Improving the Double Exponential Quadrature Tanh-Sinh, Sinh-Sinh and Exp-Sinh Formulas

Dr. Robert A. van Engelen, Genivia Labs

draft March 29, 2021, final June 27, 2021, updated October 12, 2021 / April 30, 2022

Introduction

Tanh-Sinh quadrature is a method for numerical integration introduced by Hidetoshi Takahashi and Masatake Mori [1]. The method uses the *tanh* and *sinh* hyperbolic functions in a change of variable to transform the (−1, +1) open interval of the integral to an open interval on the entire real line (−∞,+∞). Singularities at one or both endpoints of the (−1, +1) interval are mapped to the (−∞,+∞) endpoints of the transformed interval, forcing the endpoint singularities to vanish. This makes the method quite insensitive to endpoint behavior, resulting in a significant enhancement of the accuracy of the numerical integration procedure compared to quadrature formulas that are based on the *trapezoidal* or *midpoint* rules with equidistant grids [5]. In most cases, the transformed integrand displays a rapid roll-off (decay) at a *double exponential* rate, enabling the numerical integrator to quickly achieve convergence [5]. This method is therefore also known as the *Double Exponential* (DE) formula [2,4]. Implementations of the DE methods can be found in popular open source *math libraries*, such as Boost for C++ and mpmath for Python, as well as in popular open source calculator software such as for the WP-34S.

The *Tanh-Sinh* method has an advantage to integrate smooth functions that are *holomorphic* (are at least differentiable), such as transcendental functions. Especially integrands with *L*- and *U*-shapes, where most of the integral's mass is located at one or both interval endpoints, are rapidly integrated with a high accuracy. Having said that, the choice to use *Tanh-Sinh* in practice may depend on the required error tolerance (*eps*) and the properties of the integrands. For rough error tolerance *eps* between 10^{-1} and 10^{-6}, *Romberg* and *Adaptive Simpson* are reasonably good general-purpose integrators over closed intervals. However, *Tanh-Sinh* favorably compares to these methods when integrating smooth functions with a high accuracy of the integral with over six digits precision $eps < 10^{-6}$.

A modification of the *Tanh-Sinh* formula was introduced by Krzysztof Michalski and Juan Mosig [2]. This modification simplifies the formulas for the abscissas and weights. This modification requires fewer arithmetic operations to speed up numerical integration.

This article presents effective improvements to the Michalski & Mosig *Tanh-Sinh* quadrature method[1]. The improvements are compared the Boost, mpmath, and WP-34S implementations of the *Tanh-Sinh* method. In addition, a new *Exp-Sinh* pre-conditioning step is proposed to compute an optimal splitting point on the integration interval for this quadrature method.

[1] The C and BASIC source code presented in this document (excluding the code and examples shown in Appendix A and C) may be distributed freely under the MIT license.

Contents

The Takahashi & Mori Tanh-Sinh quadrature formula

The characteristic *Tanh-Sinh* distribution of abscissas x_k (points) are defined by:

$$x_k = \tanh\left(\tfrac{1}{2}\pi \sinh kh\right)$$

The weights w_k are defined by:

$$w_k = \frac{\frac{1}{2}h\pi \cosh kh}{\cosh^2\left(\frac{1}{2}\pi \sinh kh\right)}$$

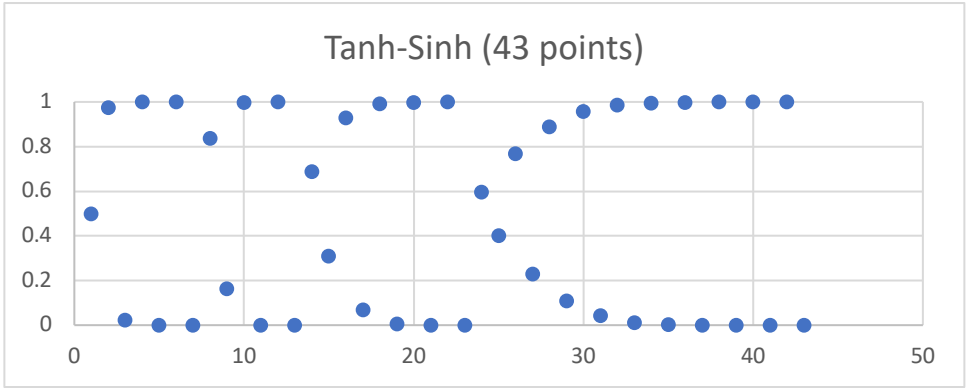
For a given step size h the integral is approximated by:

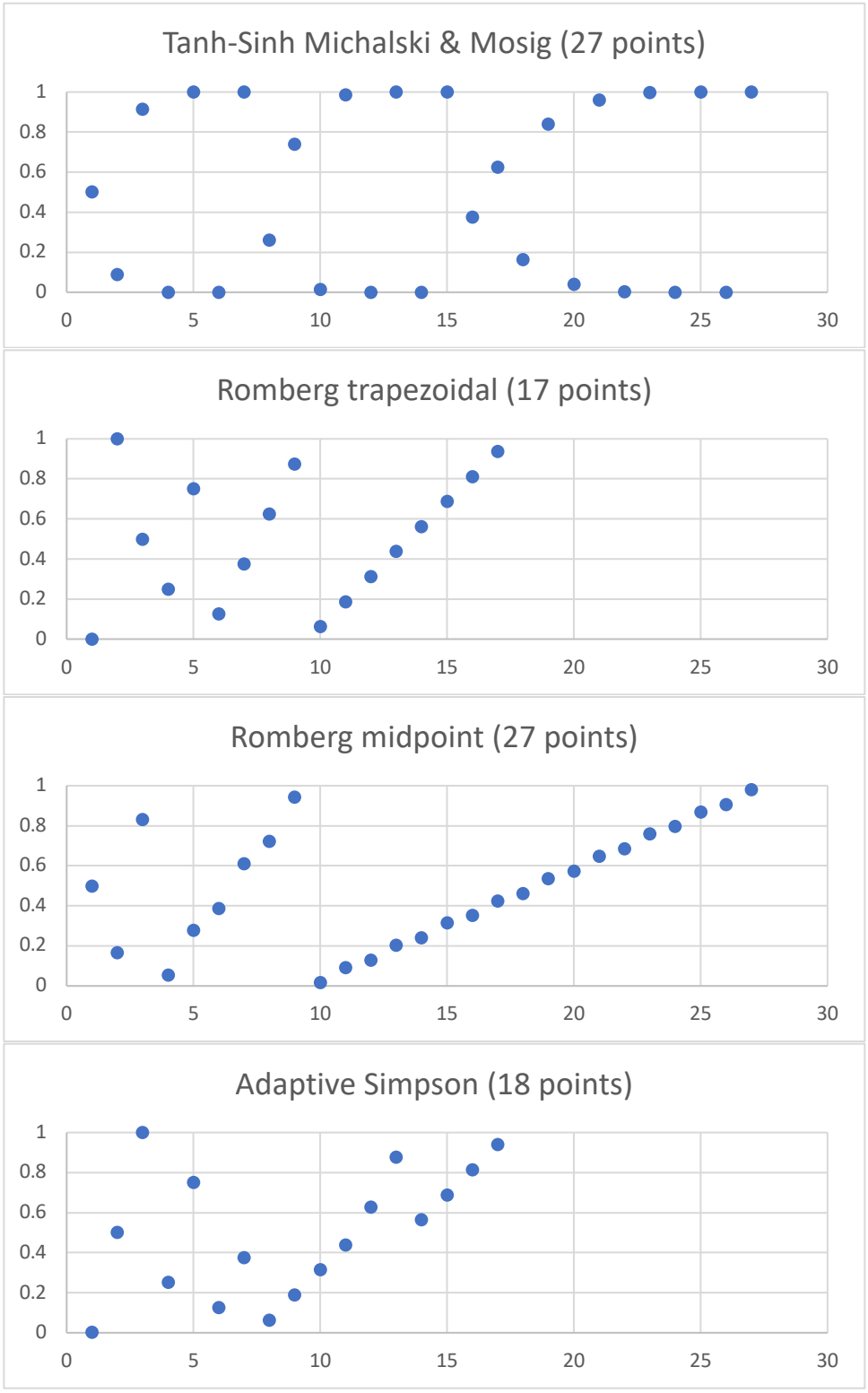
$$\int_{-1}^{1} f(x)\,dx \approx \sum_{k=-\infty}^{\infty} w_k f(x_k)$$

The open interval (−1, +1) can be adjusted with a change in variable to integrate function f over any finite interval.

Comparing Tanh-Sinh to Romberg and Adaptive Simpson

The following five graphs show the distribution of the characteristic *Tanh-Sinh* abscissas to the points generated by the *Romberg* and *Adaptive Simpson* methods. Each graph shows the locations of abscissas (y-axis) on the interval (0,1) to compute $\int_0^1 \frac{4}{1+x^2}dx$ are shown over time (x-axis) for each integration method to reach convergence for a relatively high error threshold $eps = 10^{-5}$:





Because the error threshold is deliberately high in this comparison, requiring only a few digits of precision, *Tanh-Sinh* does not outperform the *Romberg* and *Adaptive Simpson* methods in this

example. However, when the error threshold is further restricted to increase the accuracy of the integral, *Tanh-Sinh* typically outperforms most other quadrature methods.

Let's see what the performance of *Tanh-Sinh* is in practice for a variety of integrands. We pick a collection of 21 arbitrary functions that range from easy to very hard to integrate numerically.

The following table shows the performance of *Tanh-Sinh* with the default n=6 levels compared to *Romberg* (trapezoidal with max n=16 levels), *Adaptive Simpson* (max n=16 levels) and *Adaptive Gauss-Kronrod (G10,K21)* (max n=10 levels, $tol = 10^{-7}$) for $\int_0^1 f(x)\,dx$ with $eps = 10^{-9}$ for all methods. Source code listings for *Romberg*, *Adaptive Simpson* and *Adaptive Gauss-Kronrod (G10,K21)* are included in Appendix D. These methods produce error estimations (relative or absolute), which are normalized to absolute errors in the table together with the number of points evaluated as pairs (*function evaluations*, *estimated error*). The "best results" are obtained when the error is low with the fewest function evaluations:

#	*f*(*x*)	*M. & M. Tanh-Sinh improved*	*Romberg Trapezoidal*	*Adaptive Simpson*	*Adaptive (G10,K21)*
1	x^3-2x^2+x	(49,1e-13)	(9,exact)	(5,exact)	(41,exact)
2	1/(1+x)	(58,1e-13)	(33,2e-9)	(97,6e-11)	(41,3e-17)
3	4/(1+x*x)	(58,3e-14)	(65.2e-11)	(153,3e-11)	(41,1e-16)
4	acos(x)	(58,exact)	(32769,3e-8)	(433,1e-10)	(1763,1e-6)
5	sin(x)/x	(58,9e-16)	(-,-)	(-,-)	(41,4e-16)
6	sqrt(x/(1-x^2))	(120,7e-9)	(-,-)	(-,-)	(2583,1)
7	log(x)^2	(59,1e-13)	(-,-)	(-,-)	(861,7e-2)
8	1/sqrt(x)	(63,1e-13)	(-,-)	(-,-)	(861,2)
9	1/sqrt(1-x)	(120,6e-9)	(-,-)	(-,-)	(861,2)
10	x^-.8	(71,1e-14)	(-,-)	(-,-)	(861,200)
11	(1-x)^-.8	(487,5e-6)	(-,-)	(-,-)	(861,200)
12	1/sqrt(sin(pi*x))	(63,9e-9)	(-,-)	(-,-)	(1599,2)
13	sin(pi*x)^-.8	(534,1e-5)	(-,-)	(-,-)	(1599,100)
14	1/sqrt(-log(x))	(120,6e-9)	(-,-)	(-,-)	(1927,2)
15	1/sqrt(-log(1-x))	(120,6e-9)	(-,-)	(-,-)	(2501,2)
16	sin(pi*x*40)	(315,2e-5)	(9,5e-15)	(5,2e-16)	(41,2e-16)
17	1/(1+25*x^2)	(110,8e-12)	(257,1e-10)	(277,3e-11)	(41,9e-13)
18	1/(1+0.04*x^2)	(58,1e-15)	(17,1e-11)	(33,3e-11)	(41,2e-16)
19	sqrt(abs(x-.5))	(410,1e-3)	(32769,9e-8)	(713,7e-11)	(5863,1e-6)
20	floor(10*x)	(406,8e-3)	(32769,4e-6)	(345,3e-7)	(123,6e-15)
21	10*x-floor(10*x)	(406,8e-3)	(32769,4e-5)	(345,3e-7)*	(123,1e-16)

Tanh-Sinh improved includes improvements proposed in this article
(-,-) fails with floating point error due to *f*(*x*) singularities at endpoint(s)
(points,error) large reported error, the method is not usable for this integrand
(points,error)* wrong integration result 0.45 instead of 0.5, despite the low error estimate

While this comparison does not offer a comprehensive comparison, it should be illustrative of the methods' applicability and precision. *Tanh-Sinh* generally performs well for transcendental

functions and other smooth functions that are differentiable on an open integration interval. Periodic functions, such as integral #16, are not *holomorphic* and ill-suited for *Tanh-Sinh*. Non-differentiable functions, such as integral #19, integral #20 (step function) and integral #21 (saw-tooth function), pose serious problems for *Tanh-Sinh*. Integrals #19, #20 and #21 are "integration busters" that can fool a quadrature method into fitting a smooth polynomial to the function to approximate the integral, requiring many points (*Romberg*) or refinements with more points around sharp edges (*Adaptive Simpson*). Furthermore, *Romberg* (trapezoidal) and *Adaptive Simpson* require a closed interval.

Note that *Adaptive Simpson* fails to correctly integrate integral #21, which is generally hard to integrate numerically. This failure is due to early termination of the recursive scheme when the convergence checks are satisfied. Decreasing *eps* to a minimum does not improve the result. Only forcing deeper recursion by modifying the convergence check corrects the problem.

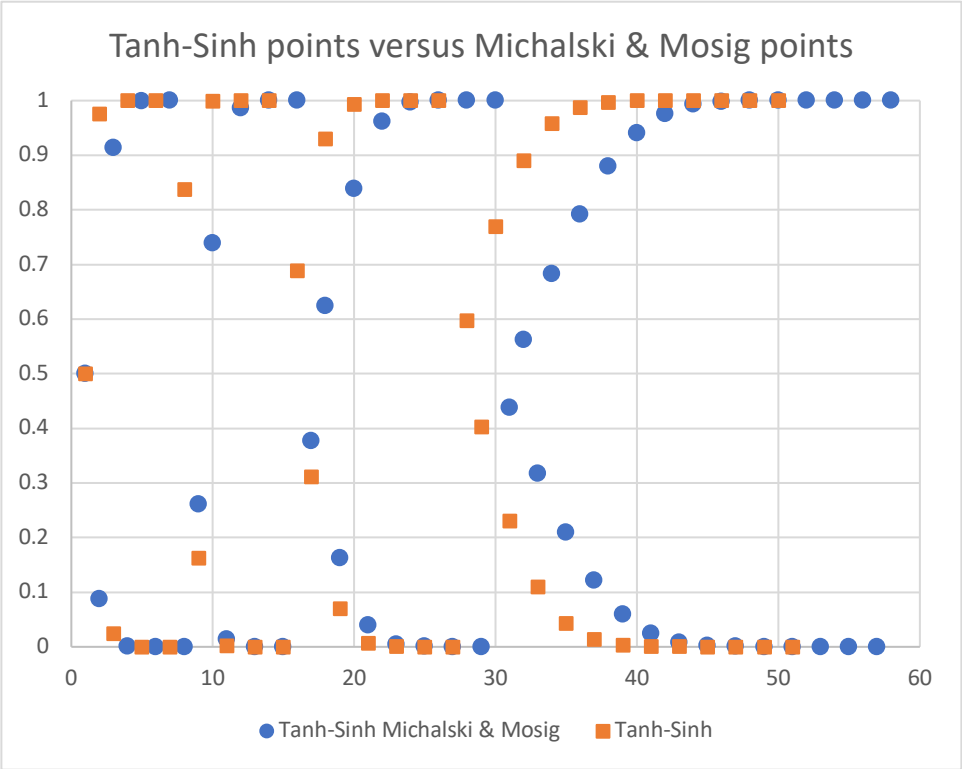
The Michalski & Mosig Tanh-Sinh formula

Krzysztof Michalski and Juan Mosig "*Efficient computation of Sommerfeld integral tails – methods and algorithms*" describe a variant of the *Tanh-Sinh* rule for finite intervals:

$$\int_a^b f(x)\,dx = \sigma \int_{-1}^{1} f(\sigma x + \gamma)\,dx \approx \sigma h \left\{ g'(0) f(\gamma) + \sum_{k=1}^{n} w_k [f(a + \sigma\delta_k) + f(b - \sigma\delta_k)] \right\}$$

with abscissas $a + \sigma\delta_k$, $b - \sigma\delta_k$ and weights $w_k = 2g'(kh)\delta_k/(1 + u_j)$, where $u_k = e^{-2g(kh)}$ and $\delta_k = 1 - \tanh(\sinh(kh)) = 1 - (1 - u_k)/(1 + u_k) = 2u_k/(1 + u_k)$. Choose $g(t) = \eta \sinh t$ and $g'(t) = \eta \cosh t$ for positive parameter η to generate variations of the *Tanh-Sinh* rule. Note that we have $2g(kh) = \eta(e^{kh} - e^{-kh})$ and $2g'(kh) = \eta(e^{kh} + e^{-kh})$. Select $\eta = 1$ and $h = 1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^k}$, then for $j = 1, \ldots, n$ define $t_j = \exp(jh)$, $u_j = \exp(-2\sinh(jh)) = \exp(1/t_j - t_j)$, and $r_j = 2u_j/(1 + u_j)$, giving adjusted weights (scaled by $g'(0)$ for the implementation) $w_j = 2\cosh(jh)\,u_j/(1 + u_j)^2 = (t_j + 1/t_j)r_j/(1 + u_j)$ and abscissas $a + dr_j$ and $b - dr_j$ with $d = \frac{1}{2}(b - a)$.The computation of the abscissas and weights is effectively simplified by dropping the $\frac{1}{2}\pi$ factors from the *Tanh-Sinh* abscissas and weights.

The visualize the different distributions of *Tanh-Sinh* points on the interval, the following graph shows the *Tanh-Sinh* general formula abscissas versus the Michalski & Mosig *Tanh-Sinh* formula abscissas. The abscissas (*y*-axis) on the interval (0,1) to compute $\int_0^1 \frac{4}{1+x^2}\,dx$ are shown over time (*x*-axis), for error threshold $eps = 10^{-9}$ and $k = 0, \ldots 3, h = 2^{-k}$:



The *Tanh-Sinh* variant described by Michalski & Mosig distributes slightly more abscissas between the endpoints of the integration interval as can be seen in the figure.

The next section defines our initial version of a Michalski & Mosig *Tanh-Sinh* algorithm "`qthsh`" (*cutiesh*") in C based on an existing open source implementation in VB (Appendix A). We aggressively optimize and improve this version in subsequent sections and compare it to the state-of-the-art *Tanh-Sinh* implementations in Python mpmath and Boost Math.

A Tanh-Sinh implementation based on Michalski & Mosig

Let's start with an unoptimized implementation of `qthsh` based to the VB code published in the Excel spreadsheet accompanying the article "*Numerical Integration with the Tanh-Sinh Quadrature V5.0*" (Appendix A) using the Michalski & Mosig *Tanh-Sinh* rule:

```
// integrate function f, range a..b, max levels n, error tolerance eps
double qthsh(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 1E-7; // 1E-7 is "optimal"
  double c = (a+b)/2; // center (mean)
  double d = (b-a)/2; // half distance
  double s = f(c);
  double e, h;
  int k = 0;
  if (n > 7)
    n = 6; // 6 is "optimal", 7 just as good taking longer
  do {
    double p = 0, q, fp = 0, fm = 0, v;
```

```
    int j = 1;
    h = pow(2, -k);
    do {
      double t = exp(j*h);
      double u = exp(1/t-t); // = exp(-2*sinh(j*h)) = 1/exp(sinh(j*h))^2
      double r = 2*u/(1+u);  // = 1 - tanh(sinh(j*h))
      double x, w;
      if (r == 0 || r == 1)
        break;
      x = d*r;
      if (a+x > a) // if too close to a then reuse previous fp
        fp = f(a+x);
      if (b-x < b) // if too close to b then reuse previous fm
        fm = f(b-x);
      w = (t+1/t)*r/(1+u); // = cosh(j*h) / cosh(sinh(j*h))^2
      q = w*(fp+fm);
      p += q;
      j += 1+(k>0);
    } while (fabs(q) > fabs(p*eps));
    v = s;
    s += p;
    e = fabs(2*v/s-1);
    ++k;
  } while (e >= tol && k <= n);
  return d*s*h; // result with estimated relative error e
}
```

Code optimizations to improve performance

We apply *loop strength reduction* twice, without affecting the computation's precision. The first strength reduction changes the code from the following:

```
double h;
int k = 0;
...
do {
    double p = 0, q, fp = 0, fm = 0, v;
    int j = 1;
    h = pow(2, -k); // h=1,1/2,1/4,1/8,...
    do {
      double t = exp(j*h);
      ...
      j += 1+(k>0);
    } while (fabs(q) > fabs(p*eps));
```

to the strength reduced version to eliminate `pow(2,-k)`:

```
double h = 2;
int k = 0;
...
do {
    double p = 0, q, fp = 0, fm = 0, v;
    h /= 2;
    do {
```

```
      double t = exp(j*h);
      ...
      j += 1+(k>0);
    } while (fabs(q) > fabs(p*eps));
...
} while (e >= tol && k <= n);
...
return d*s*h; // we need the final h here
```

We apply a second loop strength reduction by observing that `exp(j*h)=exp(h)^j` then strength-reduce the power `^j` away to obtain:

```
double h = 2;
int k = 0;
...
do {
    double p = 0, q, fp = 0, fm = 0, v;
    double t, eh;
    h /= 2;
    t = eh = exp(h);
    if (k > 0)
      eh *= eh
    do {
      ...
      t *= eh;
    } while (fabs(q) > fabs(p*eps));
...
} while (e >= tol && k <= n);
...
return d*s*h;
```

Another simplification is possible if the number of levels `n` is bounded to 7 max, by replacing `exp(h)` by a table lookup indexed by level counter `k`:

```
static const double exptbl[7] = { ... }; // exp(1), exp(.5), ..., exp(2^-7)
...
t = eh = exptbl[k];
```

Alternatively, repeated square roots from `exp(1)` may be used to reduce computational overhead further.

The following conditional loop exit can be removed because the condition is never true. This is also empirically verified with 818 integrals integrated with `qthsh`:

```
      if (r == 0 || r == 1)
        break;
```

Improving the convergence tests

There are two potential problems in the VB code and our initial C version:
- division by zero may occur in the outer loop convergence test
- underflow may occur in the inner loop convergence test

To fix division by zero, we remove the code that assigns the relative error to variable `e` and update the outer loop convergence test, by noting that `2*v-s'=s-p` with new `s'=s+p`:

```
    v = s-p;
    s += p;
    ++k;
  } while (fabs(v) > tol*fabs(s) && k <= n);
  e = fabs(v)/(fabs(s)+eps);
  return d*s*h; // result with estimated relative error e
```

With this first change, integrating $f(x) = x - \frac{1}{2}$ over [0,1] produces the correct integral 0. We also add `eps` to the denominator to obtain a usable error estimate when `fabs(s)` is close to zero. Alternatively, we can set `e=s` and `s=0` when `fabs(s)<eps`.

Underflow can be corrected. However, this change increases the relative error for some functions that converge very slowly, so this cannot be recommended:

~~`} while (fabs(q) > eps*(fabs(p)+eps));`~~

The non-final, faster *Tanh-Sinh* `qthsh` routine with changes highlighted:

```
// integrate function f, range a..b, max levels n, error tolerance eps
double qthsh(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 1E-7; // 1E-7 is "optimal"
  double c = (a+b)/2; // center (mean)
  double d = (b-a)/2; // half distance
  double s = f(c);
  double e, v, h = 2;
  int k = 0;
  if (n > 7)
    n = 6; // 6 is "optimal", 7 just as good taking longer
  do {
    double p = 0, q, fp = 0, fm = 0, t, eh,;
    h /= 2;
    t = eh = exp(h);
    if (k > 0)
      eh *= eh;
    do {
      double u = exp(1/t-t);       // = exp(-2*sinh(j*h)) = 1/exp(sinh(j*h))^2
      double r = 2*u/(1+u);        // = 1 - tanh(sinh(j*h))
      double w = (t+1/t)*r/(1+u); // = cosh(j*h)/cosh(sinh(j*h))^2
      double x = d*r;
      if (a+x > a)                 // if too close to a then reuse previous fp
        fp = f(a+x);
      if (b-x < b)                 // if too close to b then reuse previous fm
        fm = f(b-x);
      q = w*(fp+fm);
      p += q;
      t *= eh;
    } while (fabs(q) > eps*fabs(p));
    v = s-p;
    s += p;
    ++k;
```

```
  } while (fabs(v) > tol*fabs(s) && k <= n);
  e = fabs(v)/(fabs(s)+eps);
  return d*s*h; // result with estimated relative error e
}
```

This article was initially inspired by a group discussion on the HP Forum (see References and Additional Resources) demonstrating the efficiency of *Tanh-Sinh* in calculators such as the excellent WP-34S, including vintage 80s SHARP Pocket Computers with BASIC.

Our SHARP BASIC *Tanh-Sinh* routine `"QTHSH"` is just a few lines long. Like our C version, this BASIC version is optimized. However, computing `EXP(J*H)` in the inner loop is more accurate in SHARP BASIC than repeated multiplication in the loop by a variable, because variables hold 10 digits whereas computations are performed internally with 12-digit precision:

```
100 "QTHSH" E=1E-9,N=6: INPUT "f=F";F$: F$="F"+F$
110 INPUT "a=";A
120 INPUT "b=";B
' init
130 C=(A+B)/2,D=(B-A)/2,X=C: GOSUB F$: S=Y,H=1,K=0
' outer loop
140 J=1,P=0,L=0,M=0
' inner loop
150 T=EXP(J*H),U=EXP(1/T-T),R=2*U/(1+U)
160 X=A+D*R: IF X>A GOSUB F$: L=Y
170 X=B-D*R: IF X<B GOSUB F$: M=Y
180 Q=(T+1/T)*R/(1+U)*(L+M),P=P+Q,J=J+1+(K>0)
190 IF ABS Q>E*ABS P GOTO 150
' exit inner loop
200 X=S-P,S=S+P,K=K+1
210 IF ABS X>1E-7*ABS S IF K<=N LET H=H/2: GOTO 140
' exit outer loop, output result (and relative error estimate if >E)
220 Y=D*S*H,U=ABS X/(ABS S+E)
230 IF U>E LET E=U: PRINT Y,E: END
240 E=U: PRINT Y: END
```

For this up to 10-digit accurate BASIC version `N=6` levels maximum appears optimal.

However, these C and BASIC implementations are not final. Additional improvements and optimizations will be discussed and presented in additional sections in this article. First, let's see how our initial non-final version of the `qthsh` routine compares to other *Tanh-Sinh* implementations.

Implementations are not created equal

This section compares our initial, non-final `qthsh` routine to other *Tanh-Sinh* implementations in the WP-34S calculator (coded in C with IEEE 754 double precision floating point, see Appendix B), Python mpmath and C++ Boost Math (double fp). The following table shows the performance of the four methods reported as pairs in the table (*function evaluations*, *estimated relative error*) to integrate the 21 functions $\int_0^1 f(x)dx$ with $eps = 10^{-9}$ precision.

#	*f*(*x*)	qthsh (initial)	WP-34S	mpmath	Boost Math
1	x^3-2x^2+x	(49,1e-13)	(47,8e-12)	(53,1e-16)*	(74,1e-11)
2	1/(1+x)	(58,1e-13)	(49,5e-12)	(53,1e-22)	(74,1e-11)
3	4/(1+x*x)	(30,3e-8)+	(49,1e-8)+	(107,1e-34)	(147,1e-15)
4	acos(x)	(30,6e-8)	(49,2e-13)	(53,1e-25)	(74,4e-13)
5	sin(x)/x	(30,1e-8)	(49,7e-12)	(53,1e-22)	(74,1e-11)
6	sqrt(x/(1-x^2))	(61,2e-8)	(49,5e-8)	(427,1e-10)	(2216,2e-9)
7	log(x)^2	(59,1e-13)	(49,2e-12)	(53,1e-14)#	(74,8e-13)
8	1/sqrt(x)	(33,4e-8)	(25,5e-7)	(427,1e-10)	(74,6e-15)
9	1/sqrt(1-x)	(32,4e-8)	(25,5e-7)	(427,1e-10)	(2216,3e-9)
10	x^-.8	(37,3e-8)	(395,2e-4)	(427,1e-4)	(74,exact)
11	(1-x)^-.8	(487,5e-6)	(395,2e-4)	(427,1e-4)	(133724,1e-3)*
12	1/sqrt(sin(pi*x))	(63,9e-9)	(49,8e-8)	(427,1e-8)*	(586,1e-9)
13	sin(pi*x)^-.8	(534,1e-5)	(395,2e-4)	(427,1e-4)	(133724,2e-3)*
14	1/sqrt(-log(x))	(32,2e-8)	(25,4e-7)	(427,1e-10)	(2216,3e-9)
15	1/sqrt(-log(1-x))	(-,-) *see fix later*	(25,4e-7)	(427,1e-10)	(-,-)
16	sin(pi*x*40)	(315,2e-5)+	(335,1e-16)	(27,5e-22)	(19,2e-15)
17	1/(1+25*x^2)	(110,8e-12)	(49,7e-10)	(53,1e-10)#	(147,2e-11)
18	1/(1+0.04*x^2)	(58,1e-15)	(49,2e-12)	(53,1e-23)	(74,3e-12)
19	sqrt(abs(x-.5))	(410,1e-3)	(395,1e-3)	(427,1e-3)	(133724,5e-7)
20	floor(10*x)	(406,8e-3)	(395,6e-3)	(427,1e-2)	(133724,1e-5)
21	10*x-floor(10*x)	(406,8e-3)	(395,6e-3)	(427,1e-2)	(133724,1e-5)

green results are within a 10× margin of the given error bound *eps*
(-,-) fails with error
* incorrect error estimate reported (too high or too low), actual error shown
# underestimated error reported, actual error is (much) larger
+ overestimated error reported, actual error is within 10^{-9} error bound

Note:

- WP-34S with $eps = 10^{-15}$ was used in this comparison, otherwise errors are too large, e.g. integral #2 has error 10^{-6} and integral #3 has error 10^{-4}.
- Python mpmath may return "misleading" error estimates, not sure why and when this happens. The error estimate is not integrated with *Tanh-Sinh*, but separately computed from the node sets. Python mpmath appears to ignore ±inf terms in the weighted sum.

The differences between the four implementation is significant and mainly has to do with the way the endpoints are approached. This is best illustrated with point-distribution plots for two integrals #1 and #3. Plotting the *y*-axis (points) with a log scale reveals the proximity of the point distribution to the zero endpoint. In subsequent sections, we will use these "lessons learned" to make additional improvements to our initial `qthsh` routine to increase the accuracy of the method by adjusting the tolerance threshold and to handle endpoint singularities without prematurely terminating the convergence. The latter is a observed in the VB code (Appendix A), WP-34S (Appendix B) and mpmath, whereas Boost Math uses a *Tanh-Sinh* quadrature interval (0,1) instead of (-1,+1) and continues to iterate, even when one of the endpoints has a singularity. Boost Math performs suboptimal for integrals #11 and #15. Also, `qthsh` has an issue with #15, which is corrected by handling endpoint singularities differently as described later in this article.

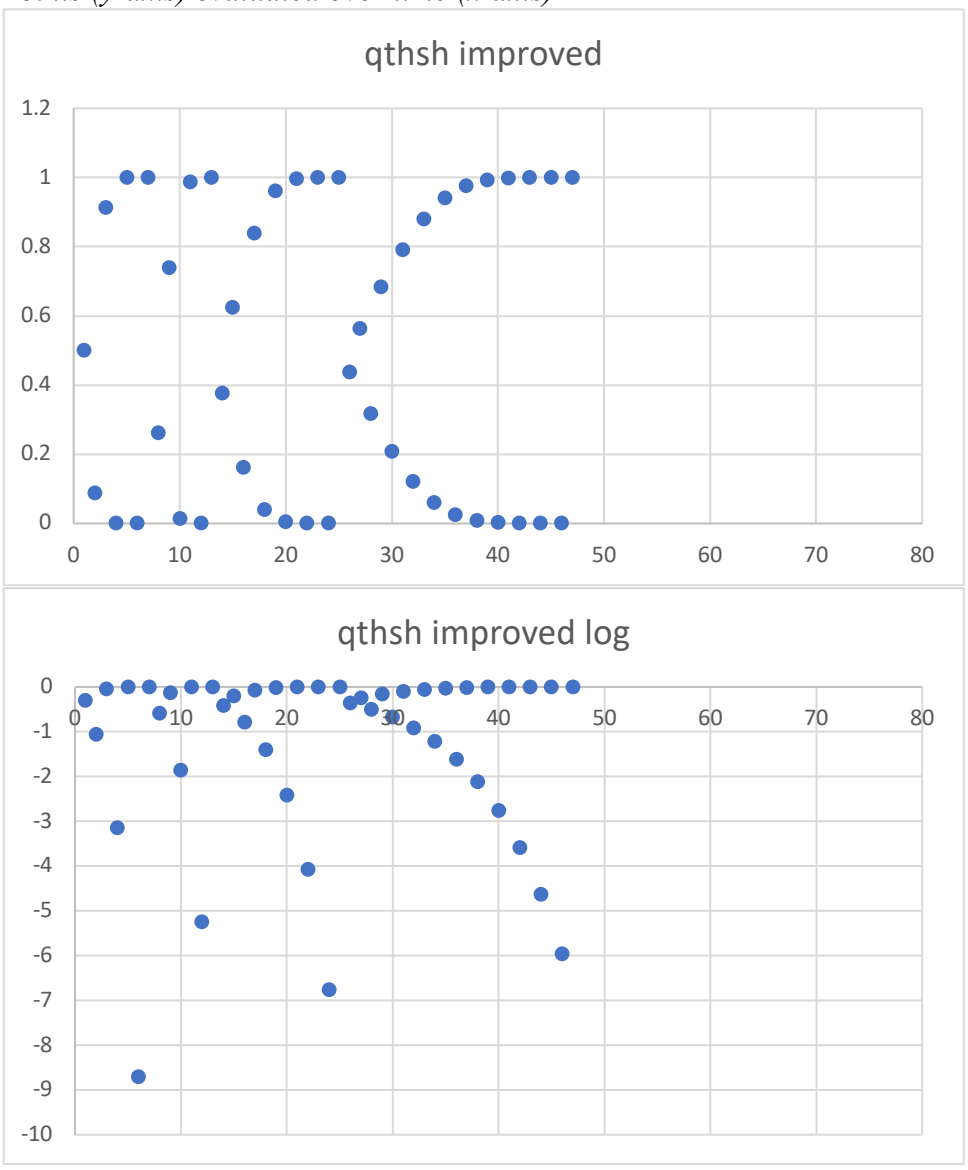
Integral #1:

$$\int_0^1 x^3 - 2x^2 + x \; dx$$

qthsh (C code): *eps*=1e-9, *n*=6, *points*=47, *est.rel.err*=1e-13

```
double f(double x) { ++ev; return x*x*x-2*x*x+x; }
ev = 0; qthsh(f, 0, 1, 6, 1E-9);
```

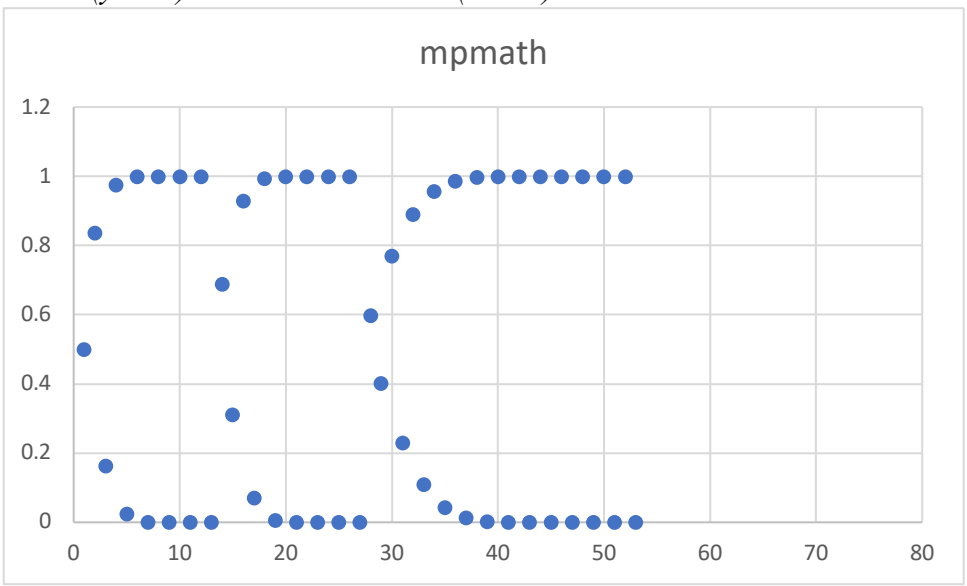
Points (y-axis) evaluated over time (x-axis)

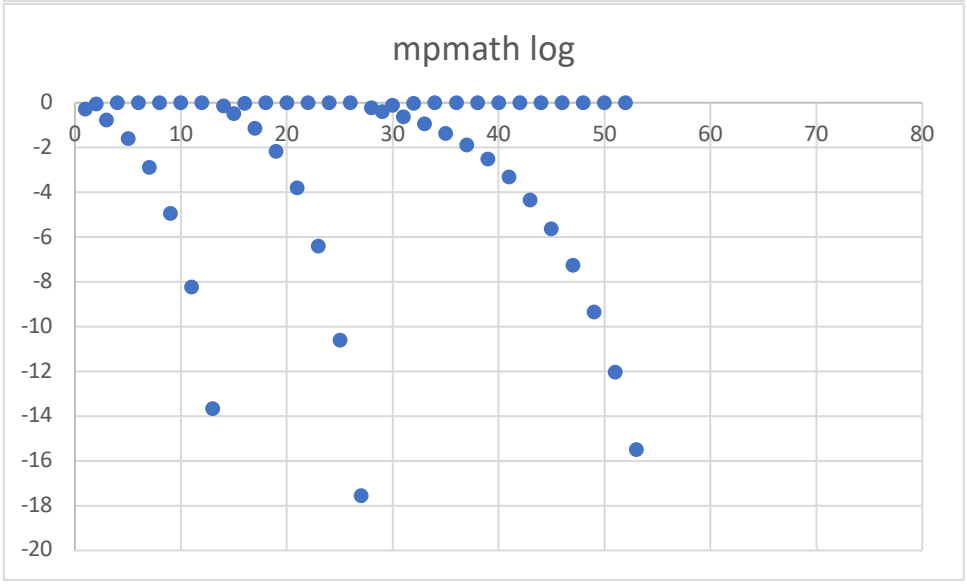


mpmath: mp.dps=15, *points*=53, *n*=6, *est.rel.err*=1e-22 (actual is about 1e-16)

```
from mpmath import *
def bump():
  global evals
  evals += 1
  return 0
f = lambda x: x*x*x-2*x*x+x +bump()
evals=0; quad(f, [0,1], method='tanh-sinh', error=True)
```

Points (y-axis) evaluated over time (x-axis)

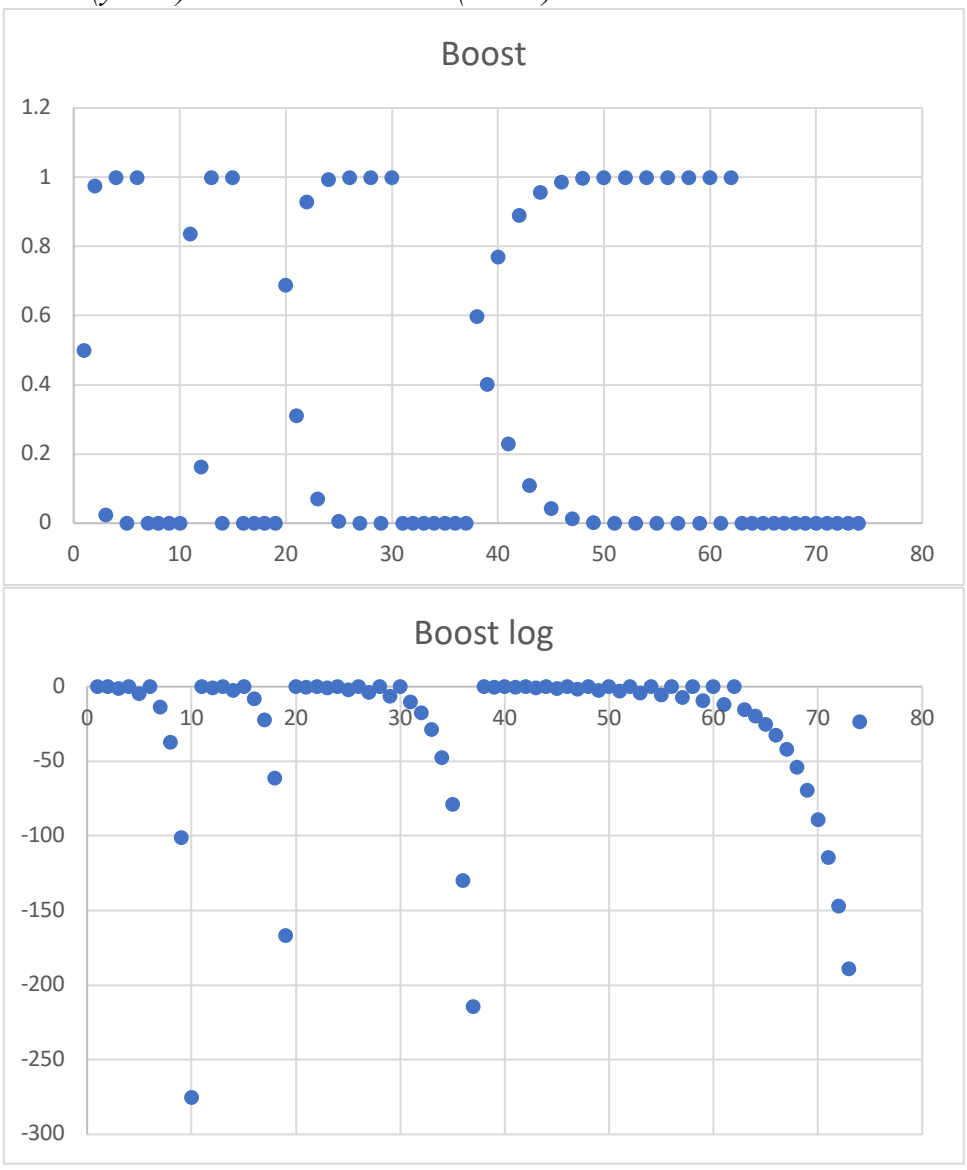




Boost Math: *eps*=1e-9, *points*=74, *est.rel.err*=1e-11

```
#include <boost/math/quadrature/tanh_sinh.hpp>
using namespace boost::math::quadrature;
static int ev = 0;
int main() {
  auto f = [](double x) { ++ev; return x*x*x-2*x*x+x; };
  double error;
  double Q = tanh_sinh<double>().integrate(f, 0.0, 1.0, 1E-9, &error);
  printf("Tanh-sinh %.15g est.rel.err=%g points=%d\n", Q, error, ev);
}
```

Points (y-axis) evaluated over time (x-axis)



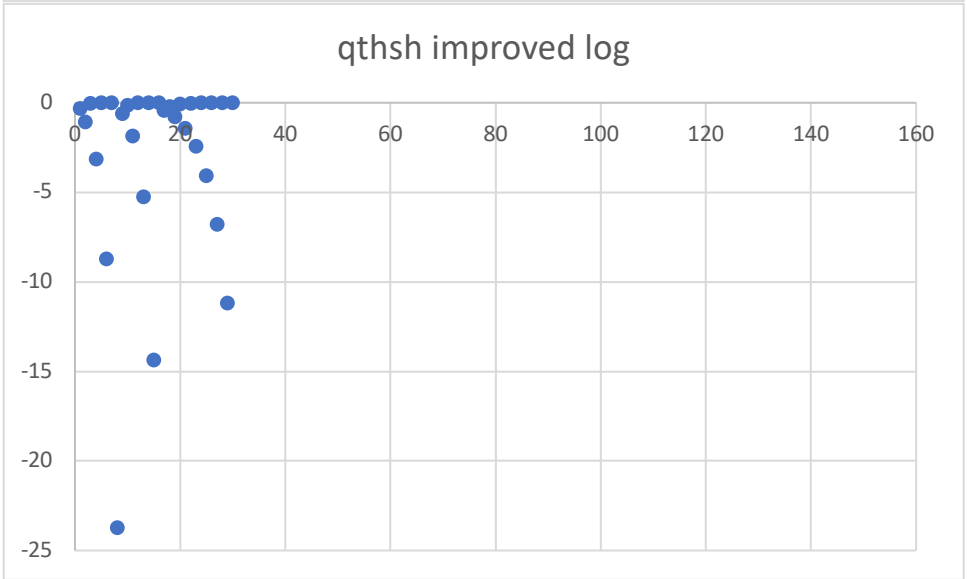
Integral #3:

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

qthsh (C code): *eps*=1e-9, *n*=6, *points*=30, *est.rel.err*=3e-8 (worse than *eps*)

```
double f(double x) { ++ev; return 4/(1+x*x); }
ev = 0; qthsh(f, 0, 1, 6, 1E-9);
```

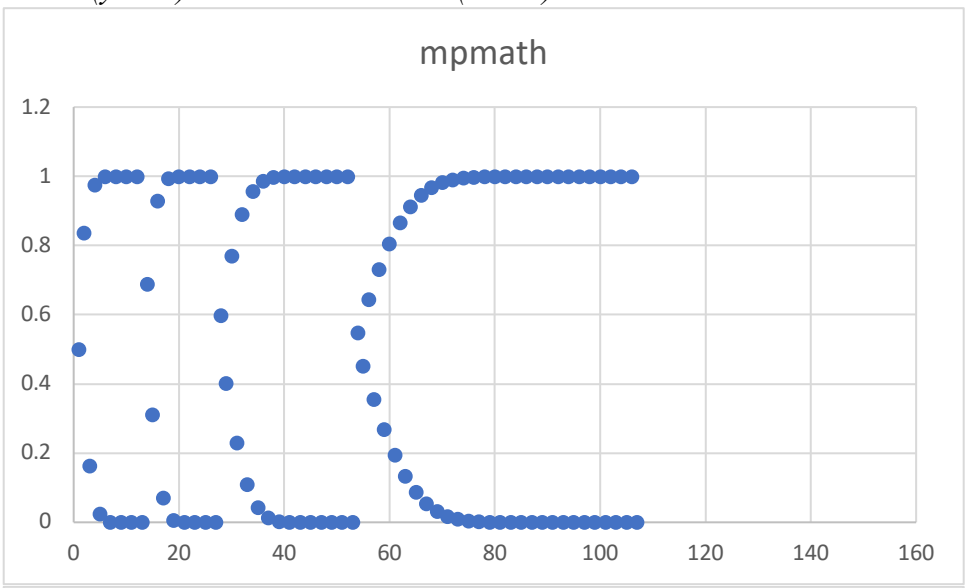
Points (y-axis) evaluated over time (x-axis)

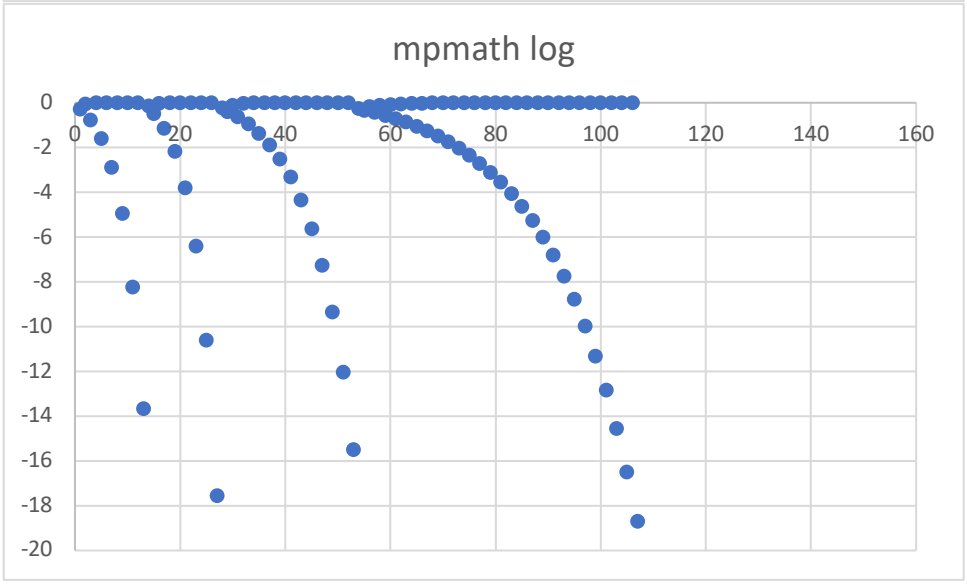


mpmath: mp.dps=15, *points*=107, *n*=6, *est.rel.err*=1e-34

```
from mpmath import *
def bump():
  global evals
  evals += 1
  return 0
f = lambda x: 4/(1+x*x) +bump()
evals=0; quad(f, [0,1], method='tanh-sinh', error=True)
```

Points (y-axis) evaluated over time (x-axis)

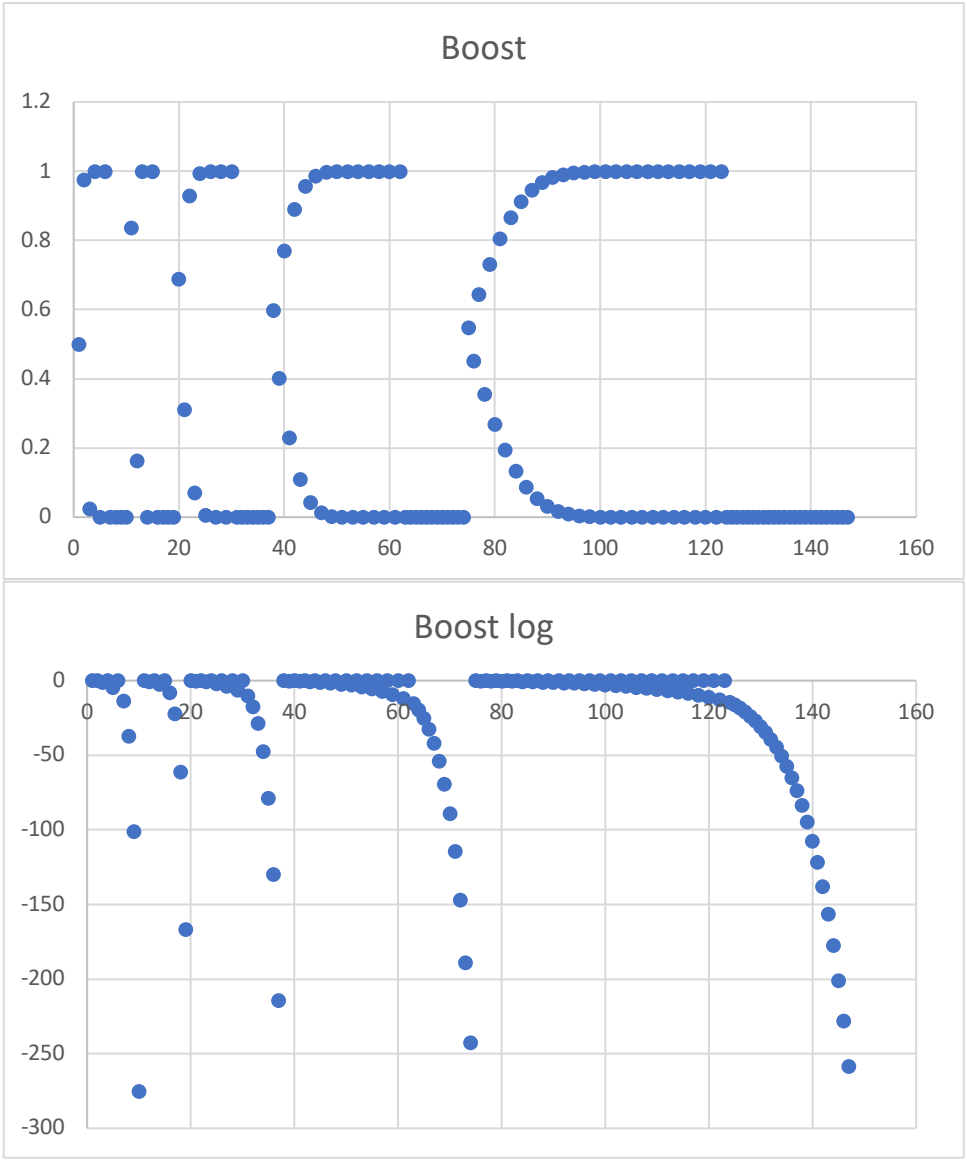




Boost Math: *eps*=1e-9, *points*=147, *n*=6, *est.rel.err*=1e-15

```
#include <boost/math/quadrature/tanh_sinh.hpp>
using namespace boost::math::quadrature;
static int ev = 0;
int main() {
  auto f = [](double x) { ++ev; return 4/(1+x*x); };
  double error;
  double Q = tanh_sinh<double>().integrate(f, 0.0, 1.0, 1E-9, &error);
  printf("Tanh-sinh %.15g est.err=%g points=%d\n", Q, error, ev);
}
```

Points (y-axis) evaluated over time (x-axis)



Improved accuracy by adjusting the tolerance threshold

With $tol = 10^{-7}$ suggested as "optimal" in the VB code (Appendix A), speed is favored over accuracy. A minimum number of points is evaluated, but this may produce *relative error estimates* that are slightly worse ($3 \cdot 10^{-8}$) than a given error tolerance ($eps = 10^{-9}$):

```
// integrate function f, range a..b, max levels n, error tolerance eps
double qthsh(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 1E-7; // 1E-7 is "optimal"
```

Adjusting $tol = eps$ with $eps = 10^{-9}$ the following integrands require more points to evaluate for the accuracy of the result to reach at least 10^{-9} as expected for integrands that should pose no problems for *Tanh-Sinh*'s fast *double exponential* convergence:

#	*f*(*x*)	qthsh *tol*=1e-7	qthsh *tol*=*eps*=1e-9
3	4/(1+x*x)	(30,3e-8)	(58,3e-14)
4	acos(x)	(30,6e-8)	(58,exact)
5	sin(x)/x	(30,1e-8)	(58,9e-16)
6	sqrt(x/(1-x^2))	(61,2e-8)	(472,6e-10)
8	1/sqrt(x)	(33,4e-8)	(63,1e-13)
9	1/sqrt(1-x)	(32,4e-8)	(472,5e-10)
10	x^-.8	(37,3e-8)	(71,1e-14)
12	1/sqrt(sin(pi*x))	(63,9e-9)	(477,5e-10)
14	1/sqrt(-log(x))	(32,2e-8)	(472,6e-10)

The fast convergence of *Tanh-Sinh* almost always overshoots the target tolerance unless the integrand is known to be problematic for *Tanh-Sinh*. We can use $tol = 10 \times eps$ (i.e. $tol = 10^{-8}$) with a factor 10× (tunable to 100×) for the relative error estimate:

#	*f*(*x*)	qthsh *tol*=1e-7	qthsh *tol*=10*eps*=1e-8
3	4/(1+x*x)	(30,3e-8)	(58,3e-14)
4	acos(x)	(30,6e-8)	(58,7e-15)
5	sin(x)/x	(30,1e-8)	(58,9e-16)
6	sqrt(x/(1-x^2))	(61,2e-8)	(120,7e-9)
8	1/sqrt(x)	(33,4e-8)	(63,1e-13)
9	1/sqrt(1-x)	(32,4e-8)	(120,6e-9)
10	x^-.8	(37,3e-8)	(71,1e-14)
12	1/sqrt(sin(pi*x))	(63,9e-9)	(63,9e-9)
14	1/sqrt(-log(x))	(32,2e-8)	(120,7e-9)

Given these results, using a `qthsh` tolerance $tol = 10 \times eps$ appears quite reasonable:

```
// integrate function f, range a..b, max levels n, error tolerance eps
double qthsh(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 10*eps;

210 IF ABS X>10*E*ABS S IF K<=N LET H=H/2: GOTO 140
```

Comparing the estimated relative error `e = fabs(v)/(fabs(s)+eps)` to the actual relative error $\frac{|result-exact|}{|exact|+eps}$ using +*eps* in the denominator to prevent the relative error from blowing up when the result is (close to) zero and comparing the estimated absolute error `fabs(d*v*h)` to the actual absolute error shows a fairly good match:

#	*f*(*x*)	`e`	actual rel.err.	`fabs(d*v*h)`	actual abs.err.
1	x^3-2x^2+x	1e-13	1e-14	1e-14	1e-15
2	1/(1+x)	1e-13	2e-16	8e-14	1e-16
3	4/(1+x*x)	3e-14	4e-16	9e-14	1e-15
4	acos(x)	7e-15	2e-16	7e-15	2e-16
5	sin(x)/x	9e-16	2e-16	9e-16	2e-16
6	sqrt(x/(1-x^2))	7e-9	2e-8	8e-9	2e-8
7	log(x)^2	1e-13	2e-13	2e-13	3e-13
8	1/sqrt(x)	1e-13	1e-13	3e-13	3e-13
9	1/sqrt(1-x)	6e-9	1e-8	1e-8	3e-8
10	x^-.8	1e-14	1e-14	5e-14	5e-14
11	(1-x)^-.8	5e-6	5e-4	3e-5	3e-3
12	1/sqrt(sin(pi*x))	9e-9	1e-8	1e-8	2e-8
13	sin(pi*x)^-.8	1e-5	3e-4	5e-5	1e-3
14	1/sqrt(-log(x))	7e-9	2e-8	1e-8	3e-8
15	1/sqrt(-log(1-x))	7e-9	2e-8	1e-8	3e-8
16	sin(pi*x*40)	2e-4	1e-7	2e-16	2e-16
17	1/(1+25*x^2)	8e-12	8e-12	2e-12	2e-12
18	1/(1+0.04*x^2)	1e-15	3e-16	1e-15	3e-16
19	sqrt(abs(x-.5))	1e-3	6e-4	5e-4	3e-4
20	floor(10*x)	9e-4	9e-4	4e-3	4e-3
21	10*x-floor(10*x)	8e-3	8e-3	4e-3	4e-3

Note that the value of `e` lies within one order of magnitude (10× or one significant digit) from the actual relative error: $e \le 10eps$ if the actual relative error $\le eps$ (the method converged) and $e \ge eps/10$ if the actual relative error $\ge eps$ (the method failed to converge).

Verification of the final version of `qthsh` with 818 integrals of which 814 are integrable (non-NaN) returned 690 results (85%) where `e` lies within one order of magnitude from the actual relative error, returned 720 results (88%) where `e` lies within two orders of magnitude from the actual relative error, and returned 760 results (93%) where `e` lies within three orders of magnitude from the actual relative error. For a wide range of *eps* values 1e-5 to 1e-10, the method generally converges with an actual relative error $\le eps$. Furthermore, the estimated relative error `e` is accurate within 10× in most cases of the 818 integrals tested:

eps=	1e-5	1e-6	1e-7	1e-8	1e-9	1e-10
average number of evaluations	57	69.4	81.7	93.4	106	132
actual relative error $\le eps$	771	766	755	707	559	182
estimated rel.err. is within 10×	809	807	799	780	690	385

The table shows the number of cases when the actual relative error ≤ eps on the second row and the number of cases when the estimated relative error is within 10× of the actual relative error on the third row, that is, $e \le 10eps$ if the actual relative error ≤ eps (the method converged) and $e \ge eps/10$ if the actual relative error ≥ eps (the method failed to converge).

Comparing these `qthsh` results to the VB code (Appendix A) results and WP-34S code (Appendix B, but with a relative error estimate) results for *eps*=1e-9:

	qthsh	VB	WP-34S
average number of evaluations	106	97.8	62
actual relative error ≤ *eps*	559	549	233
estimated rel.err. is within 10×	690	634	593

It should be noted that the VB code succeeds for 796 integrands instead of 814 for the others. This is due to NaN issues that aren't caught in the VB code. Overall, `qthsh` is more accurate, but with a slightly higher average number of function evaluations since the VB code uses `tol=100*eps` to terminate the loop earlier. To reduce the number of function evaluations in `qthsh` to 93.9 we can increase the tolerance, much like the VB code uses a higher tolerance `tol=100*eps`. However, we can use a more optimal `tol=160*eps`. We also adjust the estimated relative error `e` accordingly:

```
// integrate function f, range a..b, max levels n, error tolerance eps
double qthsh(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 160*eps;
  [...]
  e = fabs(v)/(16*fabs(s)+eps);
```

where the `160*eps` and the `16*` factor in `e` were empirically determined to optimize the accuracy of the estimated relative error `e`:

eps=	1e-5	1e-6	1e-7	1e-8	1e-9	1e-10
average number of evaluations	47.5	59.2	68.9	82	93.9	106
actual relative error ≤ *eps*	754	767	753	715	559	183
estimated rel.err. is within 10×	805	803	796	777	682	352

Note that the actual and estimated relative errors are only slightly worse while significantly accelerating the method by reducing the number of function evaluations by 10% or more. By comparison, the 93.9 average number of evaluations of `qthsh` for *eps*=1e-9 is lower than the VB code. Also `qthsh` has a higher number of accurate results within the actual and estimated relative errors.

These results suggests that `e` can be a reliable estimate of the relative error for *eps* up to 1e-9 and the tolerance can be increased to accelerate the method by reducing the number of function evaluations at a very modest penalty in the (estimated) accuracy of the result. However, this accelerated convergence with `160*eps` assumes the machine used IEEE 754 double precision. The acceleration may not be applicable to machines that do not use IEEE 754 double floating point, for example BCD machines such as calculators.

Dealing with singularities more effectively and accurately

When the endpoints are "hit" by the abscissas on an open interval, it is advantageous to reuse the previous endpoint as if computed "in the limit", e.g. $\lim_{x\to 0} \frac{\sin x}{x} = 1$. From 818 integrals tested, this approach improved the result of 58 integrals versus 5 that were only slightly worse due to noise.

When a function returns ±inf or NaN, there are two options:
- terminate the convergence loop because subsequent points will likely produce ±inf or NaN
- ignore ±inf and NaN, where "ignore" means to either replace its value with an interpolated value or assume it is zero. In this case we evaluate more function points to continue iterating.

The second option appears to be more accurate, because the values towards the other endpoint that has no singularity are still accumulated in the quadrature sum and not interrupted. Reusing previous points affects only 12 integrals of 818 tested, where in 4 cases the integration error was 10 times smaller when points are reused. As a tradeoff for this gain, more function evaluations may be required. The corresponding change to `qthsh` to handle singularities is as follows:

```
if (a+x > a) {       // if too close to a then reuse previous fp
  double y = f(a+x);
  if (isfinite(y))
    fp = y;          // if f(x) is finite, add to the local sum
}
if (b-x < b) {       // if too close to b then reuse previous fm
  double y = f(b-x);
  if (isfinite(y))
    fm = y;          // if f(x) is finite, add to the local sum
}
```

With this subtle change, function 15 is efficiently integrated:

#	*f*(*x*)	qthsh final
15	1/sqrt(-log(1-x))	(120,7e-9)

The estimated integral #15 is now the same as integral #14 as should be expected, up to the last digit 1.77245382409889. Both are the same for the same specified *eps*.

SHARP BASIC implementations can use `ON-ERROR-GOTO` to implement a similar scheme, e.g. to reuse `fp` (stored in `L`) or `fm` (stored in `M`) when an overflow or division by zero occurred:

```
100 ON ERROR GOTO 290
210 X=A+D*R,Y=L: IF X>A GOSUB F$: L=Y
220 X=B-D*R,Y=M: IF X<B GOSUB F$: M=Y
290 IF <calculation error> RETURN
```

Important: singularities *anywhere on the integration domain* are effectively interpolated, not only singularities at or close to the endpoints of the domain. As always, due diligence should be applied when integrating functions with singularities on the domain, by splitting the domain up into parts to avoid singularities and non-differentiable points in the domain.

The updated table with the final qthsh improvements shows the performance of the four methods reported as pairs in the table (*function evaluations*, *estimated relative error*) to integrate the 21 functions $\int_0^1 f(x)dx$ with a given error bound $eps = 10^{-9}$:

#	*f*(*x*)	qthsh final	WP-34S	mpmath	Boost Math
1	x^3-2x^2+x	(49,1e-13)	(47,8e-12)	(53,1e-16)*	(74,1e-11)
2	1/(1+x)	(58,1e-13)	(49,5e-12)	(53,1e-22)	(74,1e-11)
3	4/(1+x*x)	(58,3e-14)	(49,1e-8)+	(107,1e-34)	(147,1e-15)
4	acos(x)	(58,1e-15)	(49,2e-13)	(53,1e-25)	(74,4e-13)
5	sin(x)/x	(58,9e-16)	(49,7e-12)	(53,1e-22)	(74,1e-11)
6	sqrt(x/(1-x^2))	(120,7e-9)	(49,5e-8)	(427,1e-10)	(2216,2e-9)
7	log(x)^2	(59,1e-13)	(49,2e-12)	(53,1e-14)#	(74,8e-13)
8	1/sqrt(x)	(63,1e-13)	(25,5e-7)	(427,1e-10)	(74,6e-15)
9	1/sqrt(1-x)	(120,6e-9)	(25,5e-7)	(427,1e-10)	(2216,3e-9)
10	x^-.8	(71,1e-14)	(395,2e-4)	(427,1e-4)	(74,exact)
11	(1-x)^-.8	(487,5e-6)	(395,2e-4)	(427,1e-4)	(133724,1e-3)*
12	1/sqrt(sin(pi*x))	(63,9e-9)	(49,8e-8)	(427,1e-8)*	(586,1e-9)
13	sin(pi*x)^-.8	(534,1e-5)	(395,2e-4)	(427,1e-4)	(133724,2e-3)*
14	1/sqrt(-log(x))	(120,7e-9)	(25,4e-7)	(427,1e-10)	(2216,3e-9)
15	1/sqrt(-log(1-x))	(120,7e-9)	(25,4e-7)	(427,1e-10)	(-,-)
16	sin(pi*x*40)	(315,2e-5)+	(335,1e-16)	(27,5e-22)	(19,2e-15)
17	1/(1+25*x^2)	(110,8e-12)	(49,7e-10)	(53,1e-10)#	(147,2e-11)
18	1/(1+0.04*x^2)	(58,1e-15)	(49,2e-12)	(53,1e-23)	(74,3e-12)
19	sqrt(abs(x-.5))	(410,1e-3)	(395,1e-3)	(427,1e-3)	(133724,5e-7)
20	floor(10*x)	(406,8e-3)	(395,6e-3)	(427,1e-2)	(133724,1e-5)
21	10*x-floor(10*x)	(406,8e-3)	(395,6e-3)	(427,1e-2)	(133724,1e-5)

green results are within a 10× margin of the given error bound *eps*
(-,-) fails with error
* incorrect error estimate reported (too high or too low), actual error shown
# underestimated error reported, actual error is (much) larger
+ overestimated error reported, actual error is within 10^{-9} error bound

In addition to these 21 test cases, qthsh was empirically verified with 818 integrals. Overall, the final version of qthsh performs competitively if not better compared to WP-34S, mpmath and Boost Math. The aim is to produce a result within the 10^{-9} error bound while performing a low number of function evaluations (not necessarily the lowest number of evaluations, but comparatively low.) The WP-34S method appears to "underdeliver" with a higher error in the result than the desired $eps = 10^{-9}$ for integrals #8, #9, #10, #11, #14 and #15. These integrals are harder, but not too hard to integrate. Python mpmath appears to do the opposite, sometimes "overdelivering" with a low error in the result with more function evaluations, such as the easy integrands in #1, #2 and #3. However, mpmath fails to integrate #10, #11, #12 and #13. Boost Math performs well in general but requires the highest number of function evaluations compared to the other three methods and fails to integrate #11, #13 and #15.

The final improved `qthsh` routine based on the Michalski & Mosig *Tanh-Sinh* rule:

```
// integrate function f, range a..b, max levels n, error tolerance eps
double qthsh(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 10*eps;
  double c = (a+b)/2; // center (mean)
  double d = (b-a)/2; // half distance
  double s = f(c);
  double e, v, h = 2;
  int k = 0;
  if (n <= 0) // use default levels n=6
    n = 6; // 6 is "optimal", 7 just as good taking longer
  if (eps <= 0) // use default eps=1E-9
    eps = 1E-9;
  do {
    double p = 0, q, fp = 0, fm = 0, t, eh;
    h /= 2;
    t = eh = exp(h);
    if (k > 0)
      eh *= eh;
    do {
      double u = exp(1/t-t);      // = exp(-2*sinh(j*h)) = 1/exp(sinh(j*h))^2
      double r = 2*u/(1+u);       // = 1 - tanh(sinh(j*h))
      double w = (t+1/t)*r/(1+u); // = cosh(j*h)/cosh(sinh(j*h))^2
      double x = d*r;
      if (a+x > a) {              // if too close to a then reuse previous fp
        double y = f(a+x);
        if (isfinite(y))
          fp = y;                 // if f(x) is finite, add to the local sum
      }
      if (b-x < b) {              // if too close to b then reuse previous fm
        double y = f(b-x);
        if (isfinite(y))
          fm = y;                 // if f(x) is finite, add to the local sum
      }
      q = w*(fp+fm);
      p += q;
      t *= eh;
    } while (fabs(q) > eps*fabs(p));
    v = s-p;
    s += p;
    ++k;
  } while (fabs(v) > tol*fabs(s) && k <= n);
  e = fabs(v)/(fabs(s)+eps);
  return d*s*h; // result with estimated relative error e
}
```

The final improved SHARP BASIC version of the routine:

```
' VARIABLES
'  A,B     range
'  F$      function label to integrate
'  Y       result with error E
'  E       estimated error
'  N       levels (up to 6 or 7)
'  C       (a+b)/2 center
'  D       (b-a)/2 half distance
```

```
'  H       step size h=2^-k
'  K       level counter
'  P,Q,S   quadrature sums
'  T       exp(j*h)
'  L       fp=f(a+x)
'  M       fm=f(b-x)
'  O       exp(h)
'  J,R,U,X scratch
100 "QTHSH" ON ERROR GOTO 290: E=1E-9,N=6: INPUT "f=F";F$:
F$="F"+F$
110 INPUT "a=";A
120 INPUT "b=";B
' init
130 C=(A+B)/2,D=(B-A)/2,X=C: GOSUB F$: S=Y,H=1,K=0
' outer loop
140 J=1,P=0,L=0,M=0
' inner loop
150 T=EXP(J*H),U=EXP(1/T-T),R=2*U/(1+U)
160 X=A+D*R,Y=L: IF X>A GOSUB F$: L=Y
170 X=B-D*R,Y=M: IF X<B GOSUB F$: M=Y
180 Q=(T+1/T)*R/(1+U)*(L+M),P=P+Q,J=J+1+(K>0)
190 IF ABS Q>E*ABS P GOTO 150
' exit inner loop
200 X=S-P,S=S+P,K=K+1
210 IF ABS X>10*E*ABS S IF K<=N LET H=H/2: GOTO 140
' exit outer loop, output result (and relative error estimate if >E)
220 Y=D*S*H,U=ABS X/(ABS S+E)
230 IF U>E LET E=U: PRINT Y,E: END
240 E=U: PRINT Y: END
' The PC-1475 supports ON-ERROR-GOTO and ERN:
290 IF ERN=2 RETURN
```

Note: ON-ERROR-GOTO is not universally supported and can be omitted from the code.

Combining Tanh-Sinh with Exp-Sinh and Sinh-Sinh quadratures

The *Tanh-Sinh* rule is applicable to open finite intervals. The *Exp-Sinh* rule is applicable to an interval with one finite and one infinite bound:

$$\int_a^{\infty} f(x)\,dx \approx h\left\{g'(0)f(a+1) + \sum_{k=1}^{n} w_k f(a+\delta_k) + f(a+\frac{1}{\delta_k})/w_k\right\}$$

with abscissas $a+\delta_k$, $a+1/\delta_k$ and:

$$\int_{\infty}^{b} f(x)\,dx \approx -h\left\{g'(0)f(b-1) + \sum_{k=1}^{n} w_k f(b-\delta_k) + f(b-\frac{1}{\delta_k})/w_k\right\}$$

with abscissas $b-\delta_k$, $b-1/\delta_k$ and weights $w_k = \delta_k = \exp(\sinh(kh))$. For $j = 1, \dots, n$ define $t_j = \exp(jh)$, $w_j = r_j = \exp(\sinh(jh)) = \exp(t_j - 1/t_j)$. Note that the $\frac{\pi}{2}$ factor is dropped similar to the Michalski & Mosig *Tanh-Sinh* rule.

To directly combine the *Exp-Sinh* rule with the *Tanh-Sinh* rule into one routine, we reformulate the Michalski & Mosig *Tanh-Sinh* rule's abscissas as $\gamma + \sigma\delta_k$ and $\gamma - \sigma\delta_k$ with $\delta_k = \tanh(sinh(kh))$ and weights $w_k = \cosh(\sinh(kh))^{-2}$:

$$\int_a^b f(x)\,dx = \sigma\int_{-1}^{1} f(\sigma x+\gamma)\,dx \approx \sigma h\left\{g'(0)f(\gamma) + \sum_{k=1}^{n} w_k[f(\gamma+\sigma\delta_k) + f(\gamma-\sigma\delta_k)]\right\}$$

The abscissas in this combined implementation are defined by $c + dr_j$ and $c - dr_j$ with $c = (a+b)/2$, $d = (b-a)/2$ and $r_j = \tanh(sinh(jh)) = (u_j - 1/u_j)/(u_j + 1/u_j)$ and weights $w_j = \cosh(\sinh(jh))^{-2} = 4/(u_j + 1/u_j)^2$ where $u_j = \exp(t_j - 1/t_j)$. This allows c and d to be defined specific to *Tanh-Sinh* as defined above and defined for *Exp-Sinh* with $c = a$ or $c = b$ and $d = 1$ and for *Sinh-Sinh* with $c = 0$ and $d = 1$.

The *Sinh-Sinh* rule is applicable to unbounded intervals:

$$\int_{\infty}^{\infty} f(x)\,dx \approx h\left\{g'(0)f(0) + \sum_{k=1}^{n} w_k[f(\delta_k) + f(-\delta_k)]\right\}$$

With abscissas $\pm\delta_k$ and weights $w_k = \cosh(\sinh(kh))$. For $j = 1, \dots, n$ define $t_j = \exp(jh)$, $w_j = \cosh(\sinh(jh)) = \exp(t_j + 1/t_j)/2$, $r_j = \sinh(\sinh(jh)) = \exp(t_j - 1/t_j)/2$.

The combined quadrature routine with one inner loop to compute the *Tanh-Sinh*, *Exp-Sinh* and *Sinh-Sinh* quadrature:

```
// integrate function f, range a..b, max levels n, error tolerance eps
double quad(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 10*eps;
  double c = 0, d = 1, s, sign = 1, e, v, h = 2;
  int k = 0, mode = 0; // Tanh-Sinh = 0, Exp-Sinh = 1, Sinh-Sinh = 2
  if (b < a) { // swap bounds
    v = b;
    b = a;
    a = v;
    sign = -1;
  }
  if (isfinite(a) && isfinite(b)) {
    c = (a+b)/2;
    d = (b-a)/2;
    v = c;
  }
  else if (isfinite(a)) {
    mode = 1; // Exp-Sinh
    c = a;
    v = a+d;
  }
  else if (isfinite(b)) {
    mode = 1; // Exp-Sinh
    d = -d;
    sign = -sign;
    c = b;
    v = b+d;
  }
  else {
    mode = 2; // Sinh-Sinh
    v = 0;
  }
  s = f(v);
  do {
    double p = 0, q, t, eh;
    h /= 2;
    eh = exp(h);
    t = eh/2;
    if (k > 0)
      eh *= eh;
    do {
      double r, w, x, y;
      q = 0;
      r = w = exp(t-.25/t); // = exp(sinh(j*h))
      if (mode != 1) {      // if Tanh-Sinh or Sinh-Sinh
        w += 1/w;           // = 2*cosh(sinh(j*h))
        r -= 1/r;           // = 2*sinh(sinh(j*h))
        if (mode == 0) {    // if Tanh-Sinh
          r /= w;           // = tanh(sinh(j*h))
          w = 4/(w*w);      // = 1/cosh(sinh(j*h))^2
        }
        else {              // if Sinh-Sinh
          r /= 2;           // = sinh(sinh(j*h))
          w /= 2;           // = cosh(sinh(j*h))
```

```
        }
        x = c - d*r;        // will not approach a=0 as close as qthsh
        if (x > a) {
          y = f(x);
          if (isfinite(y))  // if f(x) is finite, add to local sum
            q += y*w;
        }
      }
      else {                // Exp-Sinh
        x = c + d/r;        // will not approach a=0 as close as qthsh
        if (x > a) {
          y = f(x);
          if (isfinite(y))  // if f(x) is finite, add to local sum
            q += y/w;
        }
      }
      x = c + d*r;          // will not approach b=0 as close as qthsh
      if (x < b) {
        y = f(x);
        if (isfinite(y))    // if f(x) is finite, add to local sum
          q += y*w;
      }
      q *= t+.25/t;         // q *= cosh(j*h)
      p += q;
      t *= eh;
    } while (fabs(q) > eps*fabs(p));
    v = s-p;
    s += p;
    ++k;
  } while (fabs(v) > tol*fabs(s) && k <= n);
  e = fabs(v)/(fabs(s)+eps);
  return sign*d*s*h; // result with estimated relative error e
}
```

Because the performance of the inner loop determines the overall performance of the routine, several optimizations were applied while keeping the code compact, most notably *strength reductions*, *branch eliminations*, and *variable reuse* to reduce CPU register pressure.

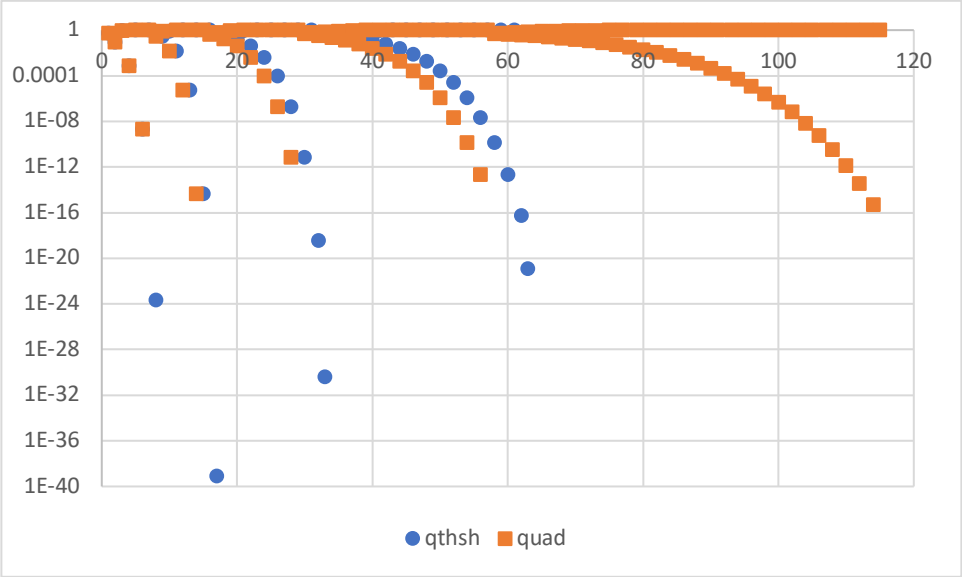
The `quad` routine implements the Michalski & Mosig *Tanh-Sinh* rule with the same abscissas and weights as `qthsh`, but ignores singularities by assuming zero as the default, rather than interpolating them with the previous points.

Another important difference between the `qthsh` and `quad` routines are the function evaluation guards in the routine to check for the endpoints: `a+d*r>a` in `qthsh` versus `c-d*r` in `quad` and `b-d*r<b` in `qthsh` versus `c+d*r<b` in `quad`. If the finite endpoints `a` or `b` are zero or close to zero, then the corresponding endpoint may be more closely approached by `qthsh` with additional abscissas, while `quad` does not. This means that in some cases `qthsh` may produce more accurate results compared to `quad` when the integrand has a significant area to a zero endpoint, such as the integral $\int_0^1 1/\sqrt{x}\ dx$:

#	*f*(*x*)	qthsh improved	quad
8	1/sqrt(x)	(63,1e-13)	(115,6e-9)

In this case, `qthsh` converged quickly at level $k = 3$ to produce a result with a low relative error 10^{-13}, whereas `quad` converged later at level $k = 4$ to a result with a larger relative error close to the specified $eps = 10^{-9}$.

To illustrate this important difference, the following chart shows the point distributions of `qthsh` and `quad` for this integrand, using a log scale to emphasize the points approaching the zero endpoint $a = 0$ showing the *Tanh-Sinh* `qthsh` abscissas versus the `quad` abscissas for $\int_0^1 \frac{1}{\sqrt{x}} dx$ with $eps = 10^{-9}$:



The `quad` points are all $x > 10^{-16}$ due to IEEE 754 double precision with a loss of significant digits in the guard `c-d*r>a` with `a=0` and `c=0.5`. By contrast, the `qthsh` points approach the zero endpoint $a = 0$ until convergence is reached, while testing the guard `a+x>a` to prevent hitting the endpoint (this condition is not the same as the condition `x>0` when `a` is nonzero, some programming languages may require parenthesis `(a+x)>a` to prevent "optimization" of the condition to the incorrect `x>0`).

The WP-34S *Tanh-Sinh* implementation (Appendix B) and the Python mpmath *Tanh-Sinh* implementation do not approach a zero endpoint closely and may suffer slow convergence and/or a loss in accuracy. By contrast, the improved version of `qthsh` and Boost Math *Tanh-Sinh* implementations approach a zero endpoint closely and therefore require fewer function evaluations to integrate this function, producing a result with a low relative error for the given $eps = 10^{-9}$:

#	*f*(*x*)	qthsh improved	WP-34S in C	mpmath	Boost Math
8	1/sqrt(x)	(63,1e-13)	(25,5e-7)	(427,1e-10)	(74,6e-15)

Therefore, it seems reasonable to split *Tanh-Sinh* from *Exp-Sinh* and *Sinh-Sinh* into separate code blocks in the `quad` routine to achieve the same accuracy as `qthsh`.

Furthermore, we can prevent similar inaccuracies in *Exp-Sinh* by quitting the summation when `x==c` rather than checking if `x>a`:

```
x = c + d/r;
if (x == c)
  break;
```

This improved `quad` routine was tested with 1084 integrals. The tests were compared to a `quad` version with $\frac{\pi}{2}$ factors (dropped in the Michalski & Mosig rule) and with $h = 1.5$ as a starting value as suggested by Michalski & Mosig. Both parameters affect the point distributions. In these tests, the simpler `quad` routine more often produced accurate results with fewer points.

A new method to improve Exp-Sinh quadrature convergence

Exp-Sinh uses $d = 1$ by default to split up the interval into a finite part and an infinite part $\int_a^\infty f(x)dx = \int_a^{a+d} f(x)dx + \int_{a+d}^\infty f(x)dx$ and $\int_{-\infty}^b f(x)dx = \int_{-\infty}^{b-d} f(x)dx + \int_{b-d}^b f(x)dx$. This choice of splitting point of the intervals at $a + d$ and $b - d$, respectively, is somewhat arbitrary, but reasonable for most integrands, as we expect a large portion of the integration area to be located close to the finite endpoint. The finite and infinite interval parts are integrated with the same number of points. However, sometimes the infinite part can be much harder to integrate and consequently may require more points to converge. Furthermore, the weights w_j for the infinite part become very big as we increase the number of points that approach the ±inf endpoint. This causes roundoff errors. In this case more accurate results can be produced with fewer points by selecting a larger splitting point *d* depending on $f(x)$. If $f(x)$ has no substantial area close to the finite endpoint of the interval, then enlarging *d* can improve the *Exp-Sinh* convergence. Consider for example $\int_0^1 x^{-0.8}dx = \int_0^\infty e^{-0.2y}dy = 5$ with the change of variable (see also the definite integral #10):

	quad Exp-Sinh with $d = 1$	quad Exp-Sinh with $d = 2$	quad Exp-Sinh with $d = 10$	quad Exp-Sinh with $d = 45$
$\int_0^\infty e^{-0.2y}dy$	(249,2e-10)	(131,9e-10)	(69,2e-9)	(71,exact)

Furthermore, in other cases a fractional *d* may improve the Exp-Sinh convergence. For example, transforming $\int_0^1 x^{-0.8}dx = \int_0^\infty \frac{e^{-0.2/y}}{y^2}dy = 5$ with a change of variable:

	quad Exp-Sinh with $d = 1$	quad Exp-Sinh with $d = .5$	quad Exp-Sinh with $d = .1$	quad Exp-Sinh with $d = \frac{1}{45}$
$\int_0^\infty \frac{e^{-0.2/y}}{y^2}dy$	(249,2e-10)	(131,9e-10)	(69,2e-9)	(71,exact)

Since it is desirable to control the *Exp-Sinh* splitting point, *d* should be an optional argument to the `quad` routine (not shown in the code listings). Likewise, for *Sinh-Sinh* a smarter splitting point instead of the default $c = 0$ can be used to split the two parts of the integration $\int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^{c} f(x)dx + \int_{c}^{\infty} f(x)dx$. This may improve the accuracy of the result when both halves are roughly as costly to integrate. A simple case is when $f(x)$ is symmetric at point $x = c$, i.e. $f(c + \delta) \approx f(c - \delta)$ or $f(c + \delta) \approx -f(c - \delta)$. It is reasonable to assign the optional argument `d` (when specified) to `c` (not shown in the code listings) to improve *Sinh-Sinh*.

The following new method introduced in this article can be used to optimize *d*. The method uses a geometric sequence to probe the symmetry of the $f(x)$ points around the specified *d*. This method adjusts *d* upward or downward to improve the accuracy of *Exp-Sinh*:

```
// return optimized Exp-Sinh integral split point d
double exp_sinh_opt_d(double (*f)(double), double a, double eps, double d) {
  double h2 = f(a + d/2) - f(a + d*2)*4;
  int i = 1, j = 32;     // j=32 is optimal to find r
  if (isfinite(h2) && fabs(h2) > 1e-5) { // if |h2| > 2^-16
    double r, fl, fr, h, s = 0, lfl, lfr, lr = 2;
    do {                 // find max j such that fl and fr are finite
      j /= 2;
      r = 1 << (i + j);
      fl = f(a + d/r);
      fr = f(a + d*r)*r*r;
      h = fl - fr;
    } while (j > 1 && !isfinite(h));
    if (j > 1 && isfinite(h) && sign(h) != sign(h2)) {
      lfl = fl;          // last fl=f(a+d/r)
      lfr = fr;          // last fr=f(a+d*r)*r*r
      do {               // bisect in 4 iterations
        j /= 2;
        r = 1 << (i + j);
        fl = f(a + d/r);
        fr = f(a + d*r)*r*r;
        h = fl - fr;
        if (isfinite(h)) {
          s += fabs(h);  // sum |h| to remove noisy cases
          if (sign(h) == sign(h2)) {
            i += j;      // search right half
          }
          else {         // search left half
            lfl = fl;    // record last fl=f(a+d/r)
            lfr = fr;    // record last fl=f(a+d*r)*r*r
            lr = r;      // record last r
          }
        }
      } while (j > 1);
      if (s > eps) {     // if sum of |h| > eps
        h = lfl - lfr;   // use last fl and fr before the sign change
        r = lr;          // use last r before the sign change
        if (h != 0)      // if last diff != 0, back up r by one step
          r /= 2;
        if (fabs(lfl) < fabs(lfr))
          d /= r;        // move d closer to the finite endpoint
        else
          d *= r;        // move d closer to the infinite endpoint
```

```
      }
    }
  }
  return d;
}
```

The quad routine invokes the optimization method as follows:

```
  else if (isfinite(a)) {
    mode = 1; // Exp-Sinh
    d = exp_sinh_opt_d(f, a, eps, d);
    c = a;
    v = a + d;
  }
  else if (isfinite(b)) {
    mode = 1; // Exp-Sinh
    d = exp_sinh_opt_d(f, b, eps, -d);
    c = b;
    v = b + d;
    sign = -sign;
  }
```

The method finds the value $r = 2^k$, $k = 1, \dots, n$ when $h = r\left(\frac{f\left(a+\frac{d}{r}\right)}{r} - rf(a+rd)\right)$ changes sign. If the sign of *h* changes from negative to positive, then *d* is multiplied by 2^{k-1}. If the sign of *h* changes from positive to negative, then *d* is divided by 2^{k-1}. If $h = 0$ then *d* is multiplied or divided by 2^k. Base 2 is used for efficiency. A different base value increases or decreases the *r* factor gaps to probe $f(x)$. Note that *h* is scaled by r to make its value sequence comparable in scale. This is not required to detect the sign change. Consider for example $\int_0^\infty e^{-0.2y}dy$. *Exp-Sinh* has an optimal *d* around 45. The following table shows the change in sign of *h*, for $d = 1$:

r	$f(a + d/2^k)$	$2^{2k}f(a + 2^k d)$	h
2	0.904837	2.68128	-1.77644
4	0.951229	7.18926	-6.23803
8	0.975310	12.9214	-11.9461
16	0.987578	10.4351	-9.44755
32	0.993769	1.70143	**-0.707665**
64	0.996880	0.0113081	**0.985572**
128	0.998439	1.24877e-07	0.998439
256	0.999219	3.80717e-18	0.999219
512	0.999609	8.84677e-40	0.999609
1024	0.999805	1.19424e-83	0.999805

With $d = 32$ estimated by the method as closer to the optimal *d*, the *Exp-Sinh* quadrature returns the exact integral value 5 for $eps = 10^{-9}$ with 71 points evaluated.

The method was tested with 208 integrals of which 28 were improved in accuracy with a reduction of 63 function evaluations on average to converge with $eps = 10^{-9}$. The overhead of `exp_sinh_opt_d` is low. First the method evaluates 2 or 4 points to determine if a change of sign occurs in the difference *h* for $r \in [2, 2^{16}]$. Bisection evaluates 8 points for a maximum of 12

points cost of this method. Of the 208 integrals, an average of 5.95 points were evaluated by `exp_sinh_opt_d`. See Appendix C for details.

The final version of the improved `quad` routine with separate branches optimized for *Tanh-Sinh*, *Exp-Sinh* and *Sinh-Sinh*, where the *Tanh-Sinh* rule is the same as `qthsh`:

```
// integrate function f, range a..b, max levels n, error tolerance eps
double quad(double (*f)(double), double a, double b, int n, double eps) {
  const double tol = 10*eps;
  double c = 0, d = 1, s, sign = 1, e, v, h = 2;
  int k = 0, mode = 0; // Tanh-Sinh = 0, Exp-Sinh = 1, Sinh-Sinh = 2
  if (b < a) { // swap bounds
    v = b;
    b = a;
    a = v;
    sign = -1;
  }
  if (isfinite(a) && isfinite(b)) {
    c = (a+b)/2;
    d = (b-a)/2;
    v = c;
  }
  else if (isfinite(a)) {
    mode = 1; // Exp-Sinh
              // alternatively d = exp_sinh_opt_d(f, a, eps, d);
    c = a;
    v = a+d;
  }
  else if (isfinite(b)) {
    mode = 1; // Exp-Sinh
    d = -d;   // alternatively d = exp_sinh_opt_d(f, b, eps, -d);
    sign = -sign;
    c = b;
    v = b+d;
  }
  else {
    mode = 2; // Sinh-Sinh
    v = 0;
  }
  s = f(v);
  do {
    double p = 0, q, fp = 0, fm = 0, t, eh;
    h /= 2;
    t = eh = exp(h);
    if (k > 0)
      eh *= eh;
    if (mode == 0) {              // Tanh-Sinh
      do {
        double u = exp(1/t-t);    // = exp(-2*sinh(j*h)) = 1/exp(sinh(j*h))^2
        double r = 2*u/(1+u);     // = 1 - tanh(sinh(j*h))
        double w = (t+1/t)*r/(1+u); // = cosh(j*h)/cosh(sinh(j*h))^2
        double x = d*r;
        if (a+x > a) {            // if too close to a then reuse previous fp
          double y = f(a+x);
          if (isfinite(y))
            fp = y; /             // if f(x) is finite, add to local sum
        }
```

```
        if (b-x < b) {           // if too close to a then reuse previous fp
          double y = f(b-x);
          if (isfinite(y))
            fm = y;              // if f(x) is finite, add to local sum
        }
        q = w*(fp+fm);
        p += q;
        t *= eh;
      } while (fabs(q) > eps*fabs(p));
    }
    else {
      t /= 2;
      do {
        double r = exp(t-.25/t); // = exp(sinh(j*h))
        double x, y, w = r;
        q = 0;
        if (mode == 1) {         // Exp-Sinh
          x = c + d/r;
          if (x == c)            // if x hit the finite endpoint then break
            break;
          y = f(x);
          if (isfinite(y))       // if f(x) is finite, add to local sum
            q += y/w;
        }
        else {                   // Sinh-Sinh
          r = (r-1/r)/2;         // = sinh(sinh(j*h))
          w = (w+1/w)/2;         // = cosh(sinh(j*h))
          x = c - d*r;
          y = f(x);
          if (isfinite(y))       // if f(x) is finite, add to local sum
            q += y*w;
        }
        x = c + d*r;
        y = f(x);
        if (isfinite(y))         // if f(x) is finite, add to local sum
          q += y*w;
        q *= t+.25/t;            // q *= cosh(j*h)
        p += q;
        t *= eh;
      } while (fabs(q) > eps*fabs(p));
    }
    v = s-p;
    s += p;
    ++k;
  } while (fabs(v) > tol*fabs(s) && k <= n);
  e = fabs(v)/(fabs(s)+eps);
  return sign*d*s*h; // result with estimated relative error e
}
```

The final BASIC version of the combined quadrature routine with the *Tanh-Sinh* rule of `qthsh`:

```
' VARIABLES
'  A,B     range, -9E99 and 9E99 are -inf and +inf
'  F$      function label to integrate
'  Y       result with error E
'  E       estimated relative error of the result
'  N       levels (up to 6 or 7)
```

```
'  C       center
'  D       half distance
'  G       sign
'  H       step size h=2^-k
'  K       level counter
'  P,Q,S   quadrature sums
'  T       exp(j*h)
'  L       f(x) point
'  M       f(x) point
'  J,R,U,X scratch
100 "QUAD" ON ERROR GOTO 490: E=1E-9,N=6: INPUT "f=F";F$: F$="F"+F$
110 INPUT "a=";A
120 INPUT "b=";B
' init and swap bounds if b<a
130 D=1,G=1,H=1,K=0: IF B<A LET X=A,A=B,B=X,G=-1
140 IF ABS A<9E99 IF ABS B<9E99 GOTO 180
150 IF ABS A<9E99 LET C=A,X=A+D: GOTO 300
160 IF ABS B<9E99 LET C=B,X=B-D,D=-D,G=-G: GOTO 300
170 GOTO 400
' Tanh-Sinh
180 C=(A+B)/2,D=(B-A)/2,X=C: GOSUB F$: S=Y
' outer loop
190 J=1,P=0,L=0,M=0
' inner loop
200 T=EXP(J*H),U=EXP(1/T-T),R=2*U/(1+U)
210 X=A+D*R,Y=L: IF X>A GOSUB F$: L=Y
220 X=B-D*R,Y=M: IF X<B GOSUB F$: M=Y
230 Q=(T+1/T)*R/(1+U)*(L+M),P=P+Q,J=J+1+(K>0)
240 IF ABS Q>E*ABS P GOTO 200
' exit inner loop
250 X=S-P,S=S+P,K=K+1
260 IF ABS X>10*E*ABS S IF K<=N LET H=H/2: GOTO 190
' exit, output result (and relative error estimate if >E)
270 Y=G*D*S*H,U=ABS X/(ABS S+E)
280 IF U>E LET E=U: PRINT Y,E: END
290 E=U: PRINT Y: END
' Exp-Sinh
300 GOSUB F$: S=Y
' outer loop
310 J=1,P=0
' inner loop
320 T=EXP(J*H)/2,U=EXP(T-.25/T)
330 X=C+D/U: IF X=C GOTO 370
340 GOSUB F$: L=Y/U,X=C+D*U: GOSUB F$: M=Y*U
350 Q=(T+.25/T)*(L+M),P=P+Q,J=J+1+(K>0)
360 IF ABS Q>E*ABS P GOTO 320
' exit inner loop
370 X=S-P,S=S+P,K=K+1
380 IF ABS X>10*E*ABS S IF K<=N LET H=H/2: GOTO 310
390 GOTO 270
' Sinh-Sinh
400 X=0: GOSUB F$: S=Y
' outer loop
410 J=1,P=0
' inner loop
420 T=EXP(J*H)/2,U=EXP(T-.25/T)/2,R=U-.25/U
430 X=-R: GOSUB F$: L=Y,X=R: GOSUB F$: M=Y
440 Q=(T+.25/T)*(U+.25/U)*(L+M),P=P+Q,J=J+1+(K>0)
```

```
450 IF ABS Q>E*ABS P GOTO 420
' exit inner loop
460 X=S-P,S=S+P,K=K+1
470 IF ABS X>10*E*ABS S IF K<=N LET H=H/2: GOTO 410
480 GOTO 270
' The PC-1475 supports ON-ERROR-GOTO and ERN:
290 IF ERN=2 RETURN
```

Note: ON-ERROR-GOTO is not universally supported and can be omitted from the code.

Conclusions

To summarize the conclusions of this article:

- The Michalski & Mosig *Tanh-Sinh* quadrature speed is further improved

- The *Tanh-Sinh* quadrature convergence conditions are improved

- The *Tanh-Sinh* quadrature tolerance is adjusted to prevent early termination to improve the accuracy of the result at a cost of function evaluations, a tradeoff that can be tuned as desired

- The *Tanh-Sinh* quadrature accurately handles singularities (±inf and NaN) close to endpoints

- The *Tanh-Sinh* quadrature is combined with *Exp-Sinh* and *Sinh-Sinh* in one routine

- A new pre-conditioning method is proposed to improve the *Exp-Sinh* quadrature method

My sincere thanks go to Albert Chan for his constructive comments and suggestions he shared with me and the other members on the HP-Forum. I also would like to thank the author of the WP-34S *Tanh-Sinh* quadrature implementation César Rodríguez for his comments he shared on the HP-Forum.

Biography

Dr. Robert A. van Engelen is the CEO/CTO of Genivia.com, a US technology company he founded in 2003. He is a professor in Computer Science and Scientific Computing and worked for 20 years in the department of Computer Science at the Florida State University, where he also served as department chair. He is the single author of the Ctadel code generation system and CAS for partial differential equations, the gSOAP toolkit for C/C++ web services, the ugrep search utility, the RE/flex lexical analyzer, Husky functional programming, Forth500 and many other projects in academia and industry, He published over 70 peer-reviewed technical publications in reputable international conferences and journals, He served as a member of the editorial board on the IEEE Transactions on Services Computing journal and has served on over 40 technical program committees for international conferences/workshops. Van Engelen received the B.S. and the M.S. in Computer Science from Utrecht University, the Netherlands, in 1994 and the Ph.D. in Computer Science from the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, the Netherlands, in 1998. His research interests include High-Performance Computing, Programming Languages and Compilers, Problem-Solving Environments for Scientific Computing, Cloud Computing, Services Computing, Machine Learning, and Bayesian Networks. Van Engelen's research has been recognized with awards and research funding from the US National Science Foundation and the US Department of Energy. Van Engelen is a senior member of the ACM and IEEE professional societies.

References

[1] Takahasi, Hidetosi; Mori, Masatake (1974), "Double Exponential Formulas for Numerical Integration", *Publications of the Research Institute for Mathematical Sciences*, 9 (3): 721–741
http://www.ems-ph.org/journals/show_pdf.php?issn=0034-5318&vol=9&iss=3&rank=12

[2] Mori, Masatake (2005), "Discovery of the Double Exponential Transformation and Its Developments", *Publications of the Research Institute for Mathematical Sciences*, 41 (4): 897–935,
doi:10.2977/prims/1145474600, ISSN 0034-5318

[3] Krzysztof Michalski and Juan Mosig "Efficient computation of Sommerfeld integral tails – methods and algorithms" *Journal of Electromagnetic Waves and Applications* 2016
https://doi.org/10.1080/09205071.2015.1129915

[4] Evans G.A., Forbes R.C., Hyslop J. "The tanh transformation for singular integrals" *International Journal of Computational Mathematics*. 1984;15:339–358

[5] Tanh-Sinh quadrature, Wikipedia, https://en.wikipedia.org/wiki/Tanh-sinh_quadrature

Additional resources

HP-Forum thread: https://www.hpmuseum.org/forum/thread-16549.html

The new `quad` routine in HP PPL: https://www.hpmuseum.org/forum/thread-16635-post-148080.html#pid148080

Test scripts and results to compare methods: https://www.genivia.com/files/qthsh.zip

Article with Visual Basic Tanh-Sinh implementation (see Appendix A):
https://newtonexcelbach.com/2020/10/29/numerical-integration-with-tanh-sinh-quadrature-v-5-0/

WP-34S Tanh-Sinh implementation (see Appendix B):
https://www.hpmuseum.org/forum/thread-8021-post-70927.html
https://github.com/mcesar-rlacruz/py-double-exponential

Boost Math Tanh-Sinh implementation:
https://www.boost.org/doc/libs/1_75_0/libs/math/doc/html/math_toolkit/double_exponential/de_tanh_sinh.html

Boost Math Tanh-Sinh C++ code:
https://github.com/boostorg/math/blob/develop/include/boost/math/quadrature/tanh_sinh.hpp

mpmath Python code: https://github.com/fredrik-johansson/mpmath/blob/master/mpmath/calculus/quadrature.py

Appendix A

Tanh-Sinh V5.0 VB QUAD_TANH_SINH source code (GPL licensed)

```
Option Explicit


'    QUAD_TANH_SINH for finite intervals.
'    This is the fastest and simplest high-performance T-S program I have found.
'
'    It is based on the HP RPN calculator source code listing provided at
'    https://www.hpmuseum.org/forum/thread-8021.html.
'    The code shrunk to two Do loops as shown below.
'
'    The full RPN source code for the WP 34S calculator integration
'    program was written by M. César Rodríguez in 2017 for inclusion in
'    the WP 34s calculator software for numerical integration. It
'    covers the four intervals with four different programs:
'         a. finite interval, (a,b)
'         b. Right semi-infinite interval, (a,inf)
'         c. Left semi-infinite interval, (-inf,b)
'         d. Infinite interval, (-inf,inf)
'
'    The RPN source code selected is the first of the three versions
'    presented on the web page, shown as v1.2r-393 (20170327), and stated
'    as being suitable for keying in by hand. The RPN source code was
'    distilled down to the specific finite interval code which acted
'    as the basis of this present VBA program.
'
' The Tanh-Sinh Transform which transforms the function value is g(z) = f(x(z)) * dxdz(z)

' where   x(z) = (b + a) / 2 + (b - a) / 2 * TANH(SINH(z))
' and     dxdz(z) = (b - a) / 2 * COSH(z) / COSH(SINH(z))^2

' The purpose of the T-S transform is to transform a function over (-1,1) to a new function
' on the entire real line (-inf,inf), where the two integrals have the same value.

' It is the transformed function which is integrated by the T-S integrator
' using Trapezoidal summation.
' Because of the double exponential growth of the denominator in dxdz(z), the +-z sample
' points used for the trapezoidal rule do not have to step very far before g(z) becomes zero
' or sufficiently small for exit and termination. In other words, g(z) approaches zero
' at a double exponential rate. That is the power of the technique used in the T-S method.

' However, we don't calculate x(z) and dxdz(z) directly as above, as the term
' 1/COSH(SINH(z))^2 can overflow for large z.
'
' To prevent this, we calculate x(z) and dxdz(z) this way, courtesy of the Michalski & Mosig T-S
integrator.
' The VBA code translated by the author from the Rodriguez RPN source was modified to use this
technique.
' It improved the accuracy and reduced the execution time.

' 1. Enter value for z (z <= 709)

' 2. exz = exp(z)

' 3. q = exp(-2 * sinh(z))
'      = exp(-2 * (exz - 1/exz) / 2)
'      = exp(-(exz - 1/exz))
'      = exp(1/exz - exz)

' 4. delta = 2 * q / (1 + q) = (1 - tanh(sinh z))

' 5. x(z) = (b + a) / 2 + (b - a) / 2 * delta

' 6. fxz = f(x(z))

' 7. dxdz = dxdz(z) =  (b - a) / 2 * (exz + 1/exz) * delta / (1 + q)

' 8. The transformed function = y = fxz * dxdz

' Now, for large positive z (6.5 < z < 709), q simply underflows harmlessly to zero.
' Negative z is handled by using the symmetry of the trapezoidal rule about
' the midpoint of the interval.
```

```
Function QUAD_TANH_SINH(func As String, Symbols As Variant, Values As Variant, _
                                  Parms As Variant, Optional evaltype As Long = 1) As Variant

    Dim Result(1 To 4) As Variant, c As String
    Dim Parmscount As Long, k As Long, a As Double, b As Double, i As Long
    Dim errval As Double, maxlevel As Long, bpa2 As Double, bma2 As Double
    Dim eps As Double, tol As Double, expu As Double, sshp As Double, h As Double
    Dim n As Long, j As Long, t As Double, w As Double, r As Double, fp As Double
    Dim fm As Double, p As Double, expt As Double, u As Double, delta As Double
    Dim ssp As Double, ss As Double, sslast As Double, evals As Long, x As Double
    Dim ScreenUpdateState As Boolean, StatusBarState As Boolean
    Dim CalcState As Boolean, EventsState As Boolean, AlertsState As Boolean

    ' Error management
    On Error Resume Next

    '   Disable unwanted events so the code runs faster.
    '   Get the current status settings
    ScreenUpdateState = Application.ScreenUpdating
    StatusBarState = Application.DisplayStatusBar
    CalcState = Application.Calculation
    EventsState = Application.EnableEvents
    AlertsState = Application.DisplayAlerts

    '   Turn the settings off
    Application.ScreenUpdating = False
    Application.DisplayStatusBar = False
    Application.Calculation = xlCalculationManual
    Application.EnableEvents = False
    Application.DisplayAlerts = False

    ' Load the input parameters into arrays for access
    ' Load the input parameters into arrays for access
    GetArray Symbols                    ' Get the Symbols array.
    GetArray Values                     ' Get the values array.
    GetArray Parms                      ' Get the parameters array.

    ' Do some parameter checks.
    ' Prepare Result() array for possible error exit
    Result(1) = ""
    Result(2) = ""
    Result(3) = ""
    Result(4) = ""

    ' Check the fourth input cell range for min. amd max. parameter count
    Parmscount = UBound(Parms) - LBound(Parms) + 1

    ' If not correct parameters count, exit with message
    If Parmscount <> 3 Then             ' 3 parameters required
        Result(1) = "** ERROR: 3 params. reqd **"
        GoTo exithere
    End If

    If func = "" Then
        Result(1) = "** ERROR: No integrand! **"
        GoTo exithere
    End If

    If IsEmpty(Parms(1, 1)) Then
        Result(1) = "** ERROR: No variable! **"
        GoTo exithere
    End If

    If IsEmpty(Parms(2, 1)) Then
        Result(1) = "** ERROR: No lower limit! **"
        GoTo exithere
    End If

    If IsEmpty(Parms(3, 1)) Then
        Result(1) = "** ERROR: No upper limit! **"
        GoTo exithere
    End If

    ' Ensure the upper limit is greater than the lower limit
    If Parms(3, 1) < Parms(2, 1) Then
        Result(1) = "** ERROR: Check limit values ***"
        GoTo exithere
```

```
    End If

    ' Ensure the variable occurs at least once in the function!
    If evaltype = 1 Then
        If InStr(func, Parms(1, 1)) = False Then         ' if not found then exit with message
            Result(1) = "Variable not in Function"
            GoTo exithere
        End If
    End If

    ' Replace the Symbols in the function with the specified Values
    If evaltype = 1 Then
        For i = 1 To UBound(Symbols, 1) - LBound(Symbols, 1) + 1
            func = Replace(func, Symbols(i, 1), Values(i, 1))
        Next i
    End If

    ' Get the two limits of integration and the integrating variable:
    a = Parms(2, 1)     ' lower limit
    b = Parms(3, 1)     ' upper limit
    c = Parms(1, 1)     ' intvar

    ' Get timer count
    Result(4) = MicroTimer      ' precision timer for results

    ' Set some program constants. These are the optimum values.
    ' epsilon
    eps = 10 ^ -15                  ' eps = 10^-15 is OPTIMUM!
    ' convergence tolerance
    tol = 10 ^ -7                   ' tol = 10^-7 is OPTIMUM!
    ' max level
    maxlevel = 6                    ' maxlevel = 6 is OPTIMUM!  A slightly more accurate result with
maxlevel = 7 but takes longer.

    bma2 = (b - a) / 2   ' interval half-length
    bpa2 = (b + a) / 2   ' centre of interval
    k = 0                ' level counter
    ss = EvalFunc(func, c, bpa2, evaltype, Values)   ' centre of interval
    evals = 1

    Do
        ssp = 0
        j = 1
        h = 2 ^ -k

        Do
            t = h * j

            If t > 6.56 Then Exit Do

            expt = Exp(t)

            u = Exp(1 / expt - expt)   ' = exp(-(expt - 1/expt)) = exp(-2 (expt -1/expt) /2) = exp(-2
sinh t)

            r = 2 * u / (1 + u)        '  r = 1 - tanh(sinh t)

'           Added so as to check that r <> 0 and r <> 1 to ensure r hasn't rounded to 0 or 1 when
tanh(sinh(t)) is very close to 0 or 1.
            If r <> 0 And r <> 1 Then x = bma2 * r Else Exit Do

'           Added to check that (a + x) > a to ensure (a + x) hasn't rounded to a when x is very small.
'           This prevents calculation of the function at lower limit a.

            If (a + x) > a Then
                fp = EvalFunc(func, c, (a + x), evaltype, Values)
                evals = evals + 1
            End If

'           Added to check that (b - x) < b to ensure (b - x) hasn't rounded to b when x is very small.
'           This prevents calculation of the function at upper limit b.

            If (b - x) < b Then
                fm = EvalFunc(func, c, (b - x), evaltype, Values)
                evals = evals + 1
            End If
```

```
            w = (expt + 1 / expt) * r / (1 + u)   ' w = cosh t / cosh^2 (sinh t)     See separate
proof.

            p = (fp + fm) * w
            ssp = ssp + p
            If k > 0 Then j = j + 2 Else j = j + 1

        Loop While Abs(p) > Abs(ssp * eps)

        ss = ss + ssp
        errval = Abs(2 * sslast / ss - 1)
        sslast = ss
        k = k + 1

    Loop While errval >= tol And k <= maxlevel
    Result(1) = ss * bma2 * h             ' load the integral result,
    Result(2) = errval                    ' its error value,
    Result(3) = evals                     ' func evals
    Result(4) = MicroTimer - Result(4)    ' and the calculation time

exithere:

    QUAD_TANH_SINH = Result       ' Return the results as an array.

    ' 3.  Reset the settings back to their original state
    Application.ScreenUpdating = ScreenUpdateState
    Application.DisplayStatusBar = StatusBarState
    Application.Calculation = CalcState
    Application.EnableEvents = EventsState
    Application.DisplayAlerts = AlertsState

End Function
```

Tanh-Sinh Article's V5.0 VB QUAD_DE1 source code (GPL)

```
Option Explicit



' This (a,inf) integration program was translated from the
' RPN program in the WP34S calculator, written by M. Cesar Rodriguez in 2017.
'



Function QUAD_DE1(func As String, Symbols As Variant, Values As Variant, _
                      Parms As Variant, Optional evaltype As Long = 1) As Variant

    Dim Result(1 To 4) As Variant, c As String
    Dim Parmscount As Long, k As Long, i As Long, maxlevel As Long
    Dim eps As Double, tol As Double, tm As Double, h As Double
    Dim j As Long, t As Double, w As Double, r As Double, fp As Double
    Dim fm As Double, ssp As Double, expt As Double, u As Double
    Dim expu As Double, evals As Long, ss As Double, sslast As Double
    Dim a As Double, p As Double, ct As Double, errval As Double
    Dim ScreenUpdateState As Boolean, StatusBarState As Boolean
    Dim CalcState As Boolean, EventsState As Boolean, AlertsState As Boolean

    ' Error management
    On Error Resume Next

    '   Disable unwanted events so the code runs faster.
    '   Get the current status settings
    ScreenUpdateState = Application.ScreenUpdating
    StatusBarState = Application.DisplayStatusBar
    CalcState = Application.Calculation
    EventsState = Application.EnableEvents
    AlertsState = Application.DisplayAlerts

    '   Turn the settings off
    Application.ScreenUpdating = False
    Application.DisplayStatusBar = False
    Application.Calculation = xlCalculationManual
    Application.EnableEvents = False
    Application.DisplayAlerts = False
```

```
' Load the input parameters into arrays for access
' Load the input parameters into arrays for access
GetArray Symbols                    ' Get the Symbols array.
GetArray Values                     ' Get the values array.
GetArray Parms                      ' Get the parameters array.

' Do some parameter checks.
' Prepare Result() array for possible error exit
Result(1) = ""
Result(2) = ""
Result(3) = ""
Result(4) = ""

' Check the fourth input cell range for min. amd max. parameter count
Parmscount = UBound(Parms) - LBound(Parms) + 1

' If not correct parameters count, exit with message
If Parmscount <> 2 Then             ' 1 parameter required
    Result(1) = "** ERROR: 2 param. reqd **"
    GoTo exithere
End If

If func = "" Then
    Result(1) = "** ERROR: No integrand! **"
    GoTo exithere
End If

If IsEmpty(Parms(1, 1)) Then
    Result(1) = "** ERROR: No variable! **"
    GoTo exithere
End If

' Ensure the variable occurs at least once in the function!
If evaltype = 1 Then
    If InStr(func, Parms(1, 1)) = False Then         ' if not found then exit with message
        Result(1) = "Variable not in Function"
        GoTo exithere
    End If
End If

' Replace the Symbols in the function with the specified Values
If evaltype = 1 Then
    For i = 1 To UBound(Symbols, 1) - LBound(Symbols, 1) + 1
        func = Replace(func, Symbols(i, 1), Values(i, 1))
    Next i
End If

' Get the integrating parameters:
a = Parms(2, 1)    ' lower limit
c = Parms(1, 1)    ' intvar

' Get timer count
Result(4) = MicroTimer      ' precision timer for results

' Set some program constants. These are the optimum values.
' epsilon
eps = 10 ^ -14                    ' eps = 10^-14 is OPTIMUM!
' convergence tolerance
tol = 10 ^ -8                     ' tol = 10^-8 is OPTIMUM!
' max level
maxlevel = 6                      ' maxlevel = 6 is OPTIMUM!

ss = EvalFunc(func, c, (a + 1), evaltype, Values)
evals = 1
k = 0

Do
    ssp = 0
    j = 1
    h = 2 ^ -k

    Do
        t = h * j
        If t > 6.56 Then Exit Do
        expt = Exp(t)
        ct = (expt + 1 / expt) / 2                ' = cosh t
        r = Exp(PIon2 * (expt - 1 / expt) / 2)     ' = node = exp(pi/2 sinh t)
```

```
            If r = 0 Then Exit Do
            w = r                                          ' = weight
            fp = EvalFunc(func, c, (a + r), evaltype, Values) * w
            fm = EvalFunc(func, c, (a + 1 / r), evaltype, Values) / w
            evals = evals + 2
            p = (fp + fm) * ct
            ssp = ssp + p
            If k > 0 Then j = j + 2 Else j = j + 1
            If Abs(ssp * eps) >= Abs(p) Then Exit Do

        Loop

        ss = ss + ssp
        errval = Abs(2 * sslast / ss - 1)
        If errval < tol Then Exit Do
        sslast = ss
        k = k + 1

    Loop While k <= maxlevel

    Result(1) = ss * h * PIon2                ' so load the integral result,
    Result(2) = errval                        ' its error value,
    Result(3) = evals                         ' func evals
    Result(4) = MicroTimer - Result(4)        ' and the calculation time

exithere:

    QUAD_DE1 = Result       ' Return the results as an array.

    ' 3.  Reset the settings back to their original state
    Application.ScreenUpdating = ScreenUpdateState
    Application.DisplayStatusBar = StatusBarState
    Application.Calculation = CalcState
    Application.EnableEvents = EventsState
    Application.DisplayAlerts = AlertsState

End Function
```

Appendix B

Tanh-Sinh C source code derived from the WP-34S Tanh-Sinh Python code (MIT licensed)

Used in comparisons using identical parameters as qthsh except $eps = 10^{-15}$ to prevent early termination with large errors due to `thr` = $10\sqrt{eps} = 3 \cdot 10^{-4}$ allowing results with larger errors.

```
double wp34s(double (*f)(double), double a, double b, int n, double eps) {
  double thr = 10*sqrt(eps); // too generous for larger eps, e.g. eps=1e-9
  double c = (a+b)/2; // center (mean)
  double d = (b-a)/2; // half distance
  double s = f(c);
  double fp = 0, fm = 0;
  double p, e, v, h = 2;
  double tmax = log(2/M_PI * log((d < 1 ? 2*d : 2) / eps));
  int k = 0; // level
  do {
    double q, t;
    int j = 1;
    v = s*d*M_PI/2*h; // last sum
    p = 0;
    h /= 2;
    t = h;
    do {
      double ch = cosh(t);
      double ecs = cosh(M_PI/2 * sqrt(ch*ch - 1)); // = cosh(pi/2*sinh(t))
      double w = 1/(ecs*ecs);
      double r = sqrt(ecs*ecs - 1)/ecs;
      double x = d*r;
      if (c+x > a) {
        double y = f(c+x);
        if (isfinite(y))
          fp = y;
      }
      if (c-x < b) {
        double y = f(c-x);
        if (isfinite(y))
          fm = y;
      }
      q = ch*w*(fp+fm);
      p += q;
      j += 1+(k>0);
      t = j*h;
    } while (t <= tmax && fabs(q) > eps*fabs(p));
    s += p;
    ++k;
  } while (s && fabs(2*fabs(p) - fabs(s)) >= fabs(thr*s) && k <= n);
  s *= d*M_PI/2*h;
  e = fabs(v-s);
  if (10*e >= fabs(s)) {
    e += fabs(s);
    s = 0;
  }
  return s; // result with estimated absolute error e
}
```

Appendix C

Exp-Sinh `exp_sinh_opt_d` optimized *d* results for 208 integrals, *eps*=1e-9 (GPL licensed)

integrand	a	b	exact integral	extra points	predicted d	exp-sinh integral	exp-sinh levels	points	rel.err	abs.err	ACCURACY	EVAL COST	BETTER BY	WORSE BY
exp(-x)/x	1	INFINITY	0.219383934	4	1	0.219383934	4	129	3.50568E-11	4.53104E-12			0	0
1/(1+(x*x))	0	INFINITY	1.570796327	2	1	1.570796327	3	69	1.82635E-13	2.8999E-13			0	0
exp(-x)/sqrt(x)	0	INFINITY	1.772453851	4	1	1.772453851	4	155	1.40948E-11	5.72697E-12			0	0
exp(-1-x)/(1+x)	0	INFINITY	0.219383934	4	1	0.219383934	4	131	3.50568E-11	4.53104E-12			0	0
(x*x)*exp(-4*x)	0	INFINITY	0.03125	4	1	0.03125	4	99	1.06641E-11	3.36592E-13			0	0
pow(sqrt((x*x)+9)-x,3)	0	INFINITY	30.375	12	2	30.375	3	69	2.57082E-13	7.60991E-12			0	0
exp(-3*x)	0	INFINITY	0.333333333	4	1	0.333333333	4	133	1.10878E-12	5.2075E-13			0	0
x*exp(-2*x)/(exp(-x)+1)	0	INFINITY	0.177532967	12	2	0.177532967	4	111	1.22271E-11	2.19094E-12			0	0
log(x)/pow(x+3,2)	0	INFINITY	0.366204096	4	1	0.366204096	3	73	1.44975E-11	7.21645E-16			0	0
log(1+exp(-x))	0	INFINITY	0.822467033	12	2	0.822467033	3	69	2.86158E-09	1.93923E-12	IMPROVED	BETTER	62	0
log(1+x)/pow(3*x+4,2)	0	INFINITY	0.095894024	4	1	0.095894024	3	73	3.03912E-15	6.93889E-17			0	0
log(1+(x*x))/x/(1+(x*x))	0	INFINITY	0.822467033	4	1	0.822467033	3	61	6.61436E-15	4.996E-15			0	0
log((1+(x*x))/x)/(1+(x*x))	0	INFINITY	2.17758609	2	1	2.17758609	3	73	5.09841E-15	0			0	0
exp(-3*x)/sqrt(x)	0	INFINITY	1.023326708	4	1	1.023326708	3	81	3.12507E-09	4.95159E-14			0	0
(log(1+16*(x*x))-log(1+9*(x*x)))/(x*x)	0	INFINITY	3.141592654	4	1	3.141592642	4	129	8.6277E-09	1.14491E-08			0	0
exp(-4*(x*x))	0	INFINITY	0.443113463	4	1	0.443113463	4	131	2.73334E-11	1.23175E-11			0	0
x*exp(-x)/(exp(x)-1)	0	INFINITY	0.644934067	4	1	0.644934067	4	131	1.21645E-11	1.23166E-11			0	0
x/(1+(x*x))/sinh(pi*x)	0	INFINITY	0.193147181	4	1	0.193147181	3	69	4.45705E-09	4.14946E-14			0	0
(1-(x*x))*log(x)/pow(1+(x*x),2)	0	INFINITY	-1.570796327	2	1	-1.570796327	3	73	6.07839E-15	5.55112E-15			0	0
log(1+x)/x/(1+x)	0	INFINITY	1.644934067	4	1	1.644934067	3	71	2.56475E-15	4.21885E-15			0	0
log(1+(x*x))/(x*x)	0	INFINITY	3.141592654	4	1	3.141592633	3	71	4.37857E-09	2.07544E-08			0	0
x*exp(-x)*sqrt(1-exp(-2*x))	0	INFINITY	0.937095604	12	8	0.937095604	3	57	2.55155E-10	6.68354E-14	IMPROVED	BETTER	44	0
pow(sqrt((x*x)+4)-x,3)	0	INFINITY	6	4	1	6	3	69	2.8629E-13	1.14042E-12			0	0
1/pow(sqrt((x*x)+4)+x,3)	0	INFINITY	0.09375	4	1	0.09375	3	69	2.84217E-13	1.79023E-14			0	0
exp(-2*(x*x)-3/(x*x))	0	INFINITY	0.004671219	12	1	0.004671219	4	43	1.02792E-09	1.73472E-18			0	0
x*exp(-3*x)/(exp(-x)+1)	0	INFINITY	0.072467033	4	1	0.072467033	4	111	5.61147E-12	5.47645E-13			0	0
x*(1-exp(-x))*exp(-x)/(exp(-3*x)+1)	0	INFINITY	0.731081807	12	8	0.731081807	4	103	2.94973E-12	2.20213E-12	IMPROVED	WORSE	0	-8
(x*exp(-2*x))/sqrt(exp(x)-1)	0	INFINITY	0.258742782	4	1	0.258742782	4	121	1.85712E-11	3.83832E-12			0	0
log(3*x)/(16+(x*x))	0	INFINITY	0.975820559	4	1	0.975820559	3	73	4.19965E-09	1.11022E-15			0	0
x*atan(x)/(1+(x*x*x*x))	0	INFINITY	0.616850275	4	1	0.616850275	3	57	5.53189E-10	6.04294E-13			0	0
1/(1+pow(sqrt(x),3))	0	INFINITY	2.418399152	4	1	2.418399152	3	81	1.9832E-14	4.79616E-14			0	0
pow(log(x),2)/((x*x)+x+1)	0	INFINITY	3.536095247	2	1	3.536095247	3	73	2.32337E-14	8.88178E-14			0	0
pow(sqrt((x*x)+25)-x,7)	0	INFINITY	56966.14583	4	1	56966.14583	3	69	6.61227E-09	1.11613E-08			0	0
1/pow(x+sqrt((x*x)+25),7)	0	INFINITY	9.33E-06	2	1	9.33333E-06	3	69	6.61218E-09	1.82959E-18			0	0
1/(6+(x*x))/pow(11+(x*x),2)	0	INFINITY	0.002400908	12	2	0.002400908	3	69	8.38765E-12	3.98553E-16	IMPROVED	BETTER	62	0
(exp(-6*x)-exp(-5*x))/(1-exp(-(11)*x))	0	INFINITY	-0.041062985	4	1	-0.041062985	3	69	3.00213E-09	1.29827E-14			0	0
exp(-25*(x*x))	0	INFINITY	0.177245385	4	1	0.177245385	3	71	8.26093E-09	3.88578E-16			0	0
exp(-5*(x*x)-6/(x*x))	0	INFINITY	6.93E-06	2	1	6.92791E-06	4	39	1.09788E-09	5.92923E-21			0	0
exp(-5*x)/sqrt(1+x)	-1	INFINITY	117.641985	4	1	117.6419835	4	139	2.0224E-09	1.49243E-06			0	0
x*exp(-x)/(exp(x)-1)	0	INFINITY	0.644934067	4	1	0.644934067	4	131	1.21645E-11	1.23166E-11			0	0
x*exp(-2*x)/(exp(-x)+1)	0	INFINITY	0.177532967	12	2	0.177532967	4	111	1.22271E-11	2.19094E-12			0	0
x*(exp(-x)+1)/(exp(x)-1)	0	INFINITY	2.289868134	12	2	2.289868134	4	131	1.39504E-11	4.92668E-11			0	0
(exp(-5*x)+exp(-x))/(1-exp(-6*x))*x	0	INFINITY	1.096622711	12	4	1.096622711	4	131	1.47296E-11	1.6422E-11			0	0
(exp(-6*x)-exp(-2*x))/(exp(-8*x)+1)/x	0	INFINITY	-0.881373587	4	1	-0.881373587	4	131	2.7588E-11	2.46348E-11			0	0
(exp(-5*x)-exp(-8*x))*(exp(-4*x)-exp(-3*x))*exp(-x)/x	0	INFINITY	-0.025317808	4	1	-0.025317808	3	61	3.43753E-11	3.7817E-16			0	0
(exp(x)-exp(-x)+2)/pow(exp(x)-1,2)*(x*x)	0	INFINITY	4.579736267	12	4	4.579736267	4	131	2.1251E-11	9.85461E-11			0	0
(1.0/2.0-1/(exp(-x)+1))*exp(-2*x)/x	0	INFINITY	-0.120782238	4	1	-0.120782238	4	131	2.50478E-11	3.07898E-12			0	0
(1.0/2.0*exp(-2*x)-1/(exp(x)+1))/x	0	INFINITY	-0.572364943	4	1	-0.572364943	4	131	3.21837E-11	9.2375E-12			0	0
((x+1)*exp(-x)-exp(-x/2))/x	0	INFINITY	0.306852819	4	1	0.306852819	4	131	9.80877E-10	6.15991E-12			0	0
x/(1+(x*x))/sinh(pi*x/4)	0	INFINITY	1.467891949	4	1	1.467891949	4	131	2.63136E-11	1.56781E-11			0	0
pow(sinh(3*x),2)/sinh(10*x)/x	0	INFINITY	0.265696807	8	1	0.265696807	3	69	2.24353E-09	1.29008E-13			0	0
pow(sinh(x/2),2)/cosh(x)*exp(-x)/x	0	INFINITY	0.120782238	12	2	0.120782238	4	111	8.93181E-12	1.09514E-12		WORSE	0	-2
1/sqrt(x)/(1+x)	0	INFINITY	3.141592654	2	1	3.141592654	3	81	3.10988E-14	9.4591E-14			0	0
(x*x)*log(x)/(9+16*(x*x))/(1+(x*x))	0	INFINITY	0.04841678	4	1	0.04841678	3	73	4.09883E-14	2.08167E-17			0	0
log(x)/((x*x)+16)	0	INFINITY	0.544396523	4	1	0.544396523	3	73	8.83109E-09	1.44329E-15			0	0
(atan(pi*x)-atan(x))/x	0	INFINITY	1.7981375	4	1	1.7981375	3	69	3.88437E-12	4.06786E-13			0	0
(x*x*x)/(exp(x)-1)	0	INFINITY	6.493939402	12	16	6.493939402	4	103	5.42486E-12	3.52278E-11	IMPROVED	BETTER	70	0
exp(-x)/(1+(x*x*x*x))	0	INFINITY	0.630477835	4	1	0.630477835	3	69	9.76194E-10	1.45661E-13			0	0
1/pow(x+1,1.25)	0	INFINITY	4	4	1	4	3	91	3.44169E-15	1.42109E-14			0	0
exp(-x)/(2*x+100)	0	INFINITY	0.009807555	4	1	0.009807555	4	131	3.36067E-11	1.23164E-13			0	0
x/(exp(x)-1)	2	INFINITY	0.431039498	12	2	0.431039498	4	129	1.13413E-11	7.71122E-12			0	0
(pow(x,0.4)-pow(3,0.4))/(x-3)*(pow(x,0.4)-1)/(x-1)	0	INFINITY	2.910898989	4	1	nan	0	13	nan	nan	N/A		0	0
(x*x)*(1-x)/(1-pow(x,6))	0	INFINITY	0.302299894	4	1	nan	0	9	nan	nan	N/A		0	0
pow(x-1,-0.25)/x	1	INFINITY	4.442882938	4	1	4.442500351	6	551	6.22215E-06	0.000382587	N/A		0	0
1/(2-3*x)/pow(x-1,0.5)	1	INFINITY	-1.813799364	4	1	-1.813799342	3	69	5.2076E-09	2.23237E-08			0	0
pow(sqrt((x*x)+4)-x,3)	0	INFINITY	6	4	1	6	3	69	2.8629E-13	1.14042E-12			0	0
1/(1+(x*x)+(x*x*x*x)/(1+exp(-x)))	0	INFINITY	0.941290135	4	1	0.941290135	3	69	6.97382E-10	1.4555E-13			0	0
exp(-sqrt(x))	0	INFINITY	2	12	64	2	3	73	3.84226E-12	1.33227E-15	IMPROVED	BETTER	58	0
1/((x*x)+exp(4*x))	0	INFINITY	0.246187595	4	1	0.246187595	3	69	3.12265E-09	1.44218E-13			0	0
1/(pow(x,2.0/3.0)+pow(x,3.0/2.0))	0	INFINITY	3.963919199	4	1	3.963919199	3	87	7.8423E-16	3.55271E-15			0	0
log(x)/(1+100*(x*x))	0	INFINITY	-0.361689221	4	1	-0.361689221	4	139	2.60022E-12	3.43614E-13			0	0
x*sqrt(1+x)/pow(1+x,13)	0	INFINITY	0.008281573	4	1	0.008281573	3	61	7.89065E-11	1.14492E-16			0	0
1/(x+1)/sqrt(x)	0	INFINITY	3.141592654	2	1	3.141592654	3	81	3.10988E-14	9.4591E-14			0	0
1/sqrt(x-3)/(x-2)	3	INFINITY	3.141592654	2	1	3.141592587	5	273	8.17189E-09	6.6719E-08			0	0
pow(sqrt((x*x)+9)-x,4)	0	INFINITY	64.8	4	1	64.8	3	69	5.07485E-11	1.16103E-11			0	0
x/(1+(x*x*x))	0	INFINITY	1.209199576	4	1	1.209199576	3	69	1.18808E-13	1.48104E-13			0	0
1/(3+(x*x))/pow(5+(x*x),2)	0	INFINITY	0.015980479	4	1	0.015980479	3	69	6.48501E-10	1.97065E-15			0	0
(pow(x,0.5)-pow(x,0.25))/(x-1)/(x+2)	0	INFINITY	0.76698197	4	1	nan	0	11	nan	nan	N/A		0	0
(pow(x,0.4)-pow(3,0.4))/(x-3)*(pow(x,0.4)-1)/(x-1)	0	INFINITY	2.910898989	4	1	nan	0	13	nan	nan	N/A		0	0
(x*x)*(1-x)/(1-pow(x,6))	0	INFINITY	0.302299894	4	1	nan	0	9	nan	nan	N/A		0	0
exp(-x)	0	INFINITY	1	4	1	1	4	131	6.25233E-12	1.23168E-11			0	0
x/sinh(x)	0	INFINITY	2.4674011	12	4	2.4674011	4	131	1.96716E-11	4.9269E-11			0	0
pow(log(x),2)/(1+(x*x))	0	INFINITY	3.875784585	2	1	3.875784585	3	73	1.29476E-14	9.01501E-14			0	0
(1+(x*x))/(1+(x*x*x*x))	0	INFINITY	2.221441469	2	1	2.221441469	3	69	2.76234E-10	2.83329E-13			0	0
log(x)/pow(cosh(x),2)	0	INFINITY	-0.81878014	4	1	-0.81878014	4	137	2.58807E-11	1.31019E-11			0	0
log(1+(x*x*x))/(1-x+(x*x))	0	INFINITY	3.985324541	4	1	3.985324541	3	71	4.34581E-15	0			0	0
x*exp(-x)*sqrt(1-exp(-2*x))	0	INFINITY	0.937095604	12	8	0.937095604	3	57	2.55155E-10	6.68354E-14	IMPROVED	BETTER	44	0
1/pow(x,3.0/2.0)*sin(x/2)*exp(-x)	0	INFINITY	0.861179089	4	1	0.861179089	4	155	9.91208E-10	2.86304E-12			0	0
pow(sin(2*x),3)/(x*x*x)	0	INFINITY	4.712539269	4	1	4.712519269	6	493	6.64172E-05	0.000130289	N/A		0	0
exp(-x)*cos(x)	0	INFINITY	0.5	4	1	0.5	5	253	6.47433E-10	7.54214E-11			0	0
exp(-4*x)*sin(3*x)/x	0	INFINITY	0.643501109	4	1	0.643501109	4	133	1.93439E-12	1.56319E-12			0	0
(4*pow((3+x),2)+20)/pow((3+x),6)*sin(2*x)	0	INFINITY	0.024691358	4	1	0.024695677	6	411	0.00118293	4.31938E-06	N/A		0	0
(4*pow((3+x),2)+20)/pow((3+x),6)*cos(2*x)	0	INFINITY	0.016460905	4	1	0.016457596	6	501	0.00124195	3.30895E-06	N/A		0	0
pow(sin(3*x),6)/pow(x,6)	0	INFINITY	209.9369291	4	1	209.936929	6	499	4.69737E-09	8.93093E-08			0	0
pow(sin(3*x),3)/(x*x*x)	0	INFINITY	10.60287521	4	1	10.60282068	6	499	5.13072E-05	5.45224E-05	N/A		0	0
exp(-5*x)*cos(5*x)*x/(pow(4,4)+(x*x*x*x))	0	INFINITY	2.55E-14	12	2	2.55427E-14	6	505	2.51006E-10	7.08736E-21	IMPROVED	WORSE	0	-6
sin(x)/(cosh(3*x)+cos(x))*x/((x*x)-pi*pi)	0	INFINITY	-0.011582779	4	1	-0.011582779	4	99	9.24515E-13	1.70107E-14			0	0
x*exp(-25*(x*x))*sin(3*x)	0	INFINITY	0.009719405	4	1	0.009719405	4	103	2.6549E-12	2.58057E-14			0	0
(x*x)*exp(-25*(x*x))*cos(3*x)	0	INFINITY	0.002656637	4	1	0.002656637	4	103	3.23795E-12	8.59686E-15			0	0
(x*x*x)*exp(-25*(x*x))*sin(3*x)	0	INFINITY	5.48E-04	4	1	0.000548174	4	91	3.33464E-13	1.83339E-16			0	0
exp(-5*x)*sin(6*x)/x	0	INFINITY	0.876058051	4	1	0.876058051	4	135	4.03904E-10	2.69629E-12			0	0
(sin(x)/x)*(sin(x/3)*3/x)*(sin(x/5)*5/x)	0	INFINITY	1.570796327	12	0.5	1.570820294	6	487	0.000627074	0.000023967	IMPROVED	BETTER	2	0
pow(x,(3.0/2.0))*sin(x/2)*exp(-x)	0	INFINITY	0.861179089	4	1	0.861179089	4	155	9.91208E-10	2.86304E-12			0	0
atan(sqrt((x*x)+9))/sqrt((x*x)+9)/((x*x)+1)	0	INFINITY	0.55672851	4	1	0.55672851	3	69	4.25724E-11	5.93969E-14			0	0
pow(sin(x),2)/(x*x)	0	INFINITY	1.570796327	4	1	1.572062641	6	491	0.0022929	0.00126631	N/A		0	0
sin(x)/x/((x*x)+9)	0	INFINITY	0.165843443	12	0.5	0.165830036	6	487	0.000603735	1.34064E-05	N/A	BETTER	2	0
(1+(x*x))/(1+(x*x*x*x))	0	INFINITY	2.221441469	2	1	2.221441469	3	69	2.76234E-10	2.83329E-13			0	0

x/sinh(x)	0	INFINITY	2.4674011	12	4	2.4674011	4	131	1.96716E-11	4.9269E-11			0	0
pow(log(x),2)/(1+(x*x))	0	INFINITY	3.875784585	2	1	3.875784585	3	73	1.29476E-14	9.01501E-14			0	0
x*exp(-x)*sqrt(1-exp(-2*x))	0	INFINITY	0.937095604	12	8	0.937095604	3	57	2.55155E-10	6.68354E-14	IMPROVED	BETTER	44	0
(x*x)/sqrt(exp(x)-1)	0	INFINITY	16.3729762	12	32	16.3729762	4	109	3.56726E-13	7.28662E-12	IMPROVED	BETTER	72	0
sin(3*x)*sin(2*x)/(x*x)	0	INFINITY	3.141592654	4	1	3.144406214	6	495	0.00324789	0.00281356	N/A		0	0
cos(x)/(1+x*x)	0	INFINITY	0.577863675	4	1	0.580570089	6	479	0.000480726	0.00270641	N/A		0	0
pow(sin(3*x),3)*sin(12*x)/(x*x*x*x)	0	INFINITY	42.41150082	12	2	42.41153139	6	509	2.53111E-06	3.05701E-05	N/A	WORSE	0	-4
log(((x*x)+4)/((x*x)+1))*cos(x)	0	INFINITY	0.730559018	4	1	0.738676754	6	495	0.00113395	0.00811774	N/A		0	0
cos(3*x)*(1-(x*x))/pow(1+(x*x),2)	0	INFINITY	0.234616032	4	1	0.228083159	6	483	0.0308984	0.00653287	N/A		0	0
pow(sin(3*x),5)/(x*x)	0	INFINITY	1.580999019	12	0.0625	1.552265251	6	393	0.00137713	0.0287338	N/A	BETTER	28	0
sin(3*x)/(1+(x*x))*tanh(pi*x/2)	0	INFINITY	0.174223861	12	0.0625	0.15618316	6	513	0.192739	0.0180407	N/A	WORSE	0	-14
sin(x)/x*sin(x/3)*3/x	0	INFINITY	1.570796327	4	1	1.564126167	6	491	0.00942204	0.00667016	N/A		0	0
pow(cos(x),2)/(x*x)-cos(2*x)/(x*x)	0	INFINITY	1.570796327	4	1	1.572062645	6	483	0.0022929	0.00126632	N/A		0	0
x*sin(3*x)/pow(4+(x*x),2)	0	INFINITY	0.002920211	12	0.0625	0.002407935	6	461	0.954548	0.000512276	N/A	WORSE	0	-32
pow(x,(-3.0/2.0))*sin(x/2)*exp(-x)	0	INFINITY	0.861179089	4	1	0.861179089	4	155	9.91208E-10	2.86304E-12			0	0
pow(x,(-2.0/7.0))*exp(-((x*x)))	0	INFINITY	1.246631335	4	1	1.246631335	4	143	1.31338E-10	6.44063E-12			0	0
pow(x,29)/pow(5*(x*x)+49,17)	0	INFINITY	2.84E-17	2	1	2.84326E-17	4	99	3.54879E-16	3.53878E-29			0	0
(x*x*x)/(exp(x)-1)	0	INFINITY	6.493939402	12	16	6.493939402	4	103	5.42486E-12	3.52278E-11	IMPROVED	BETTER	70	0
(atan(pi*x)-atan(x))/x	0	INFINITY	1.7981375	4	1	1.7981375	3	69	3.88437E-12	4.06786E-13			0	0
log(x)/((x*x)+4)	0	INFINITY	0.544396523	4	1	0.544396523	3	73	1.29304E-11	1.22125E-15			0	0
((x*x*x*x)-2)*(x*x)/cosh(x*pi/2)	0	INFINITY	59	4	1	59.00000001	5	167	1.85508E-10	1.44992E-08			0	0
log(x)/(1+exp(3*x))	0	INFINITY	-0.333908839	4	1	-0.333908839	4	137	1.97291E-11	6.55176E-12			0	0
x*sin(3*x)/pow(4+(x*x),2)	0	INFINITY	0.002920211	12	0.0625	0.002407935	6	461	0.954548	0.000512276	N/A	WORSE	0	-32
sin(3*x)*sin(2*x)/(x*x)	0	INFINITY	3.141592654	4	1	3.144406214	6	495	0.00324789	0.00281356	N/A		0	0
cos(x)/(1+(x*x))	0	INFINITY	0.577863675	4	1	0.580570089	6	479	0.000480726	0.00270641	N/A		0	0
pow(sin(3*x),3)*sin(12*x)/(x*x*x*x)	0	INFINITY	42.41150082	12	2	42.41153139	6	509	2.53111E-06	3.05736E-05	N/A	WORSE	0	-4
log(((x*x)+4)/((x*x)+1))*cos(x)	0	INFINITY	0.730559018	4	1	0.738676754	6	495	0.00113395	0.00811774	N/A		0	0
cos(3*x)*(1-(x*x))/pow(1+(x*x),2)	0	INFINITY	0.234616032	4	1	0.228083159	6	483	0.0308984	0.00653287	N/A		0	0
pow(sin(3*x),5)/(x*x)	0	INFINITY	1.580999019	12	0.0625	1.552265251	6	393	0.00137713	0.0287338	N/A	BETTER	28	0
sin(3*x)/(1+(x*x))*tanh(pi*x/2)	0	INFINITY	0.174223861	12	0.0625	0.15618316	6	513	0.192739	0.0180407	N/A	WORSE	0	-14
sin(x)/x*sin(x/3)*3/x	0	INFINITY	1.570796327	4	1	1.564126167	6	491	0.00942204	0.00667016	N/A		0	0
pow(cos(x),2)/(x*x)-cos(2*x)/(x*x)	0	INFINITY	1.570796327	4	1	1.572062645	6	483	0.0022929	0.00126632	N/A		0	0
atan(exp(-x))	0	INFINITY	0.915965594	4	1	0.915965594	4	131	9.67506E-12	9.67371E-12			0	0
exp(-x)*log(x)	0	INFINITY	-0.577215665	4	1	-0.577215665	4	139	4.27544E-10	3.47278E-13			0	0
(1/(1+(x*x*x))-exp(-x))/x	0	INFINITY	0.577215665	4	1	0.577215665	4	131	5.08434E-12	1.23175E-11			0	0
log(x)*exp(-((x*x)))	0	INFINITY	-0.870057727	4	1	-0.870057727	4	137	2.53803E-10	1.31026E-11			0	0
exp(-x)*pow(log(x),2)	0	INFINITY	1.978111991	4	1	1.978111991	4	139	5.99897E-10	1.14759E-11			0	0
x*sqrt(1+x)/pow((1+x),13)	0	INFINITY	0.008281573	4	1	0.008281573	3	61	7.89065E-11	1.14492E-16			0	0
pow(sin(3*x),2)/(x*x)	0	INFINITY	4.71238898	4	1	4.706395462	6	495	0.00217365	0.00599352	N/A		0	0
(1-cos(3*x))/(x*x)	0	INFINITY	4.71238898	4	1	4.705843243	6	487	0.0015154	0.00654574	N/A		0	0
exp(-3*x)*cos(4*x)	0	INFINITY	0.12	12	2	0.12	5	263	5.15428E-11	7.09327E-12	IMPROVED	WORSE	0	-4
x/sinh(3*x)	0	INFINITY	0.274155678	4	1	0.274155678	4	131	1.37361E-11	4.10599E-12			0	0
cos(3*x)/cosh(4*x)	0	INFINITY	0.220862409	4	1	0.220862409	4	133	7.929E-10	5.21028E-13			0	0
(exp(-3*x)-exp(-4*x))/x	0	INFINITY	0.287682072	4	1	0.287682072	3	69	4.46903E-09	1.41442E-13			0	0
exp(-4*(x*x))	0	INFINITY	0.443113463	4	1	0.443113463	4	131	2.73334E-11	1.23175E-11			0	0
x/(exp(x)-1)	0	INFINITY	1.644934067	12	4	1.644934067	4	133	8.70126E-13	2.08789E-12	IMPROVED	WORSE	0	-2
x/(exp(x)+1)	0	INFINITY	0.822467033	12	8	0.822467033	3	61	1.2556E-09	3.44169E-15	IMPROVED	BETTER	46	0
log(9+(x*x))/(16+(x*x))	0	INFINITY	1.528314257	4	1	1.528314257	3	73	6.65018E-09	4.21885E-15			0	0
exp(-x)*(x*x*x)	0	INFINITY	6	12	16	6	3	53	1.42702E-10	1.77636E-15	IMPROVED	BETTER	102	0
(cos(3*x)-cos(4*x))/(x*x)	0	INFINITY	1.570796327	4	1	1.576453082	6	487	0.00198135	0.00565675	N/A		0	0
sin(4*x)/x/((x*x)+9)	0	INFINITY	0.174531853	4	1	0.174721495	6	499	0.000237208	0.000189642	N/A		0	0
1/x*(1/(1+x)-exp(-x))	0	INFINITY	0.577215665	4	1	0.577215665	4	131	5.08434E-12	1.23163E-11			0	0
log((x*x)+9)/((x*x)+16)	0	INFINITY	1.528314257	4	1	1.528314257	3	73	6.65018E-09	4.21885E-15			0	0
log(x)/((x*x)+16)	0	INFINITY	0.544396523	4	1	0.544396523	3	73	8.83109E-09	1.44329E-15			0	0
exp(-x)*log(x)	0	INFINITY	-0.577215665	4	1	-0.577215665	4	139	4.27544E-10	3.47278E-13			0	0
exp(-((x*x)))*log(x)	0	INFINITY	-0.870057727	4	1	-0.870057727	4	137	2.53803E-10	1.31026E-11			0	0
exp(-x)*pow(log(x),2)	0	INFINITY	1.978111991	4	1	1.978111991	4	139	5.99897E-10	1.14759E-11			0	0
exp(-3*(x*x)-4/(x*x))	0	INFINITY	5.01E-04	2	1	0.000501307	4	43	6.54092E-10	0			0	0
exp(-3*(x*x))*cos(8*x)	0	INFINITY	0.002470285	12	4	0.002470285	5	271	2.57111E-11	6.43066E-14	IMPROVED	WORSE	0	-4
exp(-3*x)/sqrt(x)/(x+2)	0	INFINITY	0.47677985	4	1	0.47677985	3	81	7.33245E-10	2.45914E-14			0	0
exp(-3*x)*sin(2*x)	0	INFINITY	0.153846154	4	1	0.153846154	4	111	1.65923E-09	2.19105E-12			0	0
x*exp(-3*x)*sin(2*x)	0	INFINITY	0.071005917	4	1	0.071005917	4	99	2.43128E-09	6.74849E-13			0	0
exp(-x)*pow(x,5)	0	INFINITY	120	12	32	120	4	91	3.54561E-13	4.25473E-11		BETTER	50	0
2*x/(1+(x*x))/(exp(2*pi*x)-1)	0	INFINITY	0.077215665	4	1	0.077215665	3	69	1.18213E-10	4.56718E-14			0	0
exp(-2*x)*tanh(x/2)/x/cosh(x)	0	INFINITY	0.21001823	4	1	0.21001823	4	133	9.16515E-13	2.6007E-13			0	0
x*exp(-2*(x*x))	0	INFINITY	0.25	4	1	0.25	4	111	5.99657E-11	1.09521E-12			0	0
exp(-x)/(1+x)/sqrt(x)	0	INFINITY	1.343293415	4	1	1.343293422	4	155	6.14085E-12	6.712E-09			0	0
exp(-x)*pow(x,(7.0/2.0))	0	INFINITY	11.6317284	12	16	11.6317284	4	95	7.23571E-13	6.29166E-10	IMPROVED	BETTER	48	0
sech(2*x)*cos(4*x)	0	INFINITY	0.067753738	12	2	0.067753738	5	261	1.36418E-09	7.09351E-12	IMPROVED	BETTER	248	0
sin(x)/(x*x)	3.141592654	INFINITY	-0.073667912	12	0.00195312	-0.077601046	6	513	0.205379	0.00393313	N/A	WORSE	0	-16
exp(-x/2)*sin(x)/x	0	INFINITY	1.107148718	4	1	1.107148717	6	491	4.75821E-10	5.97262E-10			0	0
x/pow(1+(x*x),3)*sin(x/2)	0	INFINITY	0.089319012	4	1	0.089319012	6	359	4.25565E-08	1.79972E-10	N/A		0	0
x*exp(-x)/(1+(x*x))	0	INFINITY	0.343377962	4	1	0.343377962	4	111	3.01509E-11	9.25315E-12			0	0
exp(-x)*(x*x*x)	0	INFINITY	6	12	16	6	3	53	1.42702E-10	1.77636E-15	IMPROVED	BETTER	102	0
exp(-x)*(x*x*x)	1	INFINITY	5.886071059	12	8	5.886071059	3	67	3.54808E-09	4.2899E-13	IMPROVED	BETTER	58	0
pow(x,7)*exp(-x)	0	INFINITY	5040	12	64	5040	4	87	6.01097E-12	4.18186E-09		BETTER	54	0
exp(-3*x)*sin(4*x)	0	INFINITY	0.16	4	1	0.16	5	219	6.02376E-11	5.96972E-12			0	0
exp(-x/2)/x*sin(x)	0	INFINITY	1.107148718	4	1	1.107148717	6	491	4.75821E-10	5.97262E-10			0	0
sin(x)/(x*x)	3.141592654	INFINITY	-0.073667912	12	0.00195312	-0.077601046	6	513	0.205379	0.00393313	N/A	WORSE	0	-16
x/(1+pow(x,6)*pow(sinh(x),2))	0	INFINITY	0.503686664	4	1	0.503686664	5	209	6.11576E-11	3.20912E-11			0	0
log(1+9*(x*x))/(1+16*(x*x))	0	INFINITY	0.439521212	12	0.5	0.439521212	3	71	9.28425E-13	3.88578E-16			0	0
(x*x*x)*exp(-x)	0	INFINITY	6	12	16	6	3	53	1.42702E-10	1.77636E-15	IMPROVED	BETTER	102	0
(exp(-x)-exp(-3*x))/x	0	INFINITY	1.098612289	4	1	1.098612289	4	131	3.05709E-11	2.46354E-11			0	0
sin(x)*exp(-x)	0	INFINITY	0.5	12	0.5	0.5	5	205	2.08955E-09	3.47373E-11		BETTER	4	0
1/cosh(x)	0	INFINITY	1.570796327	12	2	1.570796327	3	69	2.05996E-09	3.77209E-12	IMPROVED	BETTER	62	0
1/cosh(3*x)	0	INFINITY	0.523598776	4	1	0.523598776	4	131	2.32197E-11	1.23176E-11			0	0
tanh(x/2)/cosh(x)	0	INFINITY	0.693147181	12	4	0.693147181	3	59	9.13855E-09	6.99441E-15	IMPROVED	BETTER	48	0
1/pow(cosh(x),2)	0	INFINITY	1	4	1	1	3	69	9.17328E-09	3.61888E-12			0	0
(x*x*x)/sinh(x)	0	INFINITY	12.17613638	12	16	12.17613638	4	103	2.88874E-12	3.52252E-11	IMPROVED	BETTER	68	0
(x*x)/cosh(x)	0	INFINITY	3.875784585	12	16	3.875784585	4	105	6.41295E-12	3.52403E-11	IMPROVED	BETTER	68	0
(x*x)*sinh(x)/pow(cosh(x),2)	0	INFINITY	3.663862377	12	16	3.663862377	4	99	1.64621E-11	1.76348E-12	IMPROVED	BETTER	58	0
1/(1+(x*x))/cosh(pi*x)	0	INFINITY	0.429203673	4	1	0.429203673	3	69	9.11384E-10	1.42664E-13			0	0
exp(-2*x)*(cos(x)-cos(2*x))/x	0	INFINITY	0.235001815	4	1	0.235001815	5	213	4.20484E-11	1.15456E-11			0	0
exp(-x)*cos(x)	0	INFINITY	0.5	4	1	0.5	5	253	6.47433E-10	7.54214E-11			0	0
cos(x)/((x*x)+1)	0	INFINITY	0.577863675	4	1	0.580570089	6	479	0.000480726	0.00270641	N/A		0	0
log(x)/(1+pow((10*x),2))	0	INFINITY	-0.361689221	4	1	-0.361689221	4	139	2.60022E-12	3.43614E-13			0	0
cos(x)/(pow((x-2),2)+1)	0	INFINITY	-0.561273903	4	1	-0.558444846	6	493	0.00051987	0.00282906	N/A		0	0
log(x+1)/((x*x)+1)	-1	INFINITY	0.816594784	4	1	0.816594784	4	133	8.02204E-12	6.5512E-12			0	0
sin(x)*log(x)*exp(-((x*x*x)))	0	INFINITY	-0.195788516	4	1	-0.195788516	5	219	1.00406E-10	2.01501E-11			0	0
exp(-(sqrt((x*x)-1)*11.0/1000.0))*x/sqrt((x*x)-1)	1	INFINITY	90.90909091	12	512	90.9090909	4	143	6.36075E-11	9.93744E-09	IMPROVED	BETTER	130	0
log((x*x*x)+1)/((x*x)+1)	-1	INFINITY	2.737773172	4	1	2.737773172	4	133	2.37218E-11	3.29976E-11			0	0
			average=	5.95			average=	179.38		improved=	28	count=	29	14
												points=	1834	-158
												average=	63.2	-11.3

Appendix D

Romberg quadrature

Note: condition `i>2` prevents premature termination, which can be adjusted as needed. The relative error is tested in the convergence check, where the bound `eps*fabs(Ru[i])+eps` is increased by `+eps` to compare the absolute error to `eps` if the integral sum `Ru[i]` is close to zero.

```
double qromb(double (*f)(double), double a, double b, int n, double eps) {
  double R1[n], R2[n];
  double *Ro = &R1[0], *Ru = &R2[0];
  double h = b-a;
  int i, j;
  Ro[0] = (f(a)+f(b))*h/2;
  for (i = 1; i < n; ++i) {
    unsigned long long k = 1UL << i;
    unsigned long long s = 1;
    double sum = 0;
    double *Rt;
    h /= 2;
    for (j = 1; j < k; j += 2)
      sum += f(a+j*h);
    Ru[0] = h*sum + Ro[0]/2;
    for (j = 1; j <= i; ++j) {
      s <<= 2;
      Ru[j] = (s*Ru[j-1] - Ro[j-1])/(s-1);
    }
    if (i > 2 && fabs(Ro[i-1]-Ru[i]) <= eps*fabs(Ru[i])+eps)
      return Ru[i];
    Rt = Ro;
    Ro = Ru;
    Ru = Rt;
  }
  return Ro[n-1];
}
```

Adaptive Simpson quadrature

```
double qasi(double (*f)(double), double a, double b, int n, double eps) {
  double fa = f(a);
  double fm = f((a+b)/2);
  double fb = f(b);
  double v  = (fa+4*fm+fb)*(b-a)/6;
  return as(f, a, b, fa, fm, fb, v, eps, n, 0);
}
double as(double (*f)(double), double a, double b, double fa, double fm,
          double fb, double v, double eps, int n, double t) {
  double h  = (b-a)/2;
  double f1 = f(a + h/2);
  double f2 = f(b - h/2);
  double sl = h*(fa + 4*f1 + fm)/6;
  double sr = h*(fm + 4*f2 + fb)/6;
  double s  = sl+sr;
  double d  = (s-v)/15;
  double m  = a+h;
```

```
  if (n <= 0 || fabs(d) < eps)
    return t + s + d; // note: fabs(d) can be used as error estimate
  eps /= 2;
  --n;
  t = as(f, a, m, fa, f1, fm, sl, eps, n, t);
  return as(f, m, b, fm, f2, fb, sr, eps, n, t);
}
```

The *Adaptive Gauss-Kronrod (G10,K21)* quadrature method (which also returns `err`, the absolute difference between the *Gauss* and *Gauss-Kronrod* approximations).

```
double qakro(double (*f)(double), double a, double b, int n, double tol,
             double eps, double *err) {
  double c = (a+b)/2;
  double d = (b-a)/2;
  double e;
  double r = gk(f, c, d, &e);
  double s = d*r;
  double t = fabs(s*tol);
  if (tol == 0)
    tol = t;
  if (n > 0 && t < e && tol < e) {
    s = qakro(f, a, c, n-1, t/2, eps, err);
    s += qakro(f, c, b, n-1, t/2, eps, &e);
    *err += e;
    return s;
  }
  *err = e;
  return s;
}
double gk(double (*f)(double), double c, double d, double *err) {
  // abscissas and weights pre-calculated with Legendre Stieltjes polynomials
  static const double abscissas[21] = {
    0.00000000000000000e+00,
    7.65265211334973338e-02,
    1.52605465240922676e-01,
    2.27785851141645078e-01,
    3.01627868114913004e-01,
    3.73706088715419561e-01,
    4.43593175238725103e-01,
    5.10867001950827098e-01,
    5.75140446819710315e-01,
    6.36053680726515025e-01,
    6.93237656334751385e-01,
    7.46331906460150793e-01,
    7.95041428837551198e-01,
    8.39116971822218823e-01,
    8.78276811252281976e-01,
    9.12234428251325906e-01,
    9.40822633831754754e-01,
    9.63971927277913791e-01,
    9.81507877450250259e-01,
    9.93128599185094925e-01,
    9.98859031588277664e-01,
  };
  static const double weights[21] = {
    7.66007119179996564e-02,
```

```
    7.63778676720807367e-02,
    7.57044976845566747e-02,
    7.45828754004991890e-02,
    7.30306903327866675e-02,
    7.10544235534440683e-02,
    6.86486729285216193e-02,
    6.58345971336184221e-02,
    6.26532375547811680e-02,
    5.91114008806395724e-02,
    5.51951053482859947e-02,
    5.09445739237286919e-02,
    4.64348218674976747e-02,
    4.16688733279736863e-02,
    3.66001697582007980e-02,
    3.12873067770327990e-02,
    2.58821336049511588e-02,
    2.03883734612665236e-02,
    1.46261692569712530e-02,
    8.60026985564294220e-03,
    3.07358371852053150e-03,
  };
  static const double gauss_weights[10] = {
    1.52753387130725851e-01,
    1.49172986472603747e-01,
    1.42096109318382051e-01,
    1.31688638449176627e-01,
    1.18194531961518417e-01,
    1.01930119817240435e-01,
    8.32767415767047487e-02,
    6.26720483341090636e-02,
    4.06014298003869413e-02,
    1.76140071391521183e-02,
  };
  double p = 0; // kronrod quadrature sum
  double q = 0; // gauss quadrature sum
  double fp, fm;
  double e;
  int i;
  fp = f(c);
  p = fp * weights[0];
  for (i = 1; i < 21; i += 2) {
    fp = f(c + d * abscissas[i]);
    fm = f(c – d * abscissas[i]);
    p += (fp + fm) * weights[i];
    q += (fp + fm) * gauss_weights[i/2];
  }
  for (i = 2; i < 21; i += 2) {
    fp = f(c + d * abscissas[i]);
    fm = f(c – d * abscissas[i]);
    p += (fp + fm) * weights[i];
  }
  *err = fabs(p - q);
  e = fabs(2*p*1e-17); // optional, to take 1e-17 MachEps prec. into account
  if (*err < e)
    *err = e;
  return p;
}
```