DRAFT DRAFT ! Numerical quadrature in a symbolic/numeric setting

Richard Fateman

October 15, 2008

Abstract

One of the advantages of expressing a numerical integration (or "quadrature") program in a computer algebra system (CAS) setting is the availability of arbitrary-precision floating-point calculations. You can then write a single program to extend the computation so as to find a result within nearly any user-specified error bound. Meeting that requirement may force intermediate computations to be done in a much higher precision and at higher cost than ordinary machine double-float arithmetic. To have any chance of success you need at least one quadrature algorithm that can run at arbitrary working precision. This is not especially difficult, and we describe one. We speculate on how one can use it, but do not attempt to provide a universal solution to meeting user-specified error bounds.

1 Numerical Integration or "quadrature"

Quadrature or numerical integration has historically been among the most intensively studied problems in numerical analysis. Excellent programs that provide effective solutions for many problems are in major numerical libraries such as quadpack (available through netlib.org). These libraries provide fixed-precision computations (for example, double-float), and clever routines using different mathematical ideas and heuristics. For many users these are generally satisfactory. Sometimes, however, results from these programs are unsatisfactory: one hopes that in this case the program will signal that the requested accuracy has (probably) not been achieved. This often happens when the function being integrated is not well-behaved, or when sufficient accuracy simply cannot be achieved within the bounds of resources (especially machine precision) requested in the answer.

Note that in a CAS environment, the designers of the system may have included clever definite integrations programs in the package, and this symbolic approach can provide additional insight, especially when the integrand includes symbolic parameters. However, for a univariate integrand, the result is ultimately equivalent to a number. If it is irrational or transcendental, it cannot be expressed exactly as a floating-point number, but sometimes an approximate answer is not only satisfactory, but the symbolic result is only a step towards getting that answer. In such a case, sufficiently careful evaluation of a quadrature formula can be faster and more accurate than finding the exact symbolic definite integral and then evaluating that answer to a number [8].

Given the possibility of using (software-based) arbitrary-precision floating-point arithmetic, we can run programs for meeting a pre-specified error bound (at least heuristically), given that we are willing to pay the cost in computation time and memory space. Three such schemes are given by the programs by Bailey, Jeyabalan and Li [1]. Their paper uses a Fortran and C-based library program to provide high-precision solutions. They show that in some cases the numerical result can be used to deduce an exact expression in terms of symbolic constants for the definite integral.

Any programs which rely (as do these three) on sampling the unknown function at some deterministic set of points, can be fooled, but such poorly-behaved functions tend to be just those that are deliberately concocted in some adversarial fashion, towards quadrature programs.

In any case, the approach taken in this paper is to consider how to incorporate arbitrary-precision quadrature in a computer algebra system: given a "generic" scheme for quadrature (in particular we chose Gaussian quadrature with Legendre polynomials for the interval from -1 to 1). With this tool we provide the ability for refining the result in two directions, reducing round-off and reducing truncation error of the method. It is also possible to re-scale and translate this integration rule to another finite interval. (Rescaling for an infinite interval is also possible, but then another basis, Hermite polynomials, would be preferable.). We also make a minor excursion into the so-called Sinh-Tanh integration method, which is also suitable for extended precision [1].

2   Two ways to gain accuracy

First, note that Gaussian/Legendre quadrature of order n requires knowing a set of function evaluations at a set of abscissa points corresponding to the zeros of Legendre polynomials of degree n. Then each value is multiplied by a particular weight and their sum is the result. These formulas reduce the number of function evaluations necessary for a given accuracy compared to (say) the simpler Simpson's rule, which uses about twice the number of evaluations. An n point Gaussian quadrature rule would be exact for any polynomial of degree no more than $2n - 1$ if the arithmetic were exact[1].

What are the sources of floating-point error, then?

- approximating the zeros of the Legendre polynomials, $\{z_i\}$

- computing the weights, $\{w_i\}$

- computing the function values $\{f_i\} = \{f(z_i)\}$ and

- adding together the products $\{w_i f_i\}$.

These errors can be decreased by increasing the floating-point working precision. This provides more accurate values for $\{z_i\}$, $\{w_i\}$, $\{f_i\}$, and more accurate accumulation.

We can also consider reducing the error (truncation error) by changing the order of the formula: that is, we can use a higher-degree Legendre polynomial, and sample the function at more points. Another way of saying this is that we are assuming that the function is adequately modeled by a polynomial of degree $2n - 1$ for an n-term rule. We can increase n. Or we can divide the interval into two or more sub-intervals and add the integrals up: this, on average, halves the spacing between sample points. Using a formula of order $2n$ reduces the error faster than using the order n formula over two partitions, however.

Either of these changes can be expected to reduce the absolute error of the result. If, after several refinements of the computation we seem to be getting nearly the same numerical result, and we have reason to believe that the internal sources of error (principally accurate function evaluation) are under control, we can heuristically conclude that we have a good approximation to the integral.

There are other ways of trying to improve the result: in particular we can subdivide the range of integration so that more points are used in areas where the function changes rapidly. We can conversely save time by devoting fewer points where the function is small or smooth. Kahan [8] shows that conceptually one can shift some of the burden to the user, or the user-provided function. If one can compute the (non-negative) uncertainty $\{u_i\}$ in each of the $\{f_i\}$, then bounds on the floating-point error in evaluating the formula for an integral can be computing by integrating separately, using the values $\{u_i\}$. If that bound is acceptable, (heuristically) we posit that the answer is not contaminated by arithmetic errors. (It may still be inaccurate

---

[1]which it isn't.

because we have not sampled at enough points.) In a sense one is integrating the function defined by the width of a ribbon of points $\{f_i \pm u_i\}$ which enclose the actual function. How then to compute the uncertainty of a function value: by computing it in two different precisions. Compare $F_P = f(x)$ computed in precision P and then F_{P+k} in substantially higher precision $P + k$. The absolute value of the difference in their values can be used as the uncertainly in F_P. A brief program for this is provided in the Appendix. Given that one has some confidence that a formula is being evaluated without significant loss to floating-point, one can then (say) double the order of the formula and see if the resulting change is less than some requested error bound. If so, the result can be delivered.

There are other techniques that could be considered, where the error bound in a formula can be related to some derivative of f. Is this available? We are, after all, using a computer algebra system!

3   Implementation details

The abscissae and weights do not depend on the function being integrated. These items depend on the order of the rule (the degree of the Legendre polynomial), and on the floating-point precision. This data can consequently be pre-computed, or at least remembered after being computed during a particular computer run. This typically saves considerable time after the first run at a given size and precision. For a quick way to compute these items, see the paper by Golub and Welsch [7]. The method we use is based on the Bailey et al [1] program, which was easy to rewrite and debug in Lisp. We subsequently rewrote it entirely in the Macsyma language (displayed in the Appendix.)

An improvement not mentioned by Bailey et al but expressed in the code, is to take advantage of the fact that the zeros and the weights are symmetric around zero. Thus only half the storage is needed, and the formula is only slightly complicated to account for whether the degree of the Legendre polynomial is even or odd.

These programs are implemented in five ways, at the moment.

1. Using a generic arithmetic package, we have a single program (that is, a main subroutine and the ones it calls), that can be run in any precision: machine double, quad-double, MPFR (arbitrary precision). This is easy to debug but uses more memory and time than more specialized versions than the next three versions whose programming is described below.

2. We have a more specialized Lisp version that run using QD (quad-double) arithmetic. This is high, but not variable precision, but runs at essentially full QD arithmetic speed. It requires that the integrand be expressed in Lisp and run in a QD context.

3. We have an MPFR-only version, using this popular and efficient free package (http://mpfr.org). This is different from the generic version principally in that it has some additional declarations, "`with-temps`" around some calculations, to advise the Lisp compiler to emit efficient code. The compiler then macro-expands certain constructions so that temporary subexpressions can be overwritten in place, rather than allocated/deallocated repeatedly. It also uses "unwrapped" direct calls to the MPFR library, knowing that type constraints have been checked at compile time. The savings in passing the underlying structures means that there is essentially no overhead in calling the MPFR library programs from Lisp compared to (say) C++. We see a factor of three speedup over generic arithmetic if the precision 1000 bits.

4. We have another arbitrary precision version using the `bfloat` package used in Maxima and Macsyma. The bfloat package is an old (we wrote it circa 1974) library not as carefully tuned for speed as MPFR. It is written entirely in Lisp (rather than MPFR's use of C and cpu-specific assembler). Its advantage is primarily the ease with which it can be called from other programs in a variety of Lisps without worrying about compatibility of foreign function calls (into the MPFR library). For example, it runs using Gnu Common Lisp or Allegro Common Lisp, or the commercial Macsyma without change.

3

5. We have a version entirely in the Macsyma language. This is the slowest by virtue of running through an interpreter and also because it omits, for the sake of simplicity, some of the tricks used in the other code. For the reader more familiar with an Algol-style presentation than Lisp, this is the easiest to read.

4   Different ranges

The Gaussian integration routine `gaussunit` integrates from -1 to 1. A change of variables makes it possible to run between any two finite number endpoints. Another change of variables allows for infinite endpoints as well, though a user would be better advised to consider another set of orthonormal polynomials for an infinite interval.

```
/* gaussab(g,lo,hi,h) integrates the  function g(x)
         from x=lo to x=hi using n points and fpprec */

gaussab(%hh,lo,hi,n):=
  block([a:(hi-lo)/2, b:(hi+lo)/2], local(%zz),
    define (%zz(x),%hh(a*x+b)), /* transformed function */
   a* gaussunit(%zz,n))
```

5   Why are we so queasy about controlling error?

Why are we so queasy about setting out quadrature programs including accuracy requirements? Why do we suggest that tests are only heuristic-based? Here's why: One can fool any quadrature program into giving a wrong answer, assuming the only information it has is the value of a function at some set of points. The reason is that the values in between those points could be anything. So the answer may be arbitrarily different. [8].

Another objection is that for some problems, the answer is exactly zero. If the objective is to find a result that is "correct to 10 decimals", that is, a relative error less than 10^{-10}, any non-zero number, regardless of how small, will not satisfy this requirement: it has a leading digit different from 0. Requiring that the quadrature program return exactly zero is a difficult requirement. Finding absolute error bounds is more plausible.

Indeed, providing a calculation using interval arithmetic could provide an interval bound on the integral; in fact something like the "rectangular rule" could be used with only one evaluation: given $v = f([a, b]$ one can compute $(b - a) * [\min(v), \max(v)]$ for a generally crude (i.e. overly wide) interval result: this is guaranteed to contain the result. Note that this is usually not adequately precise, and it also requires information not given by point-evaluation of a function; rather, it requires enough information about f to allow interval evaluation. If one uses interval evaluation for f, it is possible to do the Gaussian integration with intervals as well. Instead of evaluating at z_i one could evaluate between the midpoints of the two adjacent gaps: halfway between z_{i-1} and z_i, and halfway between z_i and z_{i+1}.

6   Sinh-Tanh Integration

Another formula, discussed by Bailey et al [1], looks something like this as a Macsyma program:

```
quadts(%%g,n):= /* Tanh/Sinh method, integral of g from -1 to 1, 2n+1 points. */
 block([sum:0, piby2: bfloat(%pi/2), h:4/(n+2), hj, u1,u2, weight],
  for j from -n thru n do
```

```
   (hj: bfloat(j*h),
    u1: piby2*cosh(hj),
    u2: piby2*sinh(hj),
    weight:u1/cosh(u2)^2,
    sum: sum + weight*%%g(bfloat( tanh(u2)))),
    h*sum)$
```

By setting the global floating-point precision, this can be calculated to any precision at all; $2n+1$ points are used. The computation of weights and abscissas can be reduced by about half by reformulating the program slightly, and the sinh/cosh values can be computed more efficiently by computing an exp and arranging the values appropriately. Both changes are made in the next version:

```
/*saving time by using exp rather than sinh, cosh */
 quadts(%%g,n):= /* Tanh/Sinh method, integral of g from -1 to 1 */
 block([sum:0, piby2: bfloat(%pi/2), h:4/n, t2,t3,t4,u1,u2, weight, ab],
   sum:piby2*%%g(0),
   he:bfloat(exp(h)),
   t2:1,
  for j from 1 thru n do
  ( t2:he*t2, /*exp(h*j); watch for roundoff.. */
   u1: piby2*(t2+1/t2)/2,   /*cosh */
   u2: piby2*(t2-1/t2)/2,   /*sinh */
   t3: exp(u2),
   t4:(t3+1/t3)/2,          /* cosh(u2) */
   t3:(t3-1/t3)/2,          /* sinh(u2) */
   weight:u1/(t4*t4),
   ab: t3/t4,               /*tanh(u2) */
   sum: sum + weight*(%%g(ab)+%%g(-ab))),
   h*sum)$
```

See Bailey et al [1] for guidance on how to elaborate on this scheme so as to make it possible to re-use half the points, in going from one subdivision to another: this can be done by systematically doubling the count of points (halving the step size).

Here we observe that using the `uncert` functionality we can use the same program to compute both the integral and the integral's error. Note that this depends on Macsyma being perfectly happy to perform arithmetic componentwise on lists.

```
/* return a list of the approximation and the fp error */

 quadts_e(%ggg,n):=
  quadts(lambda([%z],uncert(%ggg,[%z])),n)$
```

7   How to Use

It is our intention to just sketch how these programs could be used for (heuristically) assured results by a simple example.

Consider integrating e^x from -1 to 1. Say you want 80 decimal digits of the answer. Assuming we are not in a great rush for the answer we can be generous and consider performing the calculation in

100 digit arithmetic. Computing `a10:  gaussunit_e(exp,10)` provides an answer that is about 2.35, and whose roundoff error is estimated by the program to be less than 10^{-101}. Based on this estimate it looks like we chosen sufficient precision to evaluate this formula. Now try `a20:  gaussunit_e(exp,20)` `a40: gaussunit_e(exp,40)` and `a80:  gaussunit_e(exp,80)`. The difference in value between `a10` and `a20` is about 10^{-24}.

The difference in value between `a20` and `a40` is about 10^{-60}.

The difference in value between `a40` and `a80` is about 10^{-99}.

The difference in value between `a40` or `a80` and the correct $e - 1/e$ is about 10^{-99}.

8 Less conventional uses of quadrature

As indicated earlier, quadrature is a well-studied area, but the possibility of high- or controlled- precision quadrature lends a twist to the applications.

Bailey et al [1] describe using high-precision quadrature to deduce closed forms for integrals, using PSLQ (integer relation detection) programs.

Here is another simple application: let us say that you suspect that an elaborate expression, say involving trigonometric or special functions, is in fact a polynomial of degree n or less in some range of interest. Compute its integral with a formula of order at least $(n - 1)/2$, and with a formula of higher order. Since the Gaussian integral of order k will be exact for degree $2k - 1$ or lower, the two results should be identical (except for floating-point roundoff, which can be bounded by our uncertainty trick.) As a simple example, this can be used to deduce that $\cos(10\arccos x))$ in the range [-1,1] is a polynomial of degree 10, a result known to fans of Chebyshev polynomials.

9 Further Notes

Given that the function f being integrated may be available in some symbolic form (namely as a formula) it may be prudent to try some analysis concerning (say) symmetry, and removal of easily (symbolically) integrable components, especially if they contribute to numerical difficulties. These topics of pre-processing definite integrals have been addressed in various ways by papers by Davenport, Dupee, Fateman, Geddes [3, 4, 5, 6].

The Macsyma system has historically contained a quadrature program `romberg` and a big-float version of this, `bromberg` initially written (by this author) between 1970-74. While satisfactory for many tasks, these programs have been found to be relatively inefficient and unduly sensitive to difficult integrands, and certainly missing the more comprehensive treatment that has appeared in Mathematica or Maple [6]. What to do? Recently Raymond Toy imported into Maxima a suite of high-quality Fortran numerical double-float routines for integration; these suffer from an overly elaborate interface, and are limited to double-float precision. The programs indicated here in this paper are potentially more efficient than `bromberg`, and probably interface better with the underlying Lisp. The MPFR routines, if they are used, and to the extent that they represent an improvement on the bigfloat facilities in Maxima's Lisp, may also contribute to a speedup, especially at extravagantly high precision. (Our experiments with MPFR and quadrature in Allegro Common Lisp suggest a speedup of 4 even for small problems, over Gnu Common Lisp/ Maxima/ bigfloats on a Pentium 4 computer. Larger problems will probably show more speedup.)

10 Conclusion

Free and accessible routines for arbitrary-precision Gaussian quadrature as well as the double-exponential Sinh/Tanh quadrature are provided here and can be utilized to build systems for numerical integration tasks

when requirements exceed those that can be met with scientific subroutine library double-precision routines. Allowing for the notion of computing the error (based on function uncertainty) and essentially unlimited precision, one can build a tool that incrementally increases precision to reduce uncertainty, and thereby reduced the estimate of the error in the arithmetic processing. By incrementally increasing the number of sample points or splitting the interval of integration into sub-pieces, an adaptive method can also be developed so as to reduce the (estimated) overall error, by squeezing down the errors on the separate parts through high precision or more subdivision.

The details for these higher-level programs can be developed in the very high-level languages provided to users.

References

[1] David Bailey, Karthik Jeyabalan and Xiaoye S. Li, "A Comparison of Three High-Precision Quadrature Schemes," `http://crd.lbl.gov/~xiaoye/quadrature.pdf`.

[2] Richard Fateman, Quadrature Programs `http://www.cs.berkeley.edu/~fateman/generic/quad-ga.lisp` and `http://www.cs.berkeley.edu/~fateman/generic/quad-maxima.lisp`.

[3] James Davenport, The Difficulties of Definite Integration, `http://www-calfor.lip6.fr/~rr/Calculemus03/davenport.p`

[4] Brian J. Dupee and James H. Davenport, An intelligent interface to numerical routines. In DISCO'96: Design and Implementation of Symbolic Computation Systems (Karlsruhe, 1996), J. Calmet and J. Limongelli, Eds., vol. 1128 of Lecture Notes in Computer Science, Springer Verlag, Berlin, pp. 252-262, 1996.

[5] Richard Fateman, Computer algebra and numerical integration Proceedings of the fourth ACM symposium on Symbolic and algebraic computation, (1981) Pages: 228 - 232 ISBN:0-89791-047-8

[6] K.O. Geddes Numerical integration in a symbolic context Proceedings of the Fifth ACM symposium on Symbolic and algebraic computation, (1986) Pages: 185 - 191 ISBN:0-89791-199-7

[7] Gene H. Golub, John H. Welsch, "Calculation of Gauss Quadrature Rules," Math. Comp. 23, no 106 (Arpil, 1969) pp 221-230.

[8] W. Kahan, "Handheld Calculator Evaluates Integrals," `http://www.cs.berkeley.edu/~wkahan/Math128/INTGTkey.pdf`

11 Appendix: the Macsyma/Maxima Program

The Macsyma language program is somewhat wasteful when compared to the Lisp version, since the Lisp version can be compiled to direct calls to arbitrary precision floating-point routines; this version has to do dynamic run-time type checking, and does not optimize away the allocation and deallocation of re-usable objects. Most of this extra overhead is in the calculation of the (potentially) pre-computed tables, though. Once they are computed, the Lisp and Macsyma versions should be similar in speed: most of the time is spent on evaluating the integrand function. (Maxima is the name of the open-source version of Macsyma.)

```
(
/* Integrate function over n points from -1 to 1. Compute
abscissas and weights if not already available. Precision
is that of the (global) setting for fpprec. */

gaussunit(%ggg,n):= gaussunit1(%ggg,n,ab_and_wts[n,fpprec]),
```

```
gaussunit1(%%g,n,aw):= /* the function summing the terms */
block([sum:0],
 map(lambda([%%a,%%w],sum:sum+%%w*(%%g(%%a)+%%g(-%%a))),aw[1],aw[2]),
 sum+ (if oddp(n) then -%%g(0)*aw[2][1] else 0 )),

/* Integrate function from x=lo to x=hi using n points and fpprec */
gaussab(%hh,lo,hi,n):=
  block([a:(hi-lo)/2, b:(hi+lo)/2], local(%zz),
     define (%zz(x),%hh(a*x+b)), /* transformed function */
    a* gaussunit(%zz,n)),

/* Background subroutines needed for Gauss integration */

legenpd(k,x):=  /* return P[k](x) and P'[k](x) */
if (k=0) then [1,0]
  else if (k=1) then [x,1]
  else
 block([t0:1,t1:x,ans:0],
 for i:2 thru k do
    (ans: ((2*i-1)*x*t1 -(i-1)*t0)/i,
     t0:t1,
     t1:ans),
 [t1, k*(x*t1-t0)/(x^2-1)]),

legenp(k,x):=  /* return P[k](x) */
if (k=0) then 1
  else if (k=1) then x
  else
 block([t0:1,t1:x,ans:0],
 for i:2 thru k do
    (ans: ((2*i-1)*x*t1 -(i-1)*t0)/i,
     t0:t1,
     t1:ans),
t1),

/* return a pair of lists of abscissae and weights */

ab_and_wts[n, fpprec]:=
block([a:0,v:0.0d0,np1:n+1,nph:1.0d0/(n+1/2),halfn:floor(n/2),
     val,deriv,abscissae:[],weights:[],float2bf:true],
 for i:0 thru halfn-1 do
   (v:cos(?pi*((i+3/4)*nph)), /*an approx zero of legendre-p[n] */
    /*refine it by Newton iteration */
      v:bfloat(v),
      for k:0 thru ceiling(log(fpprec)/log(2)) do /* should be enough for full prec. */
      ([val,deriv]:legenpd(n,v),
       v:v-val/deriv),
   push(v,abscissae),
```

```
    push(legenwt(n+1,v),weights)),
/*push(legendre_pdwt(n+1,v),weights)),*/
    if oddp(n) then (push(0,abscissae),
                     push(2-2*apply("+",weights), weights)),
   [abscissae,weights]),

/* The weight at x=root of _P[k]:   w[k]:=  -2/ ( (k+1)* P'[k](x)*P[k+1](x)).
  call on k+1, and compute compute -2/(k*P'[k-1]*P[k]) */

legenwt(k,x):=
 block([t0:1,t1:x,ans:0,pkp1,dpk],
   for i:2 thru k-1 do
    (ans: ((2*i-1)*x*t1 -(i-1)*t0)/i,
     t0:t1,
     t1:ans),
   pk:((2*k-1)*x*t1 -(k-1)*t0)/k, /*P[k](x)*/
   dpkm1: (k-1)*(x*t1-t0)/(x^2-1),  /*P'[k-1](x)*/
 -2/(k*dpkm1*pk)),

/*UNCERT: a Macsyma program to provide value of function f and
uncertainty, heuristic.  uncert(f,v) evaluates function f at (in
general, vector) point v at some floating-point precision somewhat in
excess of current setting of fpprec.  It returns a list of two items,
[y,u], where y is approximate value of f(v), and u = (nonnegative)
uncertainly in the provided value y. Both y and u are bigfloats. Some
functions may be devious enough as to mislead this calculation, but
this should be exceedingly rare.

Example.
[q(x):=1/(asin(atan(x))-atan(asin(x))),
uncert(q,[1/10000]),
bfloat(q(1/10000))];

Try the above variously with  fpprec:20,  fpprec:100 */

bfapply(%fun,%args,fpprec):= apply(%fun,map(bfloat,%args)),

 uncert(%fun,%args):=
  block([%ll, %hh,%dd,oldprec:fpprec],
        %ll: bfapply(%fun,%args,fpprec),
fpprec: fpprec+10,
        %hh: bfapply(%fun,%args,fpprec),
        %dd: abs(%hh-%ll),
fpprec: oldprec,
   [bfloat(%hh),  bfloat(%dd)]),

/* putting this together with quadrature */

 gaussunit_e(%ggg,n):=   /*with error */
```

```
    gaussunit1(lambda([%z],uncert(%ggg,[%z])),
                 n,ab_and_wts[n,fpprec]),

gaussab_e(%hh,lo,hi,n):=
  block([a:(hi-lo)/2, b:(hi+lo)/2], local(%zz),
     define (%zz(x),%hh(a*x+b)),
    a* gaussunit_e(%zz,n)),

/*Tanh/Sinh method, at least the basics. See the BJL paper and its code for
arranging phases so that the previous phase's result is updated, going from
n points to 2n to 4n etc points. */

quadtsN(%%g,n):= /* Naive Tanh/Sinh method, integral of g from -1 to 1, 2n+1 points. */
 block([sum:0, piby2: bfloat(%pi/2), h:4/(n+2), hj, u1,u2, weight],
  for j from -n thru n do
  (hj: bfloat(j*h),
   u1: piby2*cosh(hj),
   u2: piby2*sinh(hj),
   weight:u1/cosh(u2)^2,
   sum: sum + weight*%%g(bfloat( tanh(u2)))),
   h*sum),

 quadts(%%g,n):= /* Tanh/Sinh method, integral of g from -1 to 1 */
 block([sum:0, piby2: bfloat(%pi/2), h:4/n, hj, u1,u2, weight, ab],
   sum:piby2*%%g(0),
  for j from 1 thru n do
  (hj: bfloat(j*h),
   t2:exp(hj),
   u1: piby2*(t2+1/t2)/2,   /*cosh */
   u2: piby2*(t2-1/t2)/2,   /*sinh */
   t3: exp(u2),
   t4:(t3+1/t3)/2,          /* cosh(u2) */
   t3:(t3-1/t3)/2,          /* sinh(u2) */
   weight:u1/(t4*t4),
   ab: t3/t4,               /*tanh(u2) */
   sum: sum + weight*(%%g(ab)+%%g(-ab))),
   h*sum),

/* return a list of the TS approximation and the fp error */

 quadts_e(%ggg,n):=
  quadts(lambda([%z],uncert(%ggg,[%z])),n)
)
$
```