Available online at www.sciencedirect.com

ScienceDirect

Applied Mathematics and Computation 189 (2007) 452–461

APPLIED MATHEMATICS AND COMPUTATION

www.elsevier.com/locate/amc

A new quadrature routine for improper and oscillatory integrals

E. Sermutlu [a,*], H.T. Eyyuboğlu [b]

[a] Department of Mathematics and Computer Sciences, Çankaya University, 06530 Balgat, Ankara, Turkey
[b] Department of Electronic and Communication Engineering, Çankaya University, 06530 Balgat, Ankara, Turkey

Abstract

In MATLAB environment, a new quadrature routine based on Gaussian quadrature rule has been developed. Its performance is evaluated for improper integrals, rapidly oscillating functions and other types of functions requiring a large number of evaluations. This performance is compared against the other quadrature routines written for MATLAB in terms of capability, accuracy and computation time. It is found that our routine rates quite favourably.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Quadrature; Improper integrals; Numerical algorithms; MATLAB

1. Introduction

After years of development, numerical quadrature remains to be a problem. The existing routines can handle very difficult questions whilst giving erroneous results for some simple ones. Actually, "easy" and "difficult" have completely different meanings for symbolic and numeric integration. For example the integrals $\int_0^\pi \sin(nx)\mathrm{d}x$ and $\int_0^{10} x^n \mathrm{d}x$ are among the simplest symbolic integrals, yet, for sufficiently large n, they will make most numerical algorithms fail miserably.

So research on numerical quadrature continues. On the one hand, new rules are being developed [7] and on the other, well-known rules are being re-examined and successfully implemented [2,5]. In particular, El-Mikkawy introduced a new approach for the calculation of Cotes numbers for Newton–Cotes formula [5].

Main quadrature program of MATLAB currently employed is `quadl` [4] which is a great improvement over the previous program `quad`. It uses a four point adaptive Gauss–Lobatto rule. Maximum number of function evaluations is 10.000. The program displaces the endpoints (as small as possible, that is, at machine precision) so that it can handle some improper integrals with singularity occurring at integral limits. But sometimes it will give wrong results or return a finite number when the result is infinite. It is impossible to enter infinity (`inf`) as an endpoint. Beyond these restrictions, it is very fast and reliable.

* Corresponding author.
E-mail addresses: sermutlu@cankaya.edu.tr (E. Sermutlu), h.eyyuboglu@cankaya.edu.tr (H.T. Eyyuboğlu).

0096-3003/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.amc.2006.08.176

Here our focus is on improper integrals, oscillatory integrals and in general other integrals that converge slowly. These types of integrals are encountered in majority of engineering applications, and in our terminology named as difficult integrals. To this end, we have developed a new algorithm, named `quade`, which is equipped specifically so that it can handle majority of these cases. It can be reached at `math.cankaya.edu.tr/~sermutlu/matlab/quade.m`.

2. Aims

Our aim is not to evaluate any given integral, because this is not possible. Nor is it to find the result with the fewest number of function evaluations. Today, the speed of computers make it impossible to sense the difference between 100 and 1000 function evaluations.

A quadrature program, by using reasonably apprehended tolerance levels, should give reliable results in a short time for regular integrals, simultaneously approximating difficult ones to an acceptable degree of accuracy, even if this is at the expense of extended computation time and increased number of function evaluations. For instance, there is no point in giving up in the first 3 s, if we are able to subdivide the interval and obtain a correct result in 3 min.

A common user demand is the inclusion of improper integrals. That is, the program should accept infinite range of integration, but at the same time, it should warn the user about divergent integrals. Besides, an ideal program should be able to deliver the result if the function is infinite at a single point, yet the integral as a whole is convergent.

Most engineering applications also require the evaluation of oscillatory integrals, perhaps again coupled with infinite ranges. Therefore, our aim in this study has concentrated on the correct evaluation of improper, oscillatory, slowly convergent integrals. Note that, the theme of the present work is limited to quadrature routines implemented in MATLAB environment, reference to other types will be the subject of future studies.

3. Methods

MATLAB works extremely fast with vectors and matrices. Using vector capabilities of MATLAB, multiple operations can be performed as if it were a single operation. This means much shorter computation times, and/or, more operations within the same time duration.

A simple example will serve to illustrate this important point. The following function implements the computation of Hermite polynomial, $H_n(x)$, in terms of for loops for a multidimensional argument, x.

```
function yout = H_loop(n,x)
nn = n + 1;[row,col] = size(x);yout = [];
for rowk = 1:row
   y = [];
   for colk = 1:col
   xx = x(rowk,colk);
h(1) = 1; h(2) = xx*2;yy = h(1);
if n~ = 0
   yy = h(2);
   if n~ = 1
      for j = 3:nn;h(j) = 2*xx*h(j-1) - 2*(j-2)*h(j-2);end
      yy = h(j);
   end
end
y = [y yy];
end
yout(rowk,:) = y;
end
```

According to conventional programming practices, the above code is the natural way to perform this specific task. But a rearranged version of the above code, also listed below, reveals that the same job can be executed much faster.

```
function yout = H_mat(n,x)
h = cat(4,ones (size(x)), 2*x, 4*x.^2-2);
for j = 4:n+1
   h(:,:,:,j) = 2*x.*h(:,:,:,j-1) - 2*(j-2)*h(:,:,:,j-2);
end
yout = h(:,:,:,n+1);
```

Via simple tests conducted, we find that, the latter routine is, on average, fifty times faster than the former, when the values of x, the argument of the Hermite polynomial, comprise a matrix of 200 by 200 from the randomly chosen range of 0–10, and $n = 3$. The reason why the second method takes much less time is because, this particular routine is suitably designed such that, matrix handling capability of MATLAB is fully utilized, whereas the first routine is based entirely on the concept of a single evaluation at a time.

Keeping this point in mind, we let our program work with vectors of several thousand components at every step. So even a simple function is evaluated at a large number of points. Using vector capabilities of MATLAB, these calculations are done very fast, so there is no point in minimizing the number of function evaluations. As seen in the tests section, adopting a large number of points will lead to successful outcomes for the difficult integrals that require millions of evaluations anyway.

Two main numerical integration methods are Gaussian and Newton–Cotes, each of them having a lot of variations. In Newton–Cotes, the points are equally spaced, whereas in Gaussian, the spacing of sample points is optimized. Therefore Gaussian quadrature is considered to be more efficient than Newton–Cotes formulas. When the same number of function evaluations is used, Gaussian rules produce better results, hence they are the choice of `quadl` and many others. It is also well known that the accuracy improves with increasing number of points [6]. The choice in our method is 64-point Gauss rule. We have used the associated abscissa and weight factors listed in [1].

The function is submitted together with the limits and relative error tolerance, abbreviated as `tol`, which is defined as the absolute value of error divided by the exact result. So roughly speaking, it corresponds to the number of correct digits. This constitutes an important difference between our program and those others such as `quadl`, using absolute tolerance levels. Although the default tolerance being at 10^{-6} is common to majority of programs, it will mean completely different things in the two respective cases. For example, consider the integral

$$\int_0^{10,000} x^9\,\mathrm{d}x = 10^{39}. \tag{1}$$

With the default tolerance of 10^{-6}, `quadl` produces the incorrect result, when evaluating this integral. The correct result of 10^{39} is to be generated only when the tolerance of 10^{30} or larger is specified for the parameter, `tol`. This is no surprise, since it is unreasonable to expect a 45-digits accuracy on a 16-digit system like MATLAB. In other words, an absolute tolerance of 10^{-6} is unnecessarily small for this problem, thus to obtain the correct result, we need to specify a tolerance of no less than 10^{30}. This obviously assumes that we have an estimate of the result before we begin the evaluation. Furthermore, the use of such an error level is awkwardly disturbing, because psychologically we are used to tiny errors. In contrast, `quade` uses relative error, and experiences no difficulty in obtaining the correct result to this type of problem with the default tolerance of 10^{-6}.

With the function, limits and tolerance figures supplied, our program, `quade` first categorizes the given integral. If the limits are infinite or the function values at the limits are infinite, two different subroutines calculate the replacement limits. If these limits cannot be found, the program prints an error message, warning the user that integral is probably divergent or alternatively the tolerance is too small. In the case of acquiring the replacement limits without any problems, the program continues the evaluation in the manner of proper integrals.

For a proper integral, the program `quade` uses 6400 and 3200 points, making two estimates of the integral. That is, it successively calls upon Gauss 64 point rule 100 and 50 times. The numerical difference between these two estimates, divided by the larger value is adopted as the relative error. If this relative error is below tolerance level specified at input, the program returns. Otherwise, the integral is treated as a difficult one, and consequently we proceed by subdividing the interval by 10, writing the endpoints of this subdivision to a matrix. This eliminates the need to invoke the function within the function, therefore we do not face the maximum recursion limit error of MATLAB. At each subinterval, we calculate the integral twice, again at 6400 and 3200 points, compare the results, and add the value given by 6400 points to the sum, if the error is below the specified tolerance level. At the end of this process, if there are no subintervals left, the program returns. Note that, because we are using relative error, the tolerance need not change with reduced interval size. This is another advantage of using relative error in an adaptive quadrature routine.

In case the size of the matrix containing endpoints exceeds a certain limit, the program, `quade` gives up. This limit is presently set to 10.000. It could also be selected as 100.000 and this would enable the program to obtain correct answers to some questions it cannot handle currently, but it would also mean that for a

Table 1
Results of battery test (part 1)

TF	quade	modsim	modlob	coteda	coteglob	quadl
1	2.58e−16	1.26e−10	0	5.68e−13	5.68e−13	7.99e−14
	2.58e−16	3.07e−14	0	2.58e−16	2.58e−16	7.99e−14
2	6.34e−16	0	0	1.59e−16	1.59e−16	1.59e−16
	6.34e−16	0	0	1.59e−16	1.59e−16	1.59e−16
3	5.76e−10	2.57e−07	7.83e−08	2.18e−07	3.96e−07	1.98e−06
	4.99e−16	1.52e−11	1.14e−12	4.28e−12	7.76e−12	4.25e−11
4	9.26e−16	1.01e−09	1.39e−15	4.83e−09	4.83e−09	6.79e−10
	9.26e−16	2.02e−13	6.95e−16	5.33e−15	5.21e−15	6.95e−16
5	5.61e−16	3.32e−08	2.17e−11	7.78e−10	7.78e−10	3.38e−06
	5.61e−16	3.03e−14	2.81e−16	2.10e−12	8.95e−13	2.67e−15
6	1.80e−15	2.78e−07	4.37e−08	3.31e−07	7.38e−07	1.14e−07
	1.80e−15	1.28e−11	3.20e−13	6.40e−12	1.25e−11	8.43e−13
7	3.17e−06	6.66e−02	8.76e−02	6.01e−02	1.66e−07	3.07e−06
	3.72e−11	1.32e−06	4.92e−07	1.49e−06	3.16e−08	3.15e−10
8	2.56e−16	6.04e−10	2.11e−10	4.96e−09	4.96e−09	2.08e−07
	2.56e−16	2.25e−13	0	2.05e−15	1.02e−11	1.71e−13
9	1.92e−16	9.65e−09	1.25e−10	6.86e−10	1.09e−09	8.21e−09
	1.92e−16	2.35e−12	5.77e−16	9.93e−13	1.05e−12	7.69e−16
10	3.20e−16	1.98e−08	2.20e−11	4.90e−08	4.90e−08	8.06e−09
	3.20e−16	1.37e−12	3.36e−14	2.35e−13	3.61e−12	1.60e−15
11	7.31e−16	1.01e−10	3.51e−15	3.10e−11	3.10e−11	4.21e−12
	7.31e−16	7.22e−13	5.85e−16	2.16e−14	3.10e−11	4.21e−12
12	1.66e−06	1.52e−07	2.05e−07	1.37e−07	1.37e−07	1.44e−07
	1.31e−10	1.43e−10	1.43e−10	1.43e−10	1.43e−10	1.44e−10
13	3.16e−14	7.00e−08	1.17e−10	1.80e−09	2.55e−09	3.01e−07
	3.16e−14	1.12e−12	1.03e−14	3.27e−13	9.04e−13	5.72e−15
14	2.22e−16	5.02e−08	2.28e−09	4.76e−08	3.99e−08	1.52e−08
	2.22e−16	6.18e−08	9.90e−14	1.87e−13	3.61e−13	3.69e−14
15	4.44e−16	1.03e−07	1.33e−09	6.96e−08	2.32e−09	9.30e−10
	4.44e−16	3.30e−12	5.80e−14	1.82e−13	7.46e−14	3.35e−11

divergent integral the program would take much longer to halt and give a warning to the user. With increasing speed of computers, the halting criteria may appropriately be modified in future.

4. Tests

In this section, comparative tests results, confined to MATLAB quadrature routines, are carried out. In MATLAB, the quadrature routine, `quad` is merely retained for compatibility with earlier versions. Since it is quite inferior to `quadl`, we have excluded it from our tests. Initially, we report comparisons between the routines named `modsim`, `modlob`, `coteda`, `coteglob`, `quadl` of MATLAB, and `quade` of our design using the so called battery test. The first four routines and the list of 23 integrand functions comprising battery test list are taken from [2]. For completeness, the 23 integrand functions are also reproduced in Table 3. The results are summarized in Tables 1 and 2 in terms of relative errors.

Note that for `quadl`, in the trivial case of $\int_a^b c_1\,\mathrm{d}x$ where c_1 is a constant, the integrand has to be converted into the form $c_1 - x + x$, whereas `quade` presents no difficulty with such integrals.

The relative error parameters, e_{05} and e_{10}, placed in the first and second rows in the respective column of the routines have the following definitions:

$$e_{05} = \frac{I_{05} - I_{\text{analytic}}}{I_{05}}, \quad e_{10} = \frac{I_{10} - I_{\text{analytic}}}{I_{10}}, \tag{2}$$

where I_{05}, I_{10} will correspond to the evaluation of the battery test integrand at tolerance levels of 10^{-5} and 10^{-10} via the routine in question, while I_{analytic} refers to the numeric value attained by using the analytic solution. During the tests, we have observed that the battery test functions (TF) numbered 7, 12 and 17 cannot be handled by `modesim`, `modlob`, `coteda` and `coteglob`, unless the lower limit of zero is replaced by 10^{-15}. From the relative error point of view, all routines seem to exhibit more or less the same behavior. Comparing e_{05} and e_{10}, we see that if a lower tolerance is demanded, less accurate results are obtained particularly for integrands containing singularity at the lower limits, e.g., this is the case for test functions 7, 12 and 17. By considering the magnitudes of e_{05} and e_{10} for majority of the battery test function evaluations, the superiority of `quade` is clearly demonstrated. Errors are usualy at machine precision level ($\approx 10^{-16}$) for `quade`. Although not listed here, there does not appear to be much difference among the tested routines from computation time point of view. Here we point out that, this is not actually a reliable measure of performance, since MATLAB

Table 2
Results of battery test (part 2)

TF	quade	modsim	modlob	coteda	coteglob	quadl
16	2.22e−16	3.27e−07	7.82e−09	7.07e−08	4.88e−08	8.75e−08
	2.22e−16	1.60e−11	8.76e−13	2.05e−11	2.35e−11	1.22e−15
17	2.63e−06	1.13e−08	1.03e−10	4.34e−09	1.26e−09	9.1e−05
	2.24e−11	3.69e−13	1.04e−13	5.38e−13	3.86e−14	3.89e−15
18	9.54e−15	4.65e−10	1.79e−10	4.48e−10	1.01e−09	6.74e−12
	9.54e−15	5.72e−15	1.14e−13	6.10e−15	3.10e−12	8.77e−15
19	1.52e−06	6.32e−07	3.88e−07	4.58e−07	3.26e−07	2.99e−06
	1.11e−16	3.82e−11	4.56e−12	1.13e−11	7.04e−12	9.56e−11
20	2.84e−16	1.16e−09	7.68e−11	6.15e−10	5.94e−09	1.84e−01
	2.84e−16	5.68e−15	0	1.14e−13	5.68e−15	2.26e−11
21	4.57e−14	2.41e−03	2.41e−03	2.41e−03	2.41e−03	2.41e−03
	1.70e−16	3.91e−12	2.4e−11	4.60e−13	2.25e−11	6.28e−15
22	3.67e−15	1.50e−10	3.04e−10	8.08e−10	2.06e−09	3.31e−10
	3.67e−15	3.29e−13	1.75e−16	3.15e−14	7.73e−13	1.40e−15
23	3.86e−16	4.71e−07	4.04e−09	6.28e−08	1.42e−08	4.18e−06
	3.86e−16	1.54e−11	1.03e−14	4.17e−13	9.91e−13	3.56e−12

Table 3
Test functions of the battery test

Test functions
1. $\int_0^1 \exp(x)\,\mathrm{d}x$
2. $\int_0^1 f(x)\,\mathrm{d}x$, where $f = 1$ if $x > 0.3$ else $f = 0$
3. $\int_0^1 \sqrt{x}\,\mathrm{d}x$
4. $\int_{-1}^1 \left(\frac{23}{25}\cosh(x) - \cos(x)\right)\mathrm{d}x$
5. $\int_{-1}^1 1/(x^4 + x^2 + 0.9)\,\mathrm{d}x$
6. $\int_0^1 \sqrt{x^3}\,\mathrm{d}x$
7. $\int_0^1 1/\sqrt{x}\,\mathrm{d}x$
8. $\int_0^1 1/(1 + x^4)\,\mathrm{d}x$
9. $\int_0^1 2/(2 + \sin(10\pi x))\,\mathrm{d}x$
10. $\int_0^1 1/(1 + x)\,\mathrm{d}x$
11. $\int_0^1 1/(1 + \exp(x))\,\mathrm{d}x$
12. $\int_0^1 x/(\exp(x) - 1)\,\mathrm{d}x$
13. $\int_{0.1}^1 \sin(100\pi x)/(\pi x)\,\mathrm{d}x$
14. $\int_0^{10} \sqrt{5}\exp(-50\pi x^2)\,\mathrm{d}x$
15. $\int_0^{10} 25\exp(-25x)\,\mathrm{d}x$
16. $\int_0^{10} 50/(\pi(2500x^2 + 1))\,\mathrm{d}x$
17. $\int_0^1 50(\sin(50\pi x)/(50\pi x))^2\,\mathrm{d}x$
18. $\int_0^\pi \cos(\cos(x) + 3\sin(x) + 2\cos(2x) + 3\cos(3x))\,\mathrm{d}x$
19. $\int_0^1 f(x)\,\mathrm{d}x$ if $x > 10^{-15}$ then $f = \log(x)$ else $f = 0$
20. $\int_{-1}^1 1/(1.005 + x^2)\,\mathrm{d}x$
21. $\int_0^1 \sum_{i=1}^3 1/\cosh(20(x - 2i/10))\,\mathrm{d}x$
22. $\int_0^1 4\pi^2 x\sin(20\pi x)\cos(2\pi x)\,\mathrm{d}x$
23. $\int_0^1 1/(1 + (230x - 30)^2)\,\mathrm{d}x$

does not calculate the computation time to a high precision. Secondly, in carrying out one time computation, the time consumed for the initial take-off shadows the actual quantity to be measured. Finally we note that, there exist slightly modified versions for test functions 18 and 21 in [3].

$$\begin{aligned}
&\int_0^2 x^{999}\,\mathrm{d}x = 1.07151 \times 10^{298},\\
&\int_0^{1000} x^4\,\mathrm{d}x = 2.00000 \times 10^{14},\\
&\int_0^{40} \exp(x)\,\mathrm{d}x = 2.35385 \times 10^{17},\\
&\int_0^{10} \sinh x\cosh^4 x\,\mathrm{d}x = 3.24044 \times 10^{19}.
\end{aligned} \tag{3}$$

Next we consider difficult integrals, that is oscillatory integrals, improper integrals and in general other integrals that converge slowly. For this purpose, we start with the simplest cases of integrals appearing in Eq. (3). Our routine, `quade` will easily evaluate all of these integrals correctly up to six digit accuracy as shown, when it is called in the format, `quade(fnc,a,b)`, implying that the default tolerance of 10^{-6} is used. On the other hand, `coteglob` and `quadl` will fail in the first integral of Eq. (3), while `quadl` will also fail in the third and the fourth, mainly because for `quadl`, a tolerance matching the magnitude of the actual result has not been assigned. This means that in the case of `quadl` tolerance value for the first integral has to be larger than 10^{285}. For the third and fourth integrals the minimum tolerance levels of 10^{-1}, 10^1 are respectively required.

$$
\begin{aligned}
&\int_0^{1000} \cos x \,\mathrm{d}x = 0.826880,\\
&\int_0^{400} x^3 \exp(-x)\,\mathrm{d}x = 6.00000,\\
&\int_0^{10000} x \sin^2 x \,\mathrm{d}x = 2.49985 \times 10^7,\\
&\int_0^{50} \exp(x)\sin(100x)\,\mathrm{d}x = -8.53048 \times 10^{18},\\
&\int_1^{10^{300}} \frac{1}{x}\,\mathrm{d}x = 690.7755.
\end{aligned}
\tag{4}
$$

The integrals in Eq. (4) may be regarded to be among the most difficult ones for numerical integration in terms of number of evaluations. It is important to realize that the integration range of the last integrand is so huge that it is at the edge of MATLAB's capabilities. $2^{1023} \approx 10^{309}$ is the largest number MATLAB can handle) These integrals can be evaluated by `quade` up to tolerance level of 10^{-12}, with ease, but the same is not valid for the rest of the routines. For instance, in the first integral, `coteda` and `coteglob` cannot produce the correct result unless a tolerance level less than 10^{-8} is used, while `quadl` only gives the correct result up to the specified digit when a tolerance range of 10^{-2}–10^{-5} is used. On the other hand, for the second integral in Eq. (4), all routines will function satisfactorily except that `quadl` will only supply the correct result only when a tolerance range of 10^{-10}–10^{-17} is specified. In the third integral, `coteglob` and `quadl` encounter the problem of maximum number of function evaluations, thus generating incorrect results. Furthermore, for the evaluation of this integral, excessively longer computation times are required by the routines of `modsim`, `modlob` and `coteda`. It is possible to obtain an accuracy up to six digits in the fourth integral of Eq. (4), if only a tolerance level of 10^{-10} or less is set for `modsim`, `modlob` and `coteda`, while `quadl` will again fail due to its inherent absolute tolerance concept. Finally, in the last integral of Eq. (4), `modsim` will not be able to complete the evaluation, `modlob`, `coteda` and `coteglob` will output the result of `Inf` (infinite) and `quadl` will exhibit an incorrect result along the warning message of "`Minimum step size reached; singularity possible`". In all the testable cases of Eq. (4), `quade` is observed to have a distinct computation time advantage over the rest of the routines.

$$
\begin{array}{lllll}
 & \text{SLOW} & & \text{FAST} & \\
 & t_{\text{quadl}} & t_{\text{quade}} & t_{\text{quadl}} & t_{\text{quade}} \\
\displaystyle\int_0^{500} \cos x \,\mathrm{d}x = -0.467772, & 1.1 & 0.05 & 0.627 & 0.016 \\
\displaystyle\int_1^{1000} \frac{\sin x}{x}\,\mathrm{d}x = 0.624150, & 0.99 & 0.05 & 0.546 & 0.015 \\
\displaystyle\int_0^{1000} \exp(-x/100)\sin x \,\mathrm{d}x = 0.999874, & 1.10 & 0.05 & 0.642 & 0.015 \\
\displaystyle\int_0^{2\pi} \cos(300\sin x)\,\mathrm{d}x = -0.209221, & 1.54 & 0.05 & 0.797 & 0.015 \\
\displaystyle\int_1^{10^8} \frac{1}{x}\,\mathrm{d}x = 18.4207, & 0.11 & 0.22 & 0.062 & 0.156 \\
\displaystyle\int_0^{100} \sqrt{x}\exp(-x)\,\mathrm{d}x = 0.886227, & 0.05 & 0.05 & 0.032 & 0.015
\end{array}
\tag{5}
$$

Both `quade` and `quadl` can evaluate the integrals in Eq. (5) correctly at default tolerance of 10^{-6}. The exact results are not much larger or smaller than 1, hence absolute and relative tolerances will have the same meaning and this way `quadl` experiences no problems. As seen from the run times displayed, for oscillatory integrals, `quade` is faster, while for the other type of integrals, `quadl` will, in some situations have a shorter time. To the right of Eq. (5), computation time is given in seconds both for a platform of a Pentium 733 MHz PC having 256 MB RAM and MATLAB 6.5 installation (SLOW) and a platform of Pentium 1.5 GHz PC having 512 MB RAM and MATLAB 7 installation (FAST).

The improper integrals of Eq. (6) can be evaluated correctly by `quade` up to tolerance 10^{-14}. This means, given tolerance of 10^{-n}, `quade` will return a result correct at least up to n digits. To the right of Eq. (6), computation time is given in seconds both for a platform of a Pentium 733 MHz PC having 256MB RAM and MATLAB 6.5 installation (SLOW) and a platform of Pentium 1.5 GHz PC having 512 MB RAM and MATLAB 7 installation (FAST). In MATLAB, the use of infinite limit is permitted solely in symbolic integration. For numerical integration routines `modsim`, `modlob`, `coteda`, `coteglob` and `quadl`, it is impossible to enter `inf` as an integration limit. An equivalence may be established by replacing the infinite limit with a large finite value. But the question is to guess what numerical value will be sufficiently large. In our investigations of integrals in Eq. (6), it was discovered that upper limits of 10–10^{10} will produce accurate results ranging in one to nine digit accuracy. Such a variation is a reflection of uncertainty for other routines. Within the context of improper integrals, another difficulty faced with those routines is that, in some cases with increasing value of limits, the magnitude of the integration does not converge towards the correct value, but may instead oscillate to higher values.

The integrals in Eq. (7) are much more difficult in the sense that they are both oscillatory and improper, so the best `quade` tolerance that may be achieved is 10^{-5}. Inevitably, the computation time spent for integrals of Eq. (7) is much higher than the previous ones. Provided that convergence is ensured in the asymptotic behavior of the integrand, it is possible to extend the list offered in Eq. (7) indefinitely by simply increasing the power of the polynomial in denominator.

To the right of Eqs. (6) and (7) we again show the computation times of `quade` in seconds arranged in columns of *SLOW* and *FAST* as defined above.

Computation time for `tol` $= 10^{-6}$

	SLOW	FAST	
$\int_{-\infty}^{\infty} \frac{\mathrm{d}x}{(1+x^2)\sqrt{4+3x^2}} = \frac{\pi}{3},$	0.22	0.031	
$\int_0^{\infty} \frac{x^2\,\mathrm{d}x}{(1+x^2)^2} = \frac{\pi}{4},$	0.33	0.094	
$\int_0^{\infty} \frac{\mathrm{d}x}{\left(x+\sqrt{1+x^2}\right)^2} = \frac{2}{3},$	0.39	0.328	
$\int_0^{\infty} \exp(-4x)\,\mathrm{d}x = \frac{1}{4},$	0.06	0.015	(6)
$\int_0^{\infty} \frac{\mathrm{d}x}{1+\exp(3x)} = \frac{\ln 2}{3},$	0.06	0.016	
$\int_{-\infty}^{\infty} \frac{\mathrm{d}x}{1+x+x^2} = \frac{2\pi}{\sqrt{3}},$	0.44	0.14	
$\int_{-\infty}^{\infty} \exp(-x^2)\,\mathrm{d}x = \sqrt{\pi},$	0.11	0.063	

Computation time for `tol` $= 10^{-5}$

	SLOW	FAST	
$\int_0^{\infty} \frac{\cos x\,\mathrm{d}x}{4+x^2} = \frac{\pi}{4}\exp(-2),$	0.39	0.313	
$\int_0^{\infty} \frac{\sin x\,\mathrm{d}x}{x} = \frac{\pi}{2},$	5.33	04.26	(7)
$\int_0^{\infty} \frac{\sin^2 5x}{x^2}\mathrm{d}x = \frac{5\pi}{2},$	13.95	10.234	
$\int_0^{\infty} \frac{x\sin x\,\mathrm{d}x}{1+x^2} = \frac{\pi}{2}\exp(-1),$	48.56	40.25	

The integrals in Eq. (8) are divergent, and `quade` warns the user about this feature accordingly.

$$\begin{aligned}
&\int_0^1 \frac{1}{x}\,\mathrm{d}x,\\
&\int_1^\infty \frac{1}{x}\,\mathrm{d}x,\\
&\int_0^1 \frac{1}{x^2}\,\mathrm{d}x,\\
&\int_1^\infty \frac{1}{\sqrt{x}}\,\mathrm{d}x,\\
&\int_0^\infty \exp(x)\,\mathrm{d}x,\\
&\int_1^\infty \ln(x)\,\mathrm{d}x.
\end{aligned} \tag{8}$$

As stated in Section 2, no quadrature routine can be expected to support the entire range of integrals available in the world of mathematics. The failures of our program `quade` that we could find, are summarized in Eq. (9). These integrals are convergent, yet, our program reports them to be divergent. Note that the convergence rate of these integrals is extremely slow.

$$\begin{aligned}
&\int_0^1 \frac{1}{x^p}\,\mathrm{d}x, \quad 0.75 < p < 1,\\
&\int_1^\infty \frac{1}{x^p}\,\mathrm{d}x, \quad 1 < p < 1.25,\\
&\int_0^\infty \frac{\cos x}{\sqrt{x}}\,\mathrm{d}x,\\
&\int_0^n \sin x\,\mathrm{d}x, \quad n > 1.000.000.
\end{aligned} \tag{9}$$

5. Conclusion

For use in MATLAB, we have described a new integration routine, called `quade`. Its advantages, compared to the existing MATLAB routines may be summarized as follows:

- Can handle improper integrals with `inf` as limit.
- Can cope with oscillatory and slowly converging integrals that are numerically very difficult to evaluate.
- Much higher limits in number of function evaluations. Hence it aims to attain the correct result at the expense of increased execution time and function evaluations.

In this respect, we envisage that our program will be useful in many scientific and engineering applications, calculations of special functions by integral formulas, and other areas where rapidly oscillating integrands and infinite ranges of integration are encountered. It is foreseen a similar action for the double and triple quadrature routines, needs to be undertaken.

References

[1] M. Abramowitz, I.A. Stegun, Handbook of Mathematical Functions, vol. 918, Dover Publications, 1965.
[2] T.O. Espelid, Doubly adaptive quadrature routines based on Newton–Cotes rules, BIT 43 (2003) 319–337.
[3] G. Evans, Practical Numerical Analysis, vol. 205, John Wiley & Sons, Chichester, 1995.

[4] G. Gander, W. Gautschi, Adaptive quadrature-revisited, BIT 40 (2000) 84–101.
[5] M. El-Mikkawy, A unified approach to Newton–Cotes quadrature formulae, Applied Mathematics and Computation 138 (2003) 403–413.
[6] E. Sermutlu, Comparison of Newton–Cotes and Gaussian methods of quadrature, Applied Mathematics and Computation 171 (2005) 1048–1057.
[7] J.A.C. Weideman, D.P. Laurie, Quadrature rules based on partial fraction expansions, Numerical Algorithms 24 (2000) 159–178.