Numerical integration of functions originating from quantum mechanics

R. Armiento
Department of Physics, Royal Institute of Technology, AlbaNova University Centre, SE-106 91 Stockholm, Sweden

Applications in quantum physics commonly involve large batches of integrals of smooth but very oscillatory functions. The purpose of this work is to benchmark and compare different numerical algorithms for evaluating such integrals. The routines studied include: two from the QUADPACK package based on Gauss-Kronrod quadrature; one routine based on Patterson’s improvements of Gauss-Kronrod quadrature; and two routines that use a non-standard algorithm of applying quadrature-like rules of unrestricted order. The last algorithm has been seen in previous works, but is not in wide-spread use. The present work includes optimized implementations of this algorithm for both serial and parallel computation.

1. BACKGROUND

Applications performing quantum physics based calculations usually involve numerical treatment of ‘wave functions’. These functions originate from the wave-like Schrödinger’s differential equation and are smooth but very oscillatory. Although it is possible for these functions to involve difficult or singular points, the locations of such points relate closely to physical properties of the treated problem, and are assumed to be known in advance. It is not uncommon for applications to calculate large batches of integrals involving such functions, and hence it is of interest to perform these integrations as efficient as possible. The focus of the present work is to compare some implementations for performing such integrations over finite intervals.

Almost all readily available integration routines are based on different kinds of Gauss quadrature rules [1]. A traditional Gauss quadrature rule involves \( m \) evaluations of the integrand and integrates all polynomials of order \( 2m \) exactly. The extensions by Kronrod [2] sacrifice the exact integration of some of the polynomials of highest order for the ability to reuse integrand evaluations from a lower order formula. Furthermore, Patterson has given an algorithm for deriving formulas of increasing orders which reuse all prior integrand evaluations [3].

Application of quadrature formulas of high order on a generic non-polynomial integrand works well only if the integrand is very smooth. In contrast, most general-purpose integration routines put effort in detecting and treating badly behaved functions. This usually means that they are based on quadrature formulas of lower orders, and are less than optimal for the integrands of study in the present work, which are known to be perfectly smooth.

This work will compare and benchmark some readily available integration routines. The following routines will be studied:

DQNG: A routine in the QUADPACK [4] package. This routine successively applies a set of Gauss-Kronrod rules which reuse all prior integrand evaluations. The routines used are of order 21, 43, and 87. If order 87 is not enough, the routine...
gives up and reports an error.

**QUAD**: a routine created by Krogh and Snyder [5], based on a previous routine by Patterson [3]. The routine uses a set of Gauss-Kronrod-Patterson rules of orders 1, 3, 5, 7, 15, 31, 63, 127 and 255. If order 255 is not enough, the routine gives up and reports an error.

**DQAG**: a routine in the QUADPACK [4] package. This routine uses only one Gauss-Kronrod rule (of order selectable between 15, 21, 31, 41, 51 and 61) and adaptively subdivides the interval of integration until the required accuracy is fulfilled.

**TINT and DEFINT**: two routines that are based on a non-standard derivation of quadrature-like rules, applying rules of higher and higher orders until sufficient accuracy is found. DEFINT is available in the JCAM software collection [6]. The TINT routine was developed as a part of a recent work involving the author [7] and has thus not been readily available or thoroughly tested previously. The algorithm and its implementation will be described in the next section.

In this suite, TINT is the only routine that is trivially expandable to apply rules of arbitrary order without doing interval subdivision. However, it would be theoretically possible to create a routine that indefinitely applies successive Gauss-Kronrod-Patterson rules with no interval subdivision, but no such routine has been available to the author. However, such a routine should be rather easy to construct by combining Patterson two works [3] with a lookup table similar to the one used in the TINT routine.

Here follows a list of references to other routines that have not been included in the tests but aim for similar integrands as this work. The list should be of relevance to projects searching for a suitable routine for massive numerical integration of smooth integrals. Most of these and other routines are referenced from GAMS [8], and available from there or from Netlib [9]. 1) The Numerical Algorithms Group (NAG) library [10] includes routines for quadrature, and is available in both a serial and parallel version. The D01AHF routine uses the same Gauss-Kronrod-Patterson rules as QUAD, but also subdivides the interval if the accuracy is not enough, much like DQAG does. Its description also lists a few other adjustments aimed to improve performance and reliability. D01AUF is a routine for parallel integration. 2) IMSL Math and Stat Libraries [11] has two routines, QDAG and QDNG, which are aimed for similar applications as the DQAG and DQNG routines of QUADPACK. 3) The GNU Scientific Library (GSL) [12] is an open source alternative to commercial libraries. However, the quadrature routines are just reimplementations in C of the QUADPACK algorithms. 4) The NMS Numerical library [13] and CMLIB [14] both include a routine DQ1DAX by D. Kahaner that aims at doing efficient numerical integration. 5) the SLATEC library [15] includes the QUADPACK routines but also has two additional routines, DGAUS8 and QNC79 that are aimed at integration of smooth integrals. The first one is based on an adaptive use of a 8-point Legendre-Gauss algorithm and the second one on a 7-point Newton-Cotes quadrature rule. 6) The ACM Toms library [16] includes the QUAD routine included in the tests (as algorithm number 699), but also has some other relevant routines. DQPSRT, algorithm number 691 [17], use Gauss-Kronrod rules for quadrature based on recursive monotone stable formulas.
INTHP, Algorithm 614 [18], is based on a derivation of optimal quadrature points for a certain class of functions. 7) The archive of Harwell subroutine library [19] includes a routine QA04 that “Integrate to specified accuracy using adaptive Gaussian Integration”. 8) IBM:s Engineering and Scientific Subroutine Library (ESSL) [20] includes a set of different quadrature routines. 9) The ParInt [21] research group provides a freely available parallel integration routine for download. 10) W. Gander and W. Gautschi have worked on two routines ADAPTSIM and ADAPTOB [22] to replace the quadrature routines in MATLAB [23] before version 6. However, none of the mentioned routines seems to be as trivial to extend to use rules of arbitrary order as TINT.

2. THE ALGORITHM OF DEFINT AND TINT

Different variations of the algorithms of the TINT and DEFINT routines have been explored through papers of various authors [24; 25]. Specifically, the routine DEFINT was developed by M. Mori and is based on work of M. Mori and H. Takahasi [25]. A related variation of the algorithm was rediscovered independently during the creation of a routine aiming for efficient parallel numerical integration of integrands originating from quantum mechanics. This was done in a recent work involving the author [7] and resulted in the routine TINT. The algorithm will be described in the following.

Integration over any finite range can be substituted into an integration over a range from 0 to 1, so only that case will be discussed here. Consider a well behaved function \( f(x) \) in which we perform an integral substitution, \( x = w(x') \) which fulfill \( w(0) = 0 \) and \( w(1) = 1 \),

\[
\int_0^1 f(x)dx = \int_0^1 f(w(x))w'(x)dx. \tag{1}
\]

where \( w'(x) \) is the derivative of \( w(x) \). Now, consider \( w(x) \) to fulfill the additional requirement that its right derivatives, to any order, equal zero as \( x \to +0 \) and its left derivatives, to any order, equal zero as \( x \to 1 \). In this case the integration of the combination \( f(w(x))w'(x) \) can be seen as an integration of one period of a periodic function, as the function values and all derivatives match at the borders. The main idea here is that for such integrands ordinary trapezoid integration is known to converge rapidly, because of a cancellation of errors. This argument assumes that \( w'(x) \) going to zero in the integration limits also makes \( f(w(x))w'(x) \) go to zero. A sufficient (but not necessary) requirement is that \( f(x) \) is finite in these limits. Similar assumptions are made for the derivatives of \( f(x) \).

A possible choice for \( w(x) \) that fulfills the requirements is

\[
w(x) = \int_0^x ce^{-1/(z-z_2)}dz, \quad c = \left( \int_0^1 e^{-1/(z-z_2)}dz \right)^{-1}, \tag{2}
\]

\[
w'(x) = c e^{-1/(x-x_2)} \tag{3}
\]

Trapezoid integration of the substituted \( f(x) \) can now be recast on a form similar
to a Gaussian quadrature rule:

\[
\int_0^1 f(x) \, dx \approx h \sum_{n=1}^{\lfloor h^{-1} \rfloor} v_n f(x_n), \quad x_n = w(hn), \quad v_n = w'(hn),
\]

(4)

where \( h \) is a chosen step length, and since by construction the integrand goes to zero on the limits of the integration, the two outermost terms have been dropped. For each step length the values of \( v_n \) and \( x_n \) can be pre-calculated with some other simple numerical integration algorithm during the program initialization. The algorithm is now based on reducing \( h \) in iterative steps until the relative difference between results from two consecutive steps is less than some error bound \( \epsilon \). A major benefit inherited from the trapezoid integration is that the number of function evaluations needed for each step can be halved if \( h \) is reduced with a factor of 2 in each step, since the previous computed approximation can be reused.

Despite the fact that Eq. (4) does not include the end points of the interval and thus is formally open, the nature of the function \( w(x) \) brings \( x_1 \) and \( x_{n-1} \) extremely close to 0 and 1. Hence, when implemented with numbers of limited resolution, the formula is effectively a closed one.

The TINT routine uses the \( w(x) \) of Eq. (2), whereas the works of H. Takahasi and M. Mori [25] focus on another transformation called the DE-rule, which consequently is used in the routine DEFINT. The DEFINT routine also has a more refined error estimate than only estimating the error as the relative difference of two consecutive iterations.

3. IMPLEMENTATION

The implementation of TINT in ANSI Fortran 77 [26] is present in the Appendix. The algorithm relies on fixed values of the primitive function of Eq. (3) and the routine uses an initialization subroutine, TINIT, which calculates these values by numerical integration and stores them in a lookup table. These numerical integrations are done by calling the external QUADPACK DQK61 routine which applies a 61 points Gauss-Kronrod rule, which has been observed to give enough accuracy. In this way the weights and abscissae for decreasing step sizes are calculated and put in the lookup table for later use during applications of Eq. (4). The weights and abscissae are stored intermixed in one long array TINTDT to ensure optimal use of the cache memory. To keep track of start and stop points for different step sizes in this array, another small lookup table is used, TINTRG. This saves a few mathematical operations compared to computing the start and stop points each time we use the routine, for the small cost of an array of only a few elements. A further possible optimization which is not done here, is to use the symmetry of Eq. (3) around \( x = 0.5 \) to halve the size of the lookup table. The lookup table currently ends at \( 2^{17} \) interval divisions, however, this can be trivially adjusted through the parameters MAXORD and DTSIZE (the latter should just be set to \( 2^{\text{MAXORD}+1} \)).

Once the lookup table has been initialized any number of calls to the integration routine, TINT, can be performed. This routine is just a straightforward application of the pre-calculated abscissae and weights to the function according to the formula Eq. (4).

For the parallel version of the routines (TINITP, TINTP), some adjustments
addition, integrals whose normalization constant becomes a number of magnitude integrated within the limited refinements used by the routines QNG and QUAD. In set of integrals that only includes functions of a few oscillations which can be values with \( b \) any localized behavior of the routines.

set to 10 and the parameters constants have been chosen to level the difficulty of the integrands. Integrals and primitive functions are tabulated in Table I. All integrands have a free parameter they integrate to exactly 1. The 10 unnormalized integrands and their primitive exponential functions. They will be normalized with a known exact solution, so that As explained, the focus of this work is integration over finite intervals of smooth.

4. BENCHMARKING SERIAL ROUTINES

Table I. Integrands used to benchmark the routines in this work. The primitive functions are used to pre-calculate a normalization constant making the value of the integrals exactly 1. The integrands are all constructed to be heavily oscillatory and descending. Integrals and primitive functions have been produced by taking derivatives of suitable primitive functions. The numerical constants have been chosen to level the difficulty of the integrands.

<table>
<thead>
<tr>
<th>Name</th>
<th>Integrand</th>
<th>Primitive function</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>( e^{-0.01x} \left( 0.01 \cos(0.3ex) + 0.3 \sin(0.3ex) \right) )</td>
<td>( \cos(0.3ex)e^{-0.01x} )</td>
</tr>
<tr>
<td>12</td>
<td>( 0.001(2 \cos(0.001cx^2) - 2 \sin(0.001cx^2)) )</td>
<td>( \sin(0.001cx^2) )</td>
</tr>
<tr>
<td>13</td>
<td>( 0.003(2 \cos(0.003cx^2) - \frac{1+\ln(x)}{(1+\ln(x))} \sin(0.003cx^2)) )</td>
<td>( \sin(0.003cx^2) )</td>
</tr>
<tr>
<td>14</td>
<td>( 0.2c(1+x) \cos(0.2cx) - \sin(0.2cx) )</td>
<td>( \sin(0.2cx) )</td>
</tr>
<tr>
<td>15</td>
<td>( (1+x^2)^{\frac{1}{2}}((1+x^2) \cos(x) \sin(0.05ex) + (0.05e(1+x^2) \cos(0.05ex) - 2x \sin(0.05ex)) \sin(x))) )</td>
<td>( \sin(0.05ex)\sin(x) )</td>
</tr>
<tr>
<td>16</td>
<td>( \frac{\sin(\sqrt{x^2+80cx})}{\sqrt{x^2+80cx}} )</td>
<td>( \cos(\sqrt{1+80cx}) )</td>
</tr>
<tr>
<td>17</td>
<td>( e^{-\sqrt{1+7x}}(0.5e(0.5cx) - \sin(0.5cx)) )</td>
<td>( \sin(0.5cx)e^{-\sqrt{1+7x}} )</td>
</tr>
<tr>
<td>18</td>
<td>( e^{-0.01x}(0.002cx \cos(0.001cx^2) - 0.01 \sin(0.001cx^2)) )</td>
<td>( \sin(0.001cx^2)e^{-0.01x} )</td>
</tr>
<tr>
<td>19</td>
<td>( 2 \cos(x) + 0.002cx^2 \cos(0.001cx^2) + x \sin(x) - 2x \sin(0.001cx^2) )</td>
<td>( \sin(0.001cx^2) )</td>
</tr>
<tr>
<td>20</td>
<td>( 0.05cx \cos(0.05ex) \sin(0.05cx^2) + \sin(0.05ex) )</td>
<td>( \cos(0.05ex) )</td>
</tr>
</tbody>
</table>

have been made. The starting step size is now chosen as to make the integrands evaluations evenly divisible between the parallel nodes. To avoid load balancing issues for integrands which are unevenly hard to evaluate for different abscissae, the values of the lookup table are distributed among the nodes to make all nodes compute values throughout the whole integration interval. The parallelization of the integration routine is then done in the straightforward way of distributing the work of the loops over quadrature coefficients. The implementation in the Appendix has been made with as few deviations from the ANSI Fortran 77 standard as allowed by the MPI standard [27].

In this paper the routine was run with the MINORDER parameter set to 5 to ensure at least 31 evaluations of the integrand. This helped eliminate some unreliability, and it is advisable to use this choice unless the routine is applied to a batch of significantly easier integrands.

4. BENCHMARKING SERIAL ROUTINES

As explained, the focus of this work is integration over finite intervals of smooth oscillatory functions. Such functions will be simulated using sine, cosine and exponential functions. They will be normalized with a known exact solution, so that they integrate to exactly 1. The 10 unnormalized integrands and their primitive functions are tabulated in Table I. All integrands have a free parameter \( c \). We also refer to the limits of the integration as \( a \) to \( b \). Integrations are performed with \( a \) set to 10 and the parameters \( b \) and \( c \) taking on wide range of values to average out any localized behavior of the routines.

The first test is to evaluate all integrals for 250 x 250 evenly spaced parameter values with \( b \) going from 20 to 100 and \( c \) going from 1 to 2. This gives an ‘easy’ set of integrals that only includes functions of a few oscillations which can be integrated within the limited refinements used by the routines QNG and QUAD. In addition, integrals whose normalization constant becomes a number of magnitude
'Easy' tests, 62447 integrals

Fig. 1. (A) 250 x 250 'easy' variations of the integrands of table I integrated by the different integration routines (except for certain troublesome variations). The routines based on usual quadrature rules outperform TINT and DEFINT for this kind of integrals. The required accuracy of DEFINT is adjusted with a factor 0.05 to make its number of successful returns be on the same order of magnitude as other routines. (B) Measurement of reliability for 1000 x 1000 'easy' integrals. Unsuccessful integrals are classified in three categories: 1) US (unreliable success): the routine returns an error or warning, but the returned value still fulfills the accuracy requirements. RF (reliable failure): the routine reports an error or warning and the returned value do not fulfill requested accuracy. 'UF' (unreliable failure): a value not fulfilling the accuracy goal is returned, without any errors or warnings from the routine. This graph is somewhat unfair to DEFINT, since 543 of its unsuccessful integrals come from I10 alone. If these are excluded its reliability is about the same as TINT (i.e., after adjusting its accuracy requirement with a 0.05 factor).

below $10^{-5}$ are removed to avoid too unconditioned integrals. Furthermore all difficult parameter values, for which any routine either reported trouble or did not return a result of sufficient accuracy, are also removed. The motivation behind this is that even just a few such points may lead to many extra integrand evaluations, and since the troublesome points may be different for different routines and are rare this may affect the test unfairly. However, while employing this scheme it was noticed that DEFINT is much more aggressive in its error estimate than the other routines, which lead to the removal of a huge number of points and skewed the test.

To get a more fair comparison, the error bound on DEFINT is therefore increased with a factor of 0.05. This made its number of successful returns be on the same order of magnitude as TINT.

To test the reliability of the routines, a full set of 1000 x 1000 integrals in the easy parameter range is used. The outcome of the routine is classified in one of four categories. The categories are either 'success' or 'failure' depending on whether the routine delivers a result fulfilling the required accuracy; and 'reliable' or 'unreliable' depending on whether warnings issued are correct or unnecessary/missing.

The results are shown in Fig. 1. It is seen that for these easy integrals TINT and DEFINT require about twice as many integrand evaluations as other routines. They are also somewhat less reliable than the other routines.

The next test is done for 'hard' integrals, which excludes DQNG, QUAD and DEFINT, since these routines give up before having done enough integrand evaluations. A set of 50 x 50 parameter values is used, with $b$ going from 100 to 200 and $c$ going from 2 to 25. The test is done for three options for the required accuracy, $10^{-8}$, $10^{-10}$, $10^{-12}$. Again, parameter values that give integrals that are uncon-
The integrands of table I integrated by the different integration routines on an 'hard' grid of 50 x 50 parameter values giving integrands with a huge number of oscillations. It is seen how TINT outperform DQAG for this kind of integrals.

Figure 2. The reliability of the routines on the 'hard' 1000 x 1000 grid of parameter values. Unsuccessful integrals are classified as in Fig. 1b.

conditioned or for which TINT or DQAG have difficulties are excluded. The results are shown in Fig. 2. In these tests which are closer to an actual application, TINT requires significantly less integrand evaluation than DQAG. The reliability for the 'hard' set of parameters is also investigated by running a full set of 1000 x 1000 integrals. The results are shown in Fig. 3.

Figure 4 shows the time spent in each routine per integrand evaluation. This is an attempt to do a fair measurement of the efficiency of the code of the routines. The routines perform mostly equal, which is no surprise since the central part of all routines is a straightforward loop over quadrature points.

5. BENCHMARKING PARALLEL IMPLEMENTATION OF TINT

The parallelized version of TINT is tested in the 'hard' parameter range for 1, 2, 4, 8, and 16 computer nodes. This is done with and without 'load'. The case without load is 300 x 300 of the usual integrands. The case with load is 5 x 5 integrals where the integrand is simulated to be hard to compute by forcing the integrand function to multiply 25000 double precision numbers before returning. A logarithmic execution time graph is shown in Fig. 5.

It is seen that the unloaded version have scalability problems, and already for 4 nodes the communication overhead makes the routine go slower than for 2 nodes. For the loaded case, the scalability becomes much better since a less fraction of the cpu time is spent on handling communications. This means that one usually
Fig. 4. Comparison of execution time per integrand evaluation. The five bars to the left show the results for execution of ‘easy’ integrals, and the three to the right show the results for execution of ‘hard’ integrals.

Fig. 5. A logarithmic executions graph for parallized versions of the routines running on an increasing number of nodes.

don’t want to use the parallel code on integrands that are too simple to compute, but rather on eg. integrands that consist of other integrals or are otherwise time consuming. However, the added load has here been completely evenly distributed over the integrand, which will not be the usual case. The more unbalanced the load is, the worse the scaling will be, since nodes with lighter loads will have to idle-wait for others to complete. The way this is usually handled is by dynamically redistributing the work (i.e., dynamic load balancing).

All data in this paper are from a HP Itanium2 Cluster of rx2600/zx6000 nodes with two 900 Mhz Itanium 2 "McKinley" processors per node and with myrinet as their interconnection (Myricom M3F-PCI XD-2 cards). Some incomplete runs have also been done on other architectures, but no major deviations have been observed from what is seen in the published data.
6. CONCLUSIONS

This work has provided extensive benchmarks for some common integration algorithms and their implementations, in the context of integration of functions originating from quantum mechanics. The results of these benchmarks are useful when implementing applications that perform larger batches of such integrals. The work has also put forward an implementation of an algorithm that is clearly better than the “standard” DQAG for the applications this work focus on.

It seems as when DQAG starts to subdivide the interval of integration it is outperformed by TINT which does not have to resort to any subdivisions. However, since TINT is outperformed by the Gauss-Kronrod-Patterson routines for simple integrands, it is possible that a routine applying these rules indefinitely also would perform significantly better than DQAG does.

The reliability graphs are somewhat hard to interpret. TINT has good reliability when the accuracy requirement is low. However, it also seems that TINT suffers more than DQAG when the requirement is set higher. However, for $10^{-8}$ and $10^{-10}$ DQAG returns many superfluous warnings, probably because both these cases are somewhat on the edge for where the limited numerical precision of the floating point numbers starts to affect the results. It is however surprising that DQAG is slightly more reliable with an accuracy requirement of $10^{-8}$ than $10^{-6}$. In actual applications the unreliability must be tackled with some careful supervision of the routines, since one bad evaluation can destroy much of the work done for other integrals. It is possible to increase the reability of TINT by adjusting the minimum number of integrand evaluations through the MINORD parameter to fit the integrals it is used on.

The simplicity of TINT puts it in a good position for parallel implementation. For integrands that are not too simple or too unbalanced, the provided parallel implementation should be useful.

7. ACKNOWLEDGMENTS

The author wish to acknowledge support from the project ATOMICS at the Swedish research council SSF and from the Göran Gustafsson Foundation. The computer calculations was done on the Lucidor cluster at PDC in Stockholm.

REFERENCES

6. JCAM is a collection of FORTRAN programs published in the Journal of Computational and Applied Mathematics. Routine DEFINT is available for download through a link from GAMS [8].
A. APPENDIX: IMPLEMENTATION TINT

subroutine tint(f,a,b,epsabs,epsrel,minor,d,ier,order)
* ier,order)
c**date written 2004-01-12 (yyyy-mm-dd)
c**revision date 2004-02-18 (yyyy-mm-dd)
c**keywords automatic integrator

c**author rickard armiento, kth, albanova university center,
c kth physics, theory of materials, se-106 91 stockholm,
c sweden
c**purpose the routine approximates \( \int_a^b f \) over \((a,b)\)
c for a smooth integrand \(f\), trying to satisfy absolute and
c relative claims for accuracy
c***references physical review b 66, 165117 (2002)
c***input arguments
c f - double precision
   integrand function f(x). the actual function must
   be declared /external/ in the calling program.
c a - double precision
   lower limit of integration
c b - double precision
   upper limit of integration
c epsabs - double precision
   requested absolute accuracy
c epsrel - double precision
   relative accuracy requested
c minord - integer
   force integration to make at least
   (2**minord)-1 subdivisions

c***output arguments
c result - double precision
   approximation to integral over f from a to b
   result/ should approximate integral within requested
   accuracy.
c abserr - double precision
   estimate of the absolute error
   ier = 0 normal termination. /result/ should
   integrate integral within requested
   accuracy.
c ier - integer
   ier = 1 subroutine stopped as maximum number of
   subdivisions was reached. by increasing
   /maxord/ and /dtsize/ parameters inside
   the code more subdivisions can be used.
c ier = 2*2**order
   investigate the integrand for
c difficulties.
c order - integer
   on return, the number of subintervals
   produced in the subdivision process was
   2*2**order

c***subroutine parameters
   external f
double precision a,abserr,b,epsabs,epsrel,f,result
   integer ier,minord,order
c***adjustable parameters
c if changed, make sure to synchronize throughout file
   maxord: give up after 2**maxord subdivisions
c dtsize: 2*(2**maxord), max size of lookup table
   integer maxord, dtsize
   parameter( maxord = 17 )
   parameter( dtsize = 262144 )
c***common block
   common /cmtint/ tintdt, stdx, tintrg, stord
   double precision tintdt(dtsize), stdx
   integer tintrg(maxord), stord

c***local variables
double precision dx, oldsum
double precision diff
   integer i, startp, endp

c***first executable statement tint
ier = 0
diff = b-a
c do first batch with lowest interval division
oldsum = 0.0d0
endp = tintrg(stord)-2
do 10 i=1,endp,2
   oldsum = oldsum + f(a + diff*(tintdt(i)))*
   * diff*tintdt(i+1)
10 continue
oldsum = oldsum*stdx
order = stord+1
dx = stdx
c begin loop for increasing interval division (orders)
20 startp = tintrg(order-1)
   endp = tintrg(order)-2
c loop over all subdivisions
result = 0.0d0
do 30 i=startp,endp,2
   result = result + f(a + diff*(tintdt(i)))*
   * diff*tintdt(i+1)
30 continue
result = 0.5d0*(oldsum + result*dx)
abserr = abs(result-oldsum)
c exit if accuracy is fulfilled
if( (order .ge. minord) .and. ((abserr .le. epsabs) .or.
   * (abserr .le. epsrel*abs(result) )) ) goto 40
order = order + 1
dx = dx*0.5d0
oldsum = result
c loop if not order .gt. maxorder
if(order .le. maxord) goto 20
 c abnormal exit, accuracy goal not fulfilled
ier = 1
40 return
end

c***function used for substitutions in trapetzoid integration
double precision function subfp(x)
c***subroutine parameters
double precision x
c***first executable statement subfp
subfp = 142.250375777096862d0*exp(-1.0d0/(x-x*x))
return
end

c***subprogram for initializing lookup table
subroutine tinit
   c***adjustable parameters
c if adjusted, make sure to synchronize with subroutines
c loword: do first run with 2**loword subdivisions
   integer loword, maxord, dtsi
   parameter( loword = 3 )
   parameter( maxord = 17 )
   parameter( dtsi = 262144 )
c***common block
   common /cmtint/ tintdt, stdx, tintrg, stord
double precision tintdt(dtsize), stdx
integer tintrg(maxord), stord

C*** local variables
integer n, i, j, order
double precision x, dx, nxtdx, abserr, resabs, resasc

C*** subprograms
external subfp
double precision subfp

C*** first executable statement tinit
stord = loword
stdx = 1.0d0/2**stord
n = 2**stord
dx = stdx
nxtdx = stdx*0.5d0
j = 1

C do first batch with stepsize dx
x = dx
do 110 i=1,n-1,1
   call dqk61(subfp,0.0d0,x,tintdt(j),abserr,resabs,resasc)
   tintdt(j+1) = subfp(x)
   x = x + dx
   j = j + 2
110 continue
C do following batches starting with stepsize
c dx and offset 0.5*dx, and use half
c stepsize each consecutive step
   do 130 order=stord+1,maxord
      tintrg(order-1) = j
      x = nxtdx
      do 120 i=1,n,1
         call dqk61(subfp,0.0d0,x,tintdt(j),abserr,resabs,resasc)
         tintdt(j+1) = subfp(x)
         j = j + 2
         x = x + dx
      120 continue
      n = n*2
      dx = nxtdx
      nxtdx = nxtdx * 0.5d0
   130 continue
   tintrg(maxord) = j
   return
end

B. APPENDIX: IMPLEMENTATION TINTP

subroutine tintp(f,a,b,epsabs,epsrel,minord,result,abserr,
*      ier,order)
implicit none
C*** date written 2004-01-12 (yyyy-mm-dd)
C*** revision date 2004-02-18 (yyyy-mm-dd)
C*** keywords automatic integrator
C*** author rickard armiento, kth, albanova university center,
c
     kth physics, theory of materials, se-106 91 stockholm,
c
     sweden
C*** purpose the routine approximates \int_a^b f(x) dx
C for a smooth integrand f, trying to satisfy absolute and
relative claims for accuracy

References


Input arguments

f - double precision
 integrand function f(x). The actual function must
 be declared /external/ in the calling program.

a - double precision
 lower limit of integration

b - double precision
 upper limit of integration

epsabs - double precision
 requested absolute accuracy

epsrel - double precision
 relative accuracy requested

minord - integer
 force integration to make at least

(result) = double precision
 approximation to integral over f from a to b

abserr - double precision
 estimate of the absolute error

ier - integer
 ier = 0 normal termination. /result/ should
 approximate integral within requested
 accuracy.

ier = 1 subroutine stopped as maximum number of
 subdivisions was reached. By increasing
 /maxord/ and /dtsize/ parameters inside
 the code more subdivisions can be used.
 However, it may also be advisable to
 investigate the integrand for
 difficulties.

order - integer
 on return, the number of subintervals
 produced in the subdivision process was

2*2**order

Output arguments

result = double precision
 approximation to integral over f from a to b

abserr = double precision
 estimate of the absolute error

ier = integer
 ier = 0 normal termination. /result/ should
 approximate integral within requested
 accuracy.

ier = 1 subroutine stopped as maximum number of
 subdivisions was reached. By increasing
 /maxord/ and /dtsize/ parameters inside
 the code more subdivisions can be used.
 However, it may also be advisable to
 investigate the integrand for
 difficulties.

order = integer
 on return, the number of subintervals
 produced in the subdivision process was

2*2**order

Subroutine parameters

external f
 double precision a, abserr, b, epsabs, epsrel, f, result
 integer ier, minord, order

Include files

include "mpif.h"

Adjustable parameters

If changed, make sure to synchronize throughout file

maxord: give up after 2**maxord subdivisions

dtsize: 2*(2**maxord), max size of lookup table

common /cmtinp/ tintdt, stdx, tintrg, stord, size, rank, commid

double precision tintdt(dtsize), stdx

integer tintrg(maxord), stord, size, rank, commid

Local variables
double precision dx, oldsum, psun
double precision diff
integer i, startp, endp, mpierr
c***first executable statement tintp
ier = 0
diff = b-a
c do first batch with lowest interval division
psun = 0.0d0
dendp = tintrg(stord)-2
do 10 i=1,endp,2
   psun = psun + f(a + diff*(tintdt(i)))*
   * diff*tintdt(i+1)
10 continue
call mpi_allreduce(psum,oldsum,1,mpi_double_precision,
   * mpi_sum,commid,mpierr)
oldsum = oldsum*stdx
order = stord+1
dx = stdx
c begin loop for increasing interval division (orders)
20 startp = tintrg(order-1)
   endp = tintrg(order)-2
c loop over all subdivisions
psun = 0.0d0
   do 30 i=startp,endp,2
      psun = psun + f(a + diff*(tintdt(i)))*
      * diff*tintdt(i+1)
30 continue
call mpi_allreduce(psum,result,1,MPI_DOUBLE_PRECISION,MPI_SUM,
   * commid,mpierr)
result = 0.5d0*(oldsum + result*dx)
abserr = abs(result-oldsum)
c exit if accuracy is fulfilled
   if( (order .ge. minord) .and. ((abserr .le. epsabs) .or.
      * (abserr .le. epsrel*abs(result) )) ) goto 40
      order = order + 1
      dx = dx*0.5d0
      oldsum = result
c loop if not order .gt. maxorder
   if(order .le. maxord) goto 20
c abnormal exit, accuracy goal not fulfilled
   ier = 1
40 return
c***function used for substitutions in trapetzoid integra
tion
double precision function subfpp(x)
c***subroutine parameters
double precision x
c***first executable statement subfp
subfpp = 142.2503757777058682d0*exp(-1.0d0/(x-x*x))
return
c***subprogram for initializing lookup table
subroutine tinitp(cid)
implicit none
integer cid
c***adjustable parameters
c if adjusted, make sure to synchronize with subroutines
c loword: do first run with loword*(nbr of nodes) subdivisions
    integer loword, maxord, dtsize
    parameter( loword = 1 )
    parameter( maxord = 17 )
    parameter( dtsize = 262144 )
c***common block
    common /cmtinp/ tintdt, stdx, tintrg, stord, size, rank, commid
    double precision tintdt(dtsize), stdx
    integer tintrg(maxord), stord, size, rank, commid

c***local variables
    integer n, i, j, order, mpierr
    double precision x, dx, offset, abserr, resabs, resasc

c***subprograms
    external subfpp
    double precision subfpp

c***first executable statement tinit
    commid = cid
    call mpi_comm_size(cid,size,mpierr)
    call mpi_comm_rank(cid,rank,mpierr)
    if((size .gt. 1) .or. (loword .gt. 1)) then
        stord = loword
    else
        stord = 2
    endif
    stdx = 1.0d0/(size*stord)
    n = stord
    dx = stdx*size
    j = 1
c do first batch with stepsize dx; the first step on the first node (rank0)
c is skipped unbalancing the load somewhat this first step, to make sure
c following steps are equally distributed.
    x = stdx*rank
do 110 i=1,stord
    if((rank+i) .gt. 1) then
        call dqk61(subfpp,0.0d0,x,tintdt(j),abserr,resabs,resasc)
        tintdt(j+1) = subfpp(x)
        j = j + 2
    endif
    x = x + dx
 110 continue
c do following batches starting with stepsize
c dx and offset 0.5*dx, and use half
c stepsize each consecutive step
    offset = stdx*0.5d0
    order = stord
120 tintrg(order) = j
    order = order + 1
    x = offset*(2*rank+1)
do 130 i=1,n,1
    call dqk61(subfpp,0.0d0,x,tintdt(j),abserr,resabs,resasc)
    tintdt(j+1) = subfpp(x)
    j = j + 2
    x = x + dx
130  continue
    n = n*2
    dx = dx * 0.5d0
    offset = offset * 0.5d0
    if(order .lt. maxord) goto 120
    tintrg(order) = j
    return
end