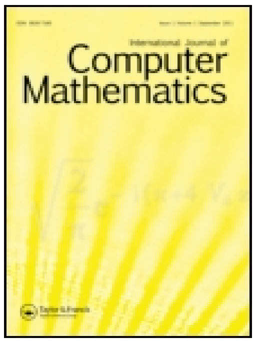

International Journal of Computer Mathematics

ISSN: 0020-7160 (Print) 1029-0265 (Online) Journal homepage: http://www.tandfonline.com/loi/gcom20

Integrating oscillatory functions in Matlab

L. F. Shampine

International Journal of Computer Mathematics
Vol. 88, No. 11, July 2011, 2348–2358

Integrating oscillatory functions in MATLAB

L.F. Shampine*

Mathematics Department, Southern Methodist University, Dallas, TX 75275, USA

(*Received 23 September 2010; accepted 25 October 2010*)

When ω is large, the integrand of $\int_a^b f(x)\mathrm{e}^{\mathrm{i}\omega x}\,\mathrm{d}x$ is highly oscillatory and conventional quadrature programs are ineffective. A new method based on a smooth cubic spline is implemented in a MATLAB program `osc` that is both easy to use and effective for large ω. Other methods are used in the program to deal effectively with small ω. Because the implementation of the basic method is adaptive, the program deals comparatively well with $f(x)$ that have peaks. With the assistance of another method, the program is able to deal effectively with $f(x)$ that have a moderate singularity at one or both ends of $[a, b]$. The algorithms and user interface of `osc` exploit the capabilities of the MATLAB computing environment.

Keywords: quadrature; oscillatory integrand; Filon; smooth cubic spline; MATLAB

2010 AMS Subject Classifications: 65D30; 65D32; 65D07

1. Introduction

The popular methods for the numerical approximation of definite integrals on finite intervals are based on approximating the integrand by a polynomial and then integrating the polynomial exactly. This approach is not satisfactory for integrals of the form

$$I(f) = \int_a^b f(x)\mathrm{e}^{\mathrm{i}\omega x}\,\mathrm{d}x \tag{1}$$

when the parameter ω is large in magnitude because the oscillatory factor $\mathrm{e}^{\mathrm{i}\omega x}$ makes the integrand hard to approximate with a polynomial. Often such integrals arise in trigonometric form, but it is convenient in both theory and practice to use the complex form and then obtain the trigonometric integrals from the real and imaginary parts of the result:

$$I(f) = \int_a^b f(x)\cos(\omega x)\,\mathrm{d}x + \mathrm{i}\int_a^b f(x)\sin(\omega x)\,\mathrm{d}x.$$

*Email: lfshampine@aol.com

ISSN 0020-7160 print/ISSN 1029-0265 online
© 2011 Taylor & Francis
DOI: 10.1080/00207160.2010.537330
http://www.informaworld.com

A classical technique [3, §2.10] for approximating Equation (1) is the asymptotic expansion given in Section 2. The lowest order term

$$Q_A(f) = \frac{1}{\mathrm{i}\omega}(\mathrm{e}^{\mathrm{i}\omega b} f(b) - \mathrm{e}^{\mathrm{i}\omega a} f(a))$$

already provides an approximation of $I(f)$ that is accurate to $O(1/\omega^2)$. This simple formula is a practical way to approximate the integral for large ω. It shows that the difficulty such integrals present for standard quadrature programs is not intrinsic, rather the result of an approach inappropriate to the task. Many authors have studied the approximation of $I(f)$ for large ω. A nice comparison of some leading possibilities is given by Evans and Webster [4]. In this paper, we approach the task from a different point of view, that of mathematical software: a program is provided with a function $f(x)$ and a *given* ω and computes an approximation of *specified* accuracy. The program should be effective for all ω.

To our knowledge, there are only two pieces of mathematical software for the task that are widely available, namely the FSER1 program of Chase and Fosdick [1,2] and the AINOS program of Piessens and Branders [9]. (Variations of AINOS appear in QUADPACK [10] and several widely used libraries.) These programs are written in Fortran. In this paper, we develop a new method for the task and implement it as a piece of software called `osc` that is written in MATLAB [8]. The first effective way of approximating $I(f)$ for all ω is that of Filon [5]. For this article, it is convenient to describe the method as approximating $f(x)$ on $[a, b]$ by splitting the interval into subintervals of equal length and forming a continuous spline $S_N(x)$ by interpolating $f(x)$ with a quadratic polynomial at the ends and middle of each subinterval. An approximation to $I(f)$ is obtained by evaluating analytically

$$Q_N(f) = \int_a^b S_N(x)\mathrm{e}^{\mathrm{i}\omega x}\,\mathrm{d}x. \tag{2}$$

The FSER1 program is an iterative, non-adaptive implementation of Filon's method. The scheme of Piessens and Branders can be described in a similar way with the spline $S_N(x)$ defined by high-order polynomial interpolation at Chebyshev nodes in each subinterval. The AINOS program is an adaptive implementation of the basic formula. We obtain a new formula of moderate order by taking $S_N(x)$ to be a smooth cubic interpolatory spline. A novelty of this formula is that the smooth cubic spline is determined by all the data and the other formulas are based on independent interpolation on subintervals. We show in Section 3 that the smooth cubic spline leads to a remarkably simple and efficient evaluation of Equation (2) in MATLAB.

To achieve a specified accuracy, it may be necessary to use subintervals so short that the usual way of evaluating formulas is unsatisfactory in finite precision arithmetic. Chase and Fosdick [1,2] discuss at length how they deal with this when evaluating the formulas of Filon's method. Piessens and Branders [9] take a different approach to similar difficulties with their method. We explain in Section 6 how we do something similar in `osc`. Related to this issue of short subintervals is the task of computing $I(f)$ when ω is small enough that the integrand has only a few oscillations on the whole interval $[a, b]$. A novelty of `osc` is that it recognizes such problems and integrates them very efficiently with a different kind of method.

Adaptive implementations of a basic quadrature formula allow a program to deal effectively with $f(x)$ that have peaks or discontinuities in low-order derivatives. Kahaner [7] describes several kinds of algorithms for adaptive quadrature. As we explain in Section 7, the algorithm of `osc` is quite different from that of AINOS because it exploits more fully the characteristics of the MATLAB computing environment. In Section 7, we explain how `osc` can also deal effectively with $f(x)$ that have moderate singularities at one or both ends of the interval. To invoke this capability, the user need only set the optional input argument `singular` to `true` – the program does not ask about which end might have a singularity or what kinds of moderate singularities might be present.

2. Preliminaries

We suppose of Equation (1) that the interval $[a, b]$ is real and finite, the parameter ω is real, and $f(x)$ is a real-valued function that is moderately smooth on subintervals of (a, b). If $f(x) \in C^m[a, b]$, integration by parts [3, p. 60] provides the asymptotic expansion

$$I(f) = -\mathrm{e}^{\mathrm{i}\omega a} \sum_{k=0}^{m-1} i^{k-1} \omega^{-k-1} f^{(k)}(a) + \mathrm{e}^{\mathrm{i}\omega b} \sum_{k=0}^{m-1} i^{k-1} \omega^{-k-1} f^{(k)}(b) + (-\mathrm{i}\omega)^{-m} \int_a^b f^{(m)}(x) \mathrm{e}^{\mathrm{i}\omega x}\, \mathrm{d}x. \quad (3)$$

Certainly, $I(f)$ is difficult to approximate for large ω with conventional methods, but we shall see that it is easy enough to approximate the integral if a suitable method is used. However, there is a practical point to be kept in mind. The asymptotic expansion (3) says that $I(f)$ is usually $O(1/\omega)$, so if ω is large and we want an approximation with some significant figures, we must use some kind of relative error control. We have implemented a scaled absolute error control in `osc`. Specifically, the tolerance is to bound the absolute error relative to $1/\max(|\omega|, 1)$. The basic method of `osc` is of order four, which is not appropriate for stringent relative accuracies. With this in mind, the default tolerance of `osc` is 10^{-3}. If $|\omega| > 1$, `osc` will not attempt a scaled error smaller than 10^{-8}, increasing the tolerance input to this value as needed.

Although the algorithms we use are attractive in general scientific computation, we study here the approximation of $I(f)$ in the MATLAB [8] computing environment. In this environment, commands are normally interpreted, but critical computations are compiled for speed. For this reason, the efficiency of an algorithm can often be enhanced greatly by skilful use of array operations and built-in functions. This is so important that all the quadrature programs that are provided with MATLAB require that evaluation of the integrand be vectorized. By this it is meant that the function for evaluating the integrand must accept a vector of arguments and return a vector of corresponding function values. With careful coding of the evaluation and the reduced cost of a single call to the function, it is often the case that the run time depends weakly on the number of arguments. In the study of numerical quadrature, it is customary to measure the cost of a program by counting the number of function evaluations. We do that for our examples, but we count the number of *array* function evaluations.

Our basic method requires the computation of a smooth cubic interpolatory spline. This involves the solution of a linear system of moderate size. Often it is thought that this is relatively expensive, but it is not, even in scalar computation, because the matrix is tridiagonal and there is a very simple, explicit algorithm for the purpose. A natural generalization of Filon's method is to interpolate with a cubic on each subinterval instead of a quadratic. It might seem to be less expensive to compute this spline than the smooth spline, but a count of arithmetic operations shows that the cost is actually comparable. Moreover, `osc` computes the smooth spline with the built-in function `spline` that does the critical computation in a different way – it treats the tridiagonal matrix as a general sparse matrix for which MATLAB has a compiled program that is very fast. We conclude that our method is practical in general scientific computation and at least as fast as natural alternatives in MATLAB.

3. Smooth cubic splines

In modern terminology, Filon [5] approximates $f(x)$ on $[a, b]$ by interpolating it with a quadratic spline $S_N(x)$, specifically a continuous spline defined by independent quadratic interpolation on subintervals of equal length. He then integrates Equation (2) analytically to obtain an approximation $Q_N(f)$ to $I(f)$. We generalize this to a smooth cubic spline, but there is some choice as to which smooth cubic spline we might use for $S_N(x)$. For given nodes

$$a = x_1 < \cdots < x_N = b \tag{4}$$

a smooth cubic spline $S_N(x)$ is a cubic polynomial on $[x_j, x_{j+1}]$ for $j = 1, \ldots, N-1$. It is $C^2[a, b]$ and interpolates $f(x)$ at each of the nodes. The complete cubic spline is defined by the end conditions

$$S'_N(a) = f'(a), \quad S'_N(b) = f'(b). \tag{5}$$

If $h = \max(x_{j+1} - x_j)$, it is shown in [11] that for a sufficiently smooth $f(x)$, the derivative $S_N^{(k)}(x)$ approximates $f^{(k)}(x)$ uniformly on $[a, b]$ with an accuracy of $O(h^{4-k})$ for $k = 0, 1, 2$. This spline has some advantages, but we did not want to ask users to supply a function to evaluate $f'(x)$ for the end conditions. There are two common ways [11] of getting a smooth cubic spline approximation with the same order of accuracy that do not require a function for the first derivative. Instead of interpolating $f'(x_1)$, we can interpolate the derivative at x_1 of a cubic polynomial that interpolates at x_1, x_2, x_3, x_4 and similarly at x_N. This is done in the `spcoef` program of [13]. Another way is to use the not-a-knot condition. It requires that the third derivative of the spline be continuous at x_2 and x_{N-1}. This implies that the cubic polynomial on $[x_1, x_2]$ is the same as that on $[x_2, x_3]$ and similarly at the other end. The `spline` program of MATLAB has an option to compute the complete cubic spline, but the default is the cubic spline with not-a-knot conditions. We experimented with both the complete cubic spline and the not-a-knot end conditions. The complete cubic spline did not show a significant advantage, which agrees with our analysis of the rates of convergence in Section 4. Accordingly, `osc` uses the smooth cubic spline with not-a-knot end conditions and so avoids asking users for a derivative function.

Because we integrate a smooth cubic spline, it is remarkably easy to evaluate analytically the approximation $Q_N(f)$ of Equation (2). On each subinterval $[x_j, x_{j+1}]$, the spline $S_N(x)$ is a cubic polynomial. When the asymptotic expansion (3) is applied to $S_N(x)$ on this subinterval, the third derivative is constant and the fourth is identically zero, so the expansion is exact. When these expressions are summed, the terms corresponding to derivatives that are continuous at interior nodes cancel out. We integrate directly the piecewise constant function $S_N^{(3)}(x)$. The result is

$$\begin{aligned} Q_N(f) = {} & \frac{1}{\mathrm{i}\omega}(-\mathrm{e}^{\mathrm{i}\omega a} f(a) + \mathrm{e}^{\mathrm{i}\omega b} f(b)) \\ & + \frac{1}{\omega^2}(-\mathrm{e}^{\mathrm{i}\omega a} S'_N(a) + \mathrm{e}^{\mathrm{i}\omega b} S'_N(b)) \\ & + \frac{\mathrm{i}}{\omega^3}(-\mathrm{e}^{\mathrm{i}\omega a} S_N^{(2)}(a) + \mathrm{e}^{\mathrm{i}\omega b} S_N^{(2)}(b)) \\ & - \frac{2\mathrm{i}}{\omega^4} \sum_{j=1}^{N-1} S_N^{(3)}(x_j+) \mathrm{e}^{\mathrm{i}\omega(x_j + h_j/2)} \sin(\omega h_j/2). \end{aligned} \tag{6}$$

As we use this formula in `osc`, the mesh spacing is a constant h and the expression then simplifies to

$$\begin{aligned} Q_N(f) &= \frac{1}{\mathrm{i}\omega}(-\mathrm{e}^{\mathrm{i}\omega a} f(a) + \mathrm{e}^{\mathrm{i}\omega b} f(b)) \\ &+ \frac{1}{\omega^2}(-\mathrm{e}^{\mathrm{i}\omega a} S_N'(a) + \mathrm{e}^{\mathrm{i}\omega b} S_N'(b)) \\ &+ \frac{i}{\omega^3}(-\mathrm{e}^{\mathrm{i}\omega a} S_N^{(2)}(a) + \mathrm{e}^{\mathrm{i}\omega b} S_N^{(2)}(b)) \\ &- \frac{2\mathrm{i}\sin(\omega h/2)\mathrm{e}^{\mathrm{i}\omega h/2}}{\omega^4} \sum_{j=1}^{N-1} S_N^{(3)}(x_j+)\mathrm{e}^{\mathrm{i}\omega x_j}. \end{aligned} \tag{7}$$

Programs for computing the cubic spline use different representations of $S_N(x)$ on subintervals, but it should be easy to extract the derivatives needed to evaluate this expression. In the case of `spline`, the piecewise polynomial representation contains a field `coefs` that is an array C of size $(N-1)\times 4$. On $[x_j, x_{j+1})$ the spline is

$$S_N(x) = C_{j,1}(x-x_j)^3 + C_{j,2}(x-x_j)^2 + C_{j,3}(x-x_j) + C_{j,4}.$$

From this it is immediate that

$$\begin{aligned} S_N^{(2)}(a) &= 2C_{1,2}, \\ S_N^{(2)}(b) &= 2C_{N-1,2} + 6C_{N-1,1}h, \\ S_N^{(3)}(x_j+) &= 6C_{j,1}. \end{aligned}$$

All the $\mathrm{e}^{\mathrm{i}\omega x_j}$ can be evaluated in a single call to the exponential function with array argument. Using the built-in function for the dot product of two vectors, the last term in Equation (7) can be evaluated efficiently in a single line of MATLAB code.

Section 5 discusses how an estimate of the error of an approximation is used to achieve a specified accuracy. Here we note that the FSER1 code estimates the error of a result by comparing it to the result of using twice as many nodes. The number of nodes is doubled so that all the $f(x_j)$ formed for one result can also be used for the second. We take the same approach in `osc`. First we compute an approximation $Q29$ using 29 equally spaced nodes and then use every other node to compute $Q15$. It is plausible, and we quantify the matter in Section 4, that $Q29$ is substantially more accurate than $Q15$. Accordingly, the error of $Q15$ is approximately equal to $Q29 - Q15$. If this error is sufficiently small, the program actually accepts $Q29$ because it is believed to be a better approximation to the integral than $Q15$. This contributes to a conservative control of error.

4. Convergence

The approach we take to approximating Equation (1) is first to approximate $f(x)$ with a function $S_N(x)$ and then to integrate analytically

$$Q_N(f) = \int_a^b S_N(x)\mathrm{e}^{\mathrm{i}\omega x}\,\mathrm{d}x.$$

It follows immediately that

$$|I(f) - Q_N(f)| \le \int_a^b |f(x) - S_N(x)|\,|\mathrm{e}^{\mathrm{i}\omega x}|\,\mathrm{d}x \le \|f - S_N\|\,|b-a|. \tag{8}$$

Here the norm

$$\|f - S_N\| = \max_{a \le x \le b} |f(x) - S_N(x)|.$$

As far as this bound is concerned, the error is not affected by the size of ω. A number of authors have considered formulas based on interpolatory quadrature. They fit into this framework if the formula is used in composite (compound) form. Such methods can be described as coming from a function $S_N(x)$ that is a polynomial of degree k on each subinterval of Equation (4). This polynomial is defined by interpolation to $f(x)$ at $k+1$ distinct points in the subinterval. Let $h_j = x_{j+1} - x_j$ and $h = \max h_j$. Filon's method can be viewed as arising in this way by quadratic interpolation at the middle and both ends of the subinterval. Interpolation at both ends of each subinterval implies that $S_N(x)$ is continuous on $[a, b]$, but it generally has jumps in the first derivative at interior nodes. Similarly, on each subinterval, the method of Piessens and Branders [9] interpolates at Chebyshev nodes with a polynomial of high degree. The polynomial interpolates at both ends of each subinterval, so this $S_N(x)$ is also continuous on $[a, b]$. The error of $Q_N(f)$ can then be bounded by applying standard results for polynomial interpolation on each subinterval. If $f(x)$ is sufficiently smooth on $[a, b]$, the interpolation error on each subinterval is $O(h_j^{k+1})$ (cf. [13]). Correspondingly, we find that $Q_N(f)$ approximates $I(f)$ with an error that is $O(h^{k+1})$. This is an easy way to see that Filon's method is accurate to $O(h^3)$, but there are other ways to obtain this result. In particular, Chase and Fosdick [1] analyse the error in a way particular to Filon's method that provides a more detailed description of the error.

We now see that the natural generalization of Filon's method to independent cubic interpolation provides an approximation $Q_N(f)$ that is accurate to $O(h^4)$. Our scheme obviously fits into this overall framework, but the approximation $S_N(x)$ is not made up of pieces computed independently by interpolation, so we must turn to other results from approximation theory to establish convergence. As we noted in Section 3, standard results for splines state that if $f(x)$ is sufficiently smooth on $[a, b]$, then $\|f - S_N\|$ is $O(h^4)$ for the complete cubic spline and for both variants that avoid the use of the first derivative at end points. We conclude that the practical differences between a cubic spline determined by independent interpolation and a smooth cubic spline are the cost of forming the spline and the smoothness of $S_N(x)$. In MATLAB, we can form a smooth cubic spline with `spline` just as fast as the spline defined by independent polynomial interpolation. We prefer the smooth cubic spline because the expression (6) provides a remarkably simple and efficient way to evaluate $Q_N(f)$ in MATLAB for the smooth cubic spline returned by `spline`.

It is to be appreciated that a quadrature program is presented as an integrand with a *given* value of ω, and it is then to produce an approximation of specified accuracy. Still, it is illuminating to consider the error of schemes for approximating $I(f)$ for large ω. Iserles and Nørsett [6] prove some interesting results of this kind for interpolatory quadrature schemes, including schemes that interpolate values of $f'(x)$ in addition to values of $f(x)$. An attractive aspect of our approach is that Equation (6) is already an asymptotic expansion of $Q_N(f)$. The first term in this expansion is precisely the simple approximation $Q_A(f)$ discussed briefly in Section 1. Using more terms in the asymptotic expansion (3), we find that for any of the smooth cubic splines

$$\begin{aligned} I(f) - Q_N(f) = {} & \frac{1}{\omega^2}[-\mathrm{e}^{\mathrm{i}\omega a}(f^{(1)}(a) - S_N^{(1)}(a)) + \mathrm{e}^{\mathrm{i}\omega b}(f^{(1)}(b) - S_N^{(1)}(b))] \\ & + \frac{i}{\omega^3}[-\mathrm{e}^{\mathrm{i}\omega a}(f^{(2)}(a) - S_N^{(2)}(a)) + \mathrm{e}^{\mathrm{i}\omega b}(f^{(2)}(b) - S_N^{(2)}(b))] \\ & + O(1/\omega^4). \end{aligned}$$

The end conditions (5) of the complete cubic spline show that for this spline, the error is $O(1/\omega^3)$. Iserles and Nørsett show that for interpolatory quadrature formulas, it is necessary to interpolate derivatives to get approximations with this order of accuracy. However, they do not consider composite formulas and when this is done, a different picture of the error emerges. The asymptotic expansion of the error shows that the rate of convergence of the smooth cubic splines that do not use derivatives is only $O(1/\omega^2)$. However, in addition to approximating $f(x)$ to $O(h^4)$, the splines also approximate its derivatives. In particular, the coefficient of the leading term of the asymptotic expansion of the error is much smaller than one might expect because $S_N^{(1)}(x) = f^{(1)}(x) + O(h^3)$. Indeed, the same is true of the next term in the expansion because $S_N^{(2)}(x) = f^{(2)}(x) + O(h^2)$. This new observation is needed to understand practical computation because ω is given and h is reduced as necessary to obtain the required accuracy. We believe that it explains why we have seen so little advantage to the complete cubic spline in experiments that compared it with the smooth cubic spline with not-a-knot end conditions of `osc`.

We consider the approximation of the complex integral

$$\int_0^1 \cosh(x)\mathrm{e}^{\mathrm{i}\omega x}\,\mathrm{d}x = -\frac{\mathrm{i}\mathrm{e}^{\mathrm{i}\omega-1}-1}{2(\omega+i)} - \frac{\mathrm{i}\mathrm{e}^{\mathrm{i}\omega+1}-i}{2(\omega-i)} \tag{9}$$

with `osc` for a wide range of ω, specifically $\omega = 10^p$ for $p = 0, 1, \dots, 6$. Because $\omega \geq 1$ in all cases, the scaled absolute error of `osc` is absolute error relative to $1/|\omega|$. The integral is $O(1/|\omega|)$, so this resembles a control of the relative error of the approximation. We also use this example to show that it is quite easy to use `osc`, especially with the default tolerance of 10^{-3}. All that is necessary to approximate this integral is to assign the desired value of ω to the variable `omega` and issue the command

```
    Q = osc(@(x) cosh(x),omega,0,1);
```

For ω equal to 1 and 10, the program called upon `quadgk` as explained in Section 6 to carry out the integration directly. Each required only one array function evaluation and the *larger* of the two scaled errors was 5.6×10^{-16}. The new method was used for the remaining values of ω. In every case the program used only *one* array function evaluation. The *largest* scaled error was 1.2×10^{-8}, which corresponds to $\omega = 100$. Clearly the task does not become harder as ω increases. Indeed, for $\omega = 10^6$, the scaled error was 2.5×10^{-13}, this despite the fact that $I(f)$ is then approximately -5.4×10^{-7}. We might observe that a strong conventional program like `quadgk` does surprisingly well at integrating $I(f)$ directly: for $\omega = 10^3$, it used only five array evaluations to get a result with a scaled error of 3.7×10^{-13}. Although it returned a message casting doubt on the accuracy of the result for $\omega = 10^4$, the program was successful. The program had a hard failure for $\omega = 10^5$.

5. Adaptive quadrature

The first piece of mathematical software for approximating (1) (in real form) is the FSER1 program of Chase and Fosdick [1,2] that implements Filon's method. To get an approximate integral of specified accuracy, it uses what is called now an iterative, non-adaptive scheme. An approximation and error estimate are computed with $[a, b]$ split into subintervals of equal length. If the approximation is not sufficiently accurate, the number of subintervals is doubled and another try is made. Doubling the number of subintervals allows the program to reuse all previous evaluations of $f(x)$. This is efficient and is even more efficient when coded in MATLAB where the function evaluations are naturally vectorized. However, the approach does not deal well with $f(x)$ that have peaks, discontinuities in low-order derivatives, or moderate singularities at endpoints.

Adaptive quadrature programs break $[a, b]$ into pieces $[\alpha_j, \beta_j]$, approximate the integral over the pieces, and sum to approximate $I(f)$. The idea is to use long subintervals where $f(x)$ is smooth and therefore easy to approximate, and rely on the increased accuracy of the basic formula on short subintervals where the integration is difficult. Adaptive quadrature programs are coded in several ways [7]. An approach that attempts to minimize the number of (scalar) evaluations of $f(x)$ is found in the other piece of mathematical software for oscillatory integrals known to us, namely the AINOS program of Piessens and Branders [9]. It uses a formula based on a high-order Chebyshev expansion. At all times, it has stored a partition of $[a, b]$ and approximate integrals over the various subintervals as well as estimates of the error of these approximations. The approximations and error estimates are summed to get the current approximation to $I(f)$ and an estimate of its error. If the estimated error is bigger than the tolerance specified by the user, the code determines a subinterval with maximum error. This subinterval along with its approximation and error estimate is replaced by the two halves of the interval and corresponding approximations and error estimates. This is very efficient in terms of scalar function evaluations, but it can involve considerable storage and overhead. The error control is quite different in the approach of `osc`. If the estimated error of an approximation on $[\alpha_j, \beta_j]$ is no more than the tolerance times $|(\beta_j - \alpha_j)/(b - a)|$, the result for that subinterval is accepted and the subinterval is of no further interest. If the error is not sufficiently small, the subinterval is placed at the tail of a queue. As we have coded it, when a subinterval is taken from the head of the queue, all the $f(x_j)$ needed to approximate the integral independently over the two halves of the interval are formed in a single-array function evaluation. This is a considerable economy because about 60 values of $f(x)$ are formed in this call. At the same time, all the necessary $\mathrm{e}^{\mathrm{i}\omega x_j}$ are formed in a single call to a built-in function. Because absolute values are used in the error test on subintervals, passing the test on all subintervals implies that the tolerance specified by the user is satisfied. This is much more conservative than the approach of AINOS which takes into account the signs of the errors on subintervals. The approach also requires relatively little storage and overhead. More details about this kind of implementation are found in [7,13].

Although our formula is of modest order, the implementation is adaptive and exploits vectorization to take many samples efficiently in each subinterval. Some examples show how effective the program `osc` is when presented a range of difficulties. Piessens and Branders [9] illustrate their adaptive quadrature program with the family

$$\int_0^1 \frac{\cos(2n\pi x)}{1 + 2\alpha\cos(2\pi x) + \alpha^2}\,\mathrm{d}x = (-1)^n \frac{\alpha^n}{1 - \alpha^2} \quad |\alpha| < 1$$

because $f(x)$ has a peak that becomes sharper as $|\alpha| \to 1$. The most difficult case they consider has $\alpha = 0.9$ and $n = 32$, which corresponds to $\omega \approx 201$. With default scaled absolute error tolerance of 10^{-3}, `osc` used 45 array function evaluations to obtain a result accurate to 1.1×10^{-5}. The integrand of

$$\int_0^{2\pi} x\log(x)\sin(100x)\,\mathrm{d}x \approx -0.1156341422791979 \tag{10}$$

is continuous at $x = 0$, but its first derivative is infinite. With default tolerance, `osc` used 11 array function evaluations to compute a result that is accurate to 1.9×10^{-9}. Exercise 3.30 of [13] considers interpolation of some measurements of the diffusion of chloroform in polystyrene. There is not much data and as a consequence, polynomial and complete cubic spline interpolation do not have the qualitative properties expected from physical considerations. The MATLAB program `pchip` computes a shape-preserving spline that does provide a plausible fit. This spline is a piecewise cubic polynomial function that preserves monotonicity in the data, but it is only C^1. We define $f(x)$ in Equation (1) as this fit to the data of Table 1 and approximate the integral with $\omega = 100$ to show how `osc` does with an $f(x)$ that has jump discontinuities in the second derivative. The fit $f(x)$ is a cubic polynomial on each $[x_j, x_{j+1}]$, so `osc` is *exact* when applied to one of these subintervals. A reference value `trueI` can therefore be obtained by applying the program to each subinterval in turn and adding the results. As it happens, `osc` called upon

Table 1. Data that is fit with a monotonicity-preserving spline $f(x)$ using `pchip`.

x_j	5	7.5	9.9	12.9	13.2	15.1	16.3	16.8
$f(x_j)$	0.0240	0.0437	0.0797	0.1710	0.1990	0.3260	0.8460	0.9720

`quadgk` to do two of these integrations for reasons explained in Section 6, but that program is also exact for cubic polynomials. In this way, we obtained

```
trueI = 6.740593293123156e-003 +6.901108436734398e-003i
```

With tolerance 10^{-3}, the program used only one array function evaluation to obtain a result with a scaled error of 2.4×10^{-4}. It is perhaps worth comment that the data are irregularly spaced, so `osc` is not accidentally applied to subintervals on which it is exact.

To illustrate the control of error, we integrate Equation (9) with $\omega = 100$ using tolerances 10^{-p} for $p = 1, \dots, 8$. For this we need only assign the desired value of the tolerance to the variable `tol` and issue the command

```
Q = osc(@(x) cosh(x),100,0,1,tol);
```

The program produces the same result for tolerances from 10^{-1} through 10^{-6}, namely one array function evaluation produces a result with scaled error of 1.2×10^{-8}. As the tolerance is reduced further, the computation costs more, but `osc` provides a result that is considerably more accurate than required. Specifically, for tolerance 10^{-7}, three evaluations gives a result with scaled error 8.0×10^{-10} and for tolerance 10^{-8}, seven evaluations gives a result with scaled error 5.0×10^{-11}.

6. Small ω

Chase and Fosdick [1,2] discuss the loss of significance that arises when computing coefficients of Filon's method as the product of ω and the length of subintervals becomes small (see also [13]). Piessens and Branders [9] recognize a similar difficulty with their method based on polynomial interpolation at Chebyshev nodes. Iserles and Nørsett [6] point out that the usual asymptotic approximation (3) is not applicable as $\omega \to 0$, but that their Filon formulas generalized to include derivatives are applicable. However, just as with Filon's original method, their formulas must be evaluated carefully to avoid a loss of significance when computing coefficients for small ω. The same kind of difficulty arises with our formula based on the complete cubic spline. Although it is possible to evaluate the formula in a way that preserves significance for small ω, we have preferred something much simpler that is both effective and easily implemented. The difficulties with precision arise when a subinterval is sufficiently short that the integrand is not oscillatory. Accordingly, we use instead a composite Simpson rule to approximate the integral of $f(x)e^{i\omega x}$ on this subinterval. Specifically, if we need to approximate the integral over an interval $[\alpha_j, \beta_j]$ for which $|\omega(\beta_j - \alpha_j)| \le 2\pi$, we use a composite Simpson rule of 29 equally spaced nodes and control the error of the imbedded formula of 15 equally spaced nodes. Piessens and Branders [9] do something similar with a 13-point Kronrod pair.

The integrand of Equation (1) oscillates only a few times in the whole interval if $|\omega(b - a)| \le 20\pi$. Our scheme based on the composite Simpson rule is satisfactory for such integrands, but `quadgk` is *much* more efficient [12], so we simply call it from `osc` to approximate $I(f)$ in this situation.

7. Endpoint singularities

Davis and Rabinowitz [3] consider two examples with moderate singularities in $f(x)$ at an endpoint. Closed formulas like those found in FSER1, AINOS and `osc` evaluate at the ends of subintervals, which makes them inconvenient, at best, in this situation. The program `quadgk` not only has an open formula, but it also changes variable so as to handle well moderate singularities at one or both endpoints. `osc` has an optional logical argument that is used to inform the program of a moderate singularity at one or both ends of $[a, b]$. If the user indicates that singularities are possible, `osc` approximates $I(f)$ in two steps. It uses `quadgk` with half the tolerance to deal with the endpoints and its usual method with half the tolerance to deal with the rest of the interval. The integrand presented to `quadgk` has the form $f(x)e^{i\omega x}$ except that it is set to zero on $[a + \delta, b - \delta]$ for $\delta = 10\pi/|\omega|$. `quadgk` has an option called `Waypoints` that allows a user to specify subintervals of $[a, b]$ where the integrand is relatively smooth. In `osc` this option is used to specify subintervals $[a, a + \delta]$, $[a + \delta, b - \delta]$, $[b - \delta, b]$. `quadgk` is an adaptive code that takes these subintervals as its initial partition of $[a, b]$. This code exploits array operations and vectorization to a much greater degree than the other quadrature programs of MATLAB and for that matter, `osc`. It uses an error control like that of `osc` to decide whether the approximation on a subinterval is good enough, but instead of processing two subintervals at a time, it processes them *all*. In particular, a single array function evaluation is used to approximate the integrand on all three initial subintervals. The formula is exact on the middle interval where the integrand is zero, so that subinterval is of no further interest. The code automatically introduces a transformation to deal with potential singularities at both ends of $[a, b]$, hence in the other two subintervals. These two subintervals are so short that the factor $e^{i\omega x}$ oscillates only a few times on each, hence the method of `quadgk` is very efficient there. This use of the `Waypoints` option and the automatic change of variables make possible a simple and effective treatment of moderate endpoint singularities. The approach is very convenient for users because the program does not ask about which end might have a singularity nor what kind of moderate singularity might be present.

Results for the two examples of Davis and Rabinowitz [3, pp. 65–66] with $\omega = 100$ show how well our approach works. One of the examples is

$$I = \int_0^{2\pi} \log(x)\sin(100x)\,dx.$$

With tolerance 10^{-3}, if we assign the value `true` to the optional input variable `singular`, this integral is approximated by

```
    Q = imag( osc(@(x) log(x),100,0,2*pi,1e-3,singular) );
```

at a *total* cost of seven array function evaluations to get a result with a scaled error of 5.0×10^{-8}. In passing we recall from Section 5 that it is not necessary to account for the singularity in the first derivative of the integrand of Equation (10), but if we do, we can approximate that integral by changing the anonymous function of the MATLAB command just stated to `@(x) x.*log(x)`. This approximation of Equation (10) costs a *total* of four array function evaluations to get a result with scaled error 2.4×10^{-9}. The other example of Davis and Rabinowitz is

$$I = \int_0^{2\pi} \frac{x}{\sqrt{1-(x/2\pi)^2}}\sin(100x)\,dx.$$

A result with a scaled error of 6.4×10^{-8} is obtained with a *total* of eight array function evaluations. An interesting example with algebraic endpoint singularity is constructed from the identity

$$\pi J_0(\omega) = \int_0^{\pi} \cos(\omega\sin(t))\,dt.$$

We cannot apply osc to this integral directly, but if we introduce the variable $x = \sin(t)$ and take into account the change of sign of $\cos(t)$ on $[0, \pi]$, we are led to

$$\pi J_0(\omega) = \int_0^1 \frac{2}{\sqrt{1-x^2}} \cos(\omega x)\,\mathrm{d}x.$$

There is a moderate singularity at $x = 1$, but with $\omega = 100$ and a tolerance of 10^{-3}, osc needs a *total* of three array function evaluations to produce a result with a scaled error of 3.4×10^{-8}.

8. Conclusions

Conventional quadrature programs are ineffective when applied to $\int_a^b f(x)\mathrm{e}^{\mathrm{i}\omega x}\,\mathrm{d}x$ for large ω, but a number of methods have been developed for the task that work well. A new method of order four is developed in this paper. The MATLAB program osc uses an adaptive implementation of this method to deal with $f(x)$ that have peaks and other isolated difficulties. The program resorts to other methods to deal with small ω and with $f(x)$ that have moderate singularities at one or both ends of $[a, b]$. The numerical examples of the paper show that osc is both easy to use and capable of solving efficiently a wide range of problems to moderate accuracy.

References

[1] S.M. Chase and L.D. Fosdick, *An algorithm for Filon quadrature*, Commun. ACM 12 (1969), pp. 453–457.
[2] S.M. Chase and L.D. Fosdick, *Algorithm 353–Filon quadrature*, Commun. ACM 12 (1969), pp. 457–458.
[3] P.J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, 2nd ed., Academic, Orlando, FL, 1984.
[4] G.A. Evans and J.R. Webster, *A comparison of some methods for the evaluation of highly oscillatory integrals*, J. Comput. Appl. Math. 112 (1999), pp. 55–69.
[5] L.N.G. Filon, *On a quadrature formula for trigonometric integrals*, Proc. R. Soc. Edinb. 49 (1928), pp. 38–47.
[6] A. Iserles and S.P. Nørsett, *Efficient quadrature of highly-oscillatory integrals using derivatives*, Proc. R. Soc. A 461 (2005), pp. 1383–1399.
[7] D. Kahaner, *Sources of information on quadrature software*, in *Sources and Development of Mathematical Software*, W. Cowell, ed., Prentice-Hall, Englewood Cliffs, NJ, 1984, pp. 134–164.
[8] MATLAB, The MathWorks, Inc., Natick, MA 01760.
[9] R. Piessens and M. Branders, *Computation of oscillating integrals*, J. Comput. Appl. Math. 1 (1975), pp. 153–164.
[10] R. Piessens, E. deDoncker, C.W. Ueberhuber, and D.K. Kahaner, *Quadpack – A Subroutine Package for Automatic Integration*, Springer, New York, 1983.
[11] L.L. Schumaker, *Spline Functions: Basic Theory*, Cambridge University Press, Cambridge, 2007.
[12] L.F. Shampine, *Vectorized adaptive quadrature in Matlab*, J. Comput. Appl. Math. 211 (2008), pp. 131–140.
[13] L.F. Shampine, R.C. Allen, Jr., and S. Pruess, *Fundamentals of Numerical Computing*, Wiley, New York, 1997.