun(r.* cos(theta), r.* sin(theta)).* r;
```

We remark that MATLAB is case sensitive so that the function FUN is distinct from the input function fun. The change of variables must be implemented in a way that converts the arrays in polar coordinates used by the program to arrays in the Cartesian coordinates for which $f(x,y)$ is defined. This is accomplished efficiently in the function stated by using array operations and the fact that $\cos(\theta)$ and $\sin(\theta)$ are vectorized. By virtue of this transformation, TwoD works with a generalized rectangle for both kinds of regions allowed. The integral over a generalized rectangle is further transformed to an integral over a rectangle by changing the variable y (or θ) of the inner integral to $y = c(x) + t(d(x) - c(x))$ for $0 \leqslant t \leqslant 1$.

The documentation for dblquad suggests a "quick and dirty" way of dealing with regions that are not rectangular, namely to extend the integrand to a rectangle by giving it the value zero outside the region. This is convenient and may prove satisfactory, but it would generally be more reliable, efficient, and accurate to map the region to a rectangle. For example, if the region of

$$\int_1^3 \int_{\pi/6}^{x^2} 2x\cos(y)\,\mathrm{d}y\,\mathrm{d}x$$

is imbedded in the rectangle $[1,3] \times [\pi/6, 9]$ and dblquad is given the pure absolute error tolerance of 10^{-5}, it uses 1676 (vector) evaluations of the integrand to compute an approximation in error by 3.5×10^{-6}. After mapping the region to a rectangle as described above, the program uses 1404 evaluations to get a result accurate to 1.4×10^{-6}. (TwoD needs only five evaluations to compute an approximation accurate to 1.6×10^{-10}!)

2.2. Boundary singularities

The DITAMO program of Robinson and de Doncker [9] approximates integrals of the form (1) except that $c(x)$ must be constant and the end points may be infinite. For regions finite in one or both directions, they first apply a two-dimensional IMT-transformation that moves the boundary points to infinity and then use the trapezoidal rule on a sequence of meshes to approximate the integral. A byproduct of the transformation is that singularities in the integrand or boundary function are weakened greatly so that the program deals well with this complication. The approach is quite attractive, but it is not adaptive and the transformation can lead to difficulties with over- and underflow. We also provided for singularities at end points in `quadgk` [3], but the situation is somewhat different in the plane. One difference is noted in the documentation for the NAG program D01DAF [8]. Like `TwoD`, this program approximates integrals of the form (1) by a change of variables to get a rectangular region. The documentation points out that this transformation can induce singularities in the boundary functions and illustrates this with the task of integrating $x + y$ over the positive quadrant of the unit circle,

$$\int_0^1 \int_0^{\sqrt{1-y^2}} (x+y)\,\mathrm{d}y\,\mathrm{d}x.$$

The documentation points out that in this instance changing to polar coordinates avoids the difficulty,

$$\int_0^1 \int_0^{\pi/2} r^2(\cos(\theta)+\sin(\theta))\,\mathrm{d}\theta\,\mathrm{d}r.$$

`TwoD` allows a user to describe a region in polar coordinates when this is more natural, as it clearly is here, but does not require the user to transform the integrand. Some numerical results for these integrals are found in Section 4.

Another difference is that the polynomial transformation used in `quadgk` to weaken end point singularities lowers the degree of precision. Because the formulas used have degrees so high that this is not important, the transformation is always done for the convenience of users. For practical reasons discussed in other sections, we use formulas of much lower degree of precision in `TwoD`. In the circumstances we have made the use of this transformation optional. It is applied after the transformation(s) of Section 2.1 when `'Singular'` is given the value `true`. Each variable is transformed as suggested by Håvie [11, p. 441]. In one variable the transformation is

$$\int_{-1}^1 f(x)\,\mathrm{d}x = \int_0^\pi f(\cos(\phi))\sin(\phi)\,\mathrm{d}\phi = \int_0^\pi g(\phi)\,\mathrm{d}\phi.$$

A little calculation shows that if $f(x) \sim \beta(1-x)^\alpha$ as $x \to 1-$, then $g(\phi)) \sim \beta\phi^{1+2\alpha}/2^\alpha$. Evidently the new integrand is not singular at the corresponding end point if $\alpha \geqslant -1/2$. The same argument applies at the other end point. A similar argument says that if $f(x) \sim \beta\log(1-x)$ as $x \to 1-$, then $g(\phi)$ is not singular at the corresponding end point. This transformation is not nearly as powerful as the IMT-transformation, but it is quite helpful, does not present coding difficulties for users, and is relatively easy to implement in the present context of functions that are coded for array arguments.

3. Error estimation and control

In `TwoD` the default absolute error tolerance $ATOL = 10^{-5}$, but any $ATOL \geqslant 0$ can be specified using the option `'Abstol'`. The default relative error tolerance $RTOL = 0$, but any $0 \leqslant RTOL < 1$ can be specified using `'Reltol'`. `TwoD` attempts to compute an approximation Q with an approximate bound on the error E such that

$$|E| \leqslant \max TOL * \ |Q|) = TOL.$$

A standard device to enhance reliability is to insist that the approximate bound on the error be no more than a fraction of TOL. After some experimentation, we took this fraction to be 1/8. Because a pure absolute error control is permitted, it is possible that a given TOL is too small for the precision of the machine. To deal with this, we increase TOL as necessary so that we do not ask for an error smaller than 100 units of roundoff in Q. When initializing the computation we specify an internal tolerance of 100 units of roundoff in Q so as to force a refinement. Some of the standard test problems discussed in Section 4 are so easy for `TwoD` that this accuracy is achieved on some of the subrectangles in the initial approximation and in a few cases, all of them.

Key to the efficiency of `TwoD` is vectorization. In the Matlab PSE, the cost of evaluating an integrand that vectorizes well depends weakly on the number of samples. Of course it does depend on the number of samples, so we would prefer not to waste too many samples when the integral over a primitive region is not sufficiently accurate. The primitive region of `TwoD` is a rectangle. We proceed in a novel way. Each evaluation of the integrand provides enough samples to approximate the integral over several subregions and estimate the error of each. The more samples used in each subregion, the higher the degree of precision of the formula over the subregion, but the more samples are wasted if an integral is not sufficiently accurate. In a discussion of product formulas and minimal point formulas, Stroud and Secrest [12] say "As a rule of thumb, the Gaussian product formulas are to be recommended for their accuracy and ease of use in relatively low-dimensions whenever they can be applied." This observation is very pertinent to our circumstances, so we use a tensor product quadrature formula on each subrectangle. We considered a number of possibilities and decided to use a pair due to Kronrod [13] which consists of the three point Gaussian formula of degree of precision five embedded in a seven point formula of degree of precision 11. The tensor product formula provides a good basic approximation and a companion of considerably higher degree of precision. We also decided to apply the tensor product formula to four subrectangles at a time. This means that we evaluate the integrand at an array of $14^2 = 196$ points in each call. The points are specified by means of an array `X` of arguments in the x variable and an array `Y` of arguments in the y variable. Of course, both arrays are 14×14. They are input to the function for evaluating the integrand, which is to compute efficiently the integrand at all these points and return a 14×14 array `Z` of corresponding values.

The approximation on a subrectangle is accepted when an estimate of its error is smaller in absolute value than $TOL/8$ times the ratio of the area of the rectangle to the area of the initial rectangle. This plays two roles. If true for all subrectangles, the approximation resulting from summing approximations on all subrectangles has an error bounded by $TOL/8$, which is comfortably smaller than the accuracy requested. This is a conservative approach to error control because it ignores the fact that if some of the errors on subrectangles have different signs, the error in the sum will be reduced. The approach is convenient for the implementation of an adaptive scheme because once an approximate integral over a rectangle passes a local error test, that rectangle need be considered no further.

In `TwoD` we have a running sum of approximate integrals over rectangles that pass the local error test. When needed, a fast built-in function is used to sum the approximate integrals over rectangles that are not sufficiently accurate and the two are added to get an approximation Q to the integral over the initial rectangle. Correspondingly, there is a running sum of absolute values of errors that pass the local error test. As needed, the absolute values of errors that do not pass the test are summed efficiently and the two portions added to get an approximate bound on the error of Q. If this bound does not show that Q is sufficiently accurate, the estimated errors on all subrectangles remaining to be processed are inspected and information about a rectangle with largest error is removed from the list so that the approximate integral over this rectangle can be improved by refinement. Finding the maximum error and processing the list are handled easily and efficiently with built-in functions. In this way we take samples where we believe they will do the most good and we minimize the amount of information to be retained.

With the embedded formulas of Kronrod used in tensor product form, we compute two approximations over each rectangle. One that we call here Q_5 is obtained with a formula of degree of precision 5 and the other, Q_{11} from a formula of degree of precision 11. Conventionally the error in Q_5 is estimated by comparison to the formula of higher degree, hence the estimate $e = Q_{11} - Q_5$. For this estimate to be valid, it must be the case that Q_{11} is more accurate than Q_5, so many codes use Q_{11} as the approximate integral and regard e as a conservative estimate of its error. We do this, too, but we also use a novel scheme to estimate the error in the higher order formula.

If an approximation over a rectangle is not sufficiently accurate, we split the region into four subrectangles. In a *single* vector evaluation of the integrand, we compute independently approximations of two orders on all four subrectangles. The errors in the lower order approximations are estimated by comparison. A more accurate approximation to the integral over the whole rectangle is obtained by adding the higher order approximations over the subrectangles. With this improved higher order approximation, we can estimate the error in Q_{11} by comparison. What we would really like to have is an estimate of the errors of the approximations computed with this formula on each of the four subrectangles. Most of the effective quadrature codes make use of heuristics in their algorithms for error estimation and control and here is where we do something of the sort in `TwoD`. We expect that the absolute value of the ratio of the error in Q_{11} to the error in Q_5 will be the same when the formula pair is applied to a subrectangle. With an estimate of this ratio on the whole rectangle, we use the estimates of the errors of the lower order formula on the subrectangles to estimate the errors of the higher order approximations there. It might seem natural to estimate these errors by extrapolation of the higher order formula, but we think that it is more reliable to consider only the relative behavior of the two formulas than to assume that the integrand is smooth enough that the higher order formula has its expected order. To improve reliability, we estimate the error in the higher order formula on subrectangles only when the estimated error in Q_{11} is smaller than the estimated error in Q_5. In this scheme we lose the sign of the estimated error of the higher order formula, but we would use the absolute value anyway because it leads to a more conservative assessment of the error as noted earlier. In our testing this scheme has improved markedly the performance of the program.

Because they played such an important role in our development of a reliable program, we discuss in this section the performance of `TwoD` on two families of random problems considered in [14]. We begin with the oscillatory family,

$$\int_0^1 \int_0^1 \cos(2\pi\xi_1 + \tau_1 x_1 + \tau_2 x_2)\,\mathrm{d}x_2\,\mathrm{d}x_1. \tag{2}$$

Each problem is constructed as follows: First ξ_1 is picked randomly from $[0,1]$. The parameters τ_1, τ_2 are picked randomly from $[0,1]$ and then scaled so that $\tau_1 + \tau_2 = 15$. Our test program allows the number of samples to be specified. It then constructs and stores this many sets of parameters and corresponding analytical values of the integrals. Proceeding in this way, we can apply `TwoD` and `dblquad` to the same problems. Each set of problems is solved for each of five pure absolute error tolerances, namely 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}. A quadrature program is said to have "failed" if the error is bigger than the tolerance. It is said to have failed "badly" if the error is bigger than 10 times the tolerance and "very badly" if it is bigger than 100 times the tolerance. Both quadrature programs proved reliable for this family. In a typical run `TwoD` has no failures at any of the five tolerances and `dblquad` has at most a couple. For instance, in one run of 100 samples, `dblquad` failed once at tolerance 10^{-2}, but it did not fail badly.

The family with product peaks,

$$\int_0^1 \int_0^1 \prod_{i=1}^{2} (\tau_1^{-2} + (x_i - \xi_i)^2)^{-1}\,\mathrm{d}x_2\,\mathrm{d}x_1 \tag{3}$$

is much more demanding. Members are constructed as described for the oscillatory family with both ξ_1, ξ_2 taken at random from $[0,1]$. For this family the other parameters are scaled so that $\tau_1 + \tau_2 = 200/2^{1.5}$. The integrals of this family might well be as big as 10^4, so they were computed with the tolerances used for the other family, but the tolerances are interpreted as a requirement on the relative error. `dblquad` allows only a pure absolute error tolerance, so we used the analytical values of the integrals to deduce an equivalent absolute error tolerance. `TwoD` provides for control of the error relative to the computed approximate integral, so we used this capability in the tests. These integrals are so hard to compute reliably that we used the family to try out all our ideas for enhancing the program's performance in this regard. As might be expected, we had to give up some efficiency on easy problems to get reliability on hard problems. The results of a run reported in Table 1 is typical as to the relative behavior of the two solvers, but the numbers can vary substantially from one run to another. Considering these results and those of many other runs, we believe that `TwoD` is vastly more reliable than `dblquad` for this family.

4. Illustrative examples

The paper [9] that presents the DITAMO program contains a large collection of examples that we discuss here. We have also collected a good many examples from well-known papers and books. Included are two

Table 1
Sample of 100 random problems at each tolerance

Tolerance	`1e-1`	`1e-2`	`1e-3`	`1e-4`	`1e-5`
`dblquad`	90	90 (46*)	77 (33*) (7**)	76 (18*) (4**)	60 (8*) (3**)
`TwoD`	5	3	1	2	1 (1*)

An entry "60 (8*) (3**)" means 60 failures of which 8 are bad and 3 are very bad.

from the QUADPACK documentation [7], two from the D01DAF documentation [8], three from Davis and Rabinowitz [11], three from a paper of Rabinowitz and Richter as stated in [11, p. 373], three from Stroud and Secrest [12], and four from Kahaner and Rechard [16]. Here we use a few of the examples to make points about `TwoD` and when possible, compare its performance to `dblquad`. The `dblquad` program requires that the function for evaluating the integrand be coded to accept vector arguments. Because `TwoD` requires that the function accept array arguments, the same function can be used for both programs. That has been done with all examples to which `dblquad` can be applied directly. *Unless stated to the contrary, all examples in this section use a pure absolute error tolerance of* 10^{-5}.

In accordance with the design of DITAMO, all the examples of [9] have constant a, b, c in the form (1) of a generalized rectangle. However, only examples #10 and #25 have constant $d(x)$, hence regions that are rectangles. The first sixteen examples are described as problems with smooth integrands and boundary functions. Solution of these sixteen examples with `TwoD` was very satisfactory: The number of vector evaluations of the integrand ranged from 1 to 5 for these examples and the *worst* error was only 1.9×10^{-13}! Such accurate results obtained with so few function evaluations simply reflect the large number of samples in each evaluation, a moderately high degree of precision, and a conservative approach to error control. Only one of these examples is over a rectangle so that `dblquad` can be applied directly. It computes an approximation that is exact using 56 evaluations of the integrand. (`TwoD` uses 1 evaluation to compute an approximation with an error of 5.6×10^{-17}, which is at the level of roundoff.) Examples #17–22 are described as integrals with singular derivatives in the integrand or the boundary function and examples #23–27, as integrals with singular integrands. Before discussing the computation of these integrals, we note that we used Maple [15] to correct the published analytical values for #22 to $-\pi/8$ and #21 to

$$\frac{100}{27\sqrt{\pi}}\, 128^{1/10}\, 3^{9/10} \sin\left(\frac{\pi}{10}\right)\Gamma\left(\frac{6}{10}\right)\Gamma\left(\frac{9}{10}\right).$$

Although `TwoD` can integrate most of these singular examples without giving any special attention to singularities on the boundaries, the results we report here were all computed with the option `'Singular'` set to `true`. Except for examples #17, 21, 22, all the singular examples were solved with 5 function evaluations and the worst error was only 6.8×10^{-13}. The exceptions were not handled as well, but the program still performed in a very satisfactory way: Examples #17 and #22 were each solved with 5 function evaluations and the errors were 2.0×10^{-8} and 7.8×10^{-7}, respectively. Example #21 is

$$I = \int_0^2 \int_0^{d(x)} (xy)^{-0.1}\, \mathrm{d}y\, \mathrm{d}x$$

with

$$d(x) = 3\left(1 - \left(\frac{x}{2}\right)^{3/2}\right)^{2/3}.$$

Though approximating this integral is much more expensive for `TwoD` than the other examples, only 54 function evaluations were needed to compute an approximation accurate to 1.6×10^{-6}. This is quite a good performance. Lest the reader be misled by the very good performance of `TwoD` on the singular examples #17–27, we recall from Section 2.2 that the transformations used by `TwoD` to weaken singularities cannot deal with singularities as strong as those allowed by DITAMO. As it happens, the singularities of all these examples are of types shown in Section 2.2 to be converted to non-singular problems by the scheme of `TwoD`. There is no suggestion in the `dblquad` documentation that it can deal effectively with integrals like #17–27 and only example #25 is over a rectangle so that `dblquad` can be applied directly. When we tried the program on this example, it reported three times the

```
Warning : Infinite or Not-a-Number function value encountered.
```

After 2830 evaluations, the program returned `NaN` as answer. To be fair, we note that `TwoD` is not able to compute this integral without the assistance of the `'Singular'` option, i.e., without the transformation to weaken the singularity. Indeed, it quits with the error message

```
Maximum number of subintervals : Q does NOT pass error test.
```

In Section 2.2 we discussed an example from the documentation [8] for the D01DAF program. When the region is treated as a generalized rectangle with a degenerate side, there is a singularity in the boundary function. `TwodD` used 27 evaluations of the integrand to compute a result in error by 1.9×10^{-6}. This is quite satisfactory, but when the `'Singular'` option was set `true`, only 5 evaluations gave a result in error by 9.7×10^{-14}. The region is a generalized sector, so it is more natural to take advantage of the possibility in `TwoD` of describing the region in polar coordinates by setting the option `'Sector'` to `true`. With this integrand, `TwoD` is then accurate to roundoff level, 1.1×10^{-16}, after two evaluations. If the suggestion of the NAG documentation is adopted, namely introducing polar coordinates by hand to get a rectangular region, the `dblquad` program can be applied. It used 56 evaluations to get a result in error by 2.5×10^{-8}.

Stroud and Secrest [12, p. 50] provide numerical results for three examples that arise as integrals over the surface of a three-dimensional sphere, but are converted to integrals over a rectangle by a change of variables. Using Maple [15] the analytical result stated for I_7 was corrected to $(\pi^5 - \pi^3)/3 - 8\pi$. `TwoD` did well on all these examples and we provide details for one because `dblquad` was surprisingly inefficient. This example is

$$I_9 = \int_{-\pi/2}^{\pi/2} \int_{-\pi}^{\pi} y^2 \sin^2(y + x) \cos(x)\, dy\, dx = \frac{2\pi^3}{3} - \frac{\pi}{3}$$

with five evaluations of the integrand, `TwoD` computed an approximation with error 1.3×10^{-9}. `dblquad` used 1574 evaluations to obtain a result in error by 1.1×10^{-7}.

The three examples we have taken from Davis and Rabinowitz [11] all present difficulties. One is to integrate $1/(1 - xy)$ over $[0, 1] \times [0, 1]$. It is the singularity at $x = y = 1$ that causes trouble. When `dblquad` is applied to this problem, it reports

```
Warning: Infinite or Not-a-Number function value encountered
```

and

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

Using 2328 evaluations of the integrand it computed an approximate integral in error by 1.2×10^{-4}. `TwoD` can be applied to integrands with singularities on the boundary because it uses open formulas. With 16 evaluations it produced an approximation in error by 6.1×10^{-7}. When the option `'Singular'` was set `true`, it used 10 evaluations to get a result in error by 1.5×10^{-7}. An illuminating example is to integrate $\sqrt{|x - y|}$ over $[0, 1] \times [0, 1]$. Note the infinite derivatives along the diagonal $x = y$. Using 194 evaluations, `dblquad` computed an approximation in error by 8.6×10^{-5}. `TwoD` did not do nearly so well. Indeed, this is the *only* example we have encountered for which this is the case. `TwoD` reported

```
Warning: Maximum number of subintervals: Q appears to pass error test.
```

The approximation `Q` *does* pass the error test for it is in error by 2.0×10^{-6}. However, locating and dealing with the infinite derivatives was expensive, costing 749 evaluations of the integrand. The performance of `TwoD` can be improved dramatically by splitting the region into two parts along $x = y$. As it happens, the integral is twice the integral over one part, a fact we used, but this is unimportant. What is important is to have the singular behavior on the boundary instead of the interior of the region. This alone reduced the cost to 10 evaluations and yielded an error of 3.0×10^{-6}. It is even better to set `'Singular'` to `true`. This reduced the cost to five evaluations and the error to 9.3×10^{-15}!

In Section 3 we considered two families of problems with randomly selected parameters. Here we report the costs of the runs described in that section. For the family (2), `TwoD` used 1992 evaluations of the integrand to compute the 100 integrals at five tolerances, which took 1.15 s, and `dblquad` used 56,560 evaluations, which took 4.34 s. For the family (3), `TwoD` used 28,241 evaluations to compute the 100 integrals at five tolerances, which took 12.3 s, and `dblquad` used 366,734 evaluations, which took 25.0 s. Of course run times differ from one run to another and more substantially from one computer to another, but `TwoD` appears to be notably more efficient than `dblquad`.

References

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

[2] E. Sermuthu, H.T. Eyyuboğlu, A new quadrature routine for improper and oscillatory integrals, Appl. Math. Comput. 189 (2007) 452–461.

[3] L.F. Shampine, Vectorized adaptive quadrature in MATLAB, J. Comput. Appl. Math. 211 (2008) 131–140.
[4] F.N. Fritsch, D.K. Kahaner, J.N. Lyness, Double integration using one-dimensional adaptive quadrature routines: a software interface problem, ACM Trans. Math. Softw. 7 (1981) 46–75.
[5] 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.
[6] J. Lyness, When not to use an automatic quadrature routine, SIAM Rev. 25 (1983) 63–87.
[7] R. Piessens, E. de Doncker-Kapenga, C. Überhuber, D. Kahaner, QUADPACK: Subroutine Package for Automatic Integration, Springer-Verlag, New York, 1983.
[8] NAG Fortran 90 Library, Release 4, Numerical Algorithms Group Inc., Oxford, UK.
[9] I. Robinson, E. de Doncker, Algorithm 45 automatic computation of improper integrals over a bounded or unbounded planar region, Computing 27 (1981) 253–284.
[10] R. Cools, D. Laurie, L. Pluym, Algorithm 764: Cubpack++: a C++ package for automatic two-dimensional cubature, ACM Trans. Math. Softw. 23 (1997) 1–15.
[11] P.J. Davis, P. Rabinowitz, Methods of Numerical Integration, second ed., Academic, Orlando, 1984.
[12] A.H. Stroud, D. Secrest, Gaussian Quadrature Formulas, Prentice-Hall, Englewood Cliffs, NJ, 1966.
[13] A. Kronrod, Nodes and Weights of Quadrature Formulas (Translation), Consultants Bureau, New York, 1965.
[14] J. Berntsen, T.O. Espelid, A. Genz, An adaptive algorithm for the approximate calculation of multiple integrals, ACM Trans. Math. Softw. 17 (1991) 437–451.
[15] Maple 11, Maplesoft, 615 Kumpf Dr., Waterloo, CA.
[16] D.K. Kahaner, O.W. Rechard, TWODQD an adaptive routine for two-dimensional integration, J. Comput. Appl. Math. 17 (1987) 215–234.