Accurate Calculation of $\exp(\pm x^2)$

J.M. Arnold (jearnold@cern.ch)

November 10, 2020

Background

Two new functions are proposed — `expxsqr(x)` and `expmxsqr(x)` — which compute Binary64 (i.e., `double`) approximations to e^{x^2} and e^{-x^2} respectively.

The values of e^{x^2} and e^{-x^2} are often used in the evaluation of functions such as erfc, erfcx and other related functions. For example, `exp(x*x)` and `exp(-x*x)` appear multiple times in the code of the Faddeeva Package [5]. Using standard library implementations of `exp` to calculate `exp(x*x)` and `exp(-x*x)` can be problematic: the absolute error can exceed 500 ulps for large arguments. This is not caused by any deficiency in the implementation of the `exp` function but rather because of the approximation error in using `x*x` to represent x^2. Splitting x^2 into high and low parts (e.g., $x^2 \to t_h + t_\ell$) and computing `exp(t_h)*exp(t_l)` produces results with much smaller absolute errors but the new proposed functions are more accurate and (probably) have better performance.

Implementation

The primary goal of these implementations of `expxsqr(x)` and `expmxsqr(x)` is to produce correctly rounded results; this goal is more important than performance. Every attempt has been made, however, to maximize performance as long as accuracy is maintained. Even though these implementations attempt to produce correctly rounded results, not all results are, in fact, correctly rounded. For the vast majority (more than 99%) of arguments tested, however, the results are correctly rounded. Extensive testing has failed to find any case for which the result is not at least faithfully rounded (i.e., no cases have been found for which the absolute error is greater than 1 ulp).

While the implementations are somewhat similar to that used internally in [1] in that they use `doubledouble` arithmetic to achieve the desired accuracy, these routines differ in that they are designed to be available for general use. They also assume that an IEEE-754-compliant `FMA` operation is available. This allows for the use of `doubledouble` routines [6, 8] and argument reduction techniques [2, 7, 9] with improved accuracy and performance.

The routines also make assumptions about the in-memory storage format of Binary64 floating-point values.

None of these assumptions are absolutely required but their invalidity would require rewriting some of the code, and performance would be adversely affected.

The `sollya` tool [3] was used to generate the various numeric constants and polynomial coefficients.

These new routines do not set `errno` or attempt to correctly set any floating-point exception flags although, if the argument is a signaling NaN, the floating-point exception `FE_INVALID` will be set. The routines do not cause gratuitous underflows or overflows.

Results of Accuracy Testing

Reference implementations of `expxsqr(x)` and `expmxsqr(x)` were created with `MPFR` [4]. A precision of `8*DBL_MANT_DIG` (424 bits) is used.

The tests were conducted on `lxplus.cern.ch` (CentOS 7) and Mac OS X 10.1 5 (Catalina).

libm

Figures 1 and 2 show the errors associated with computing $e^{\pm x^2}$ using `exp(x*x)` and `exp(-x*x)`. For large arguments, the error can be significant. Again, this is not because of any error in the `exp` routine itself but rather because of the roundoff error in approximating x^2 by `x*x`.

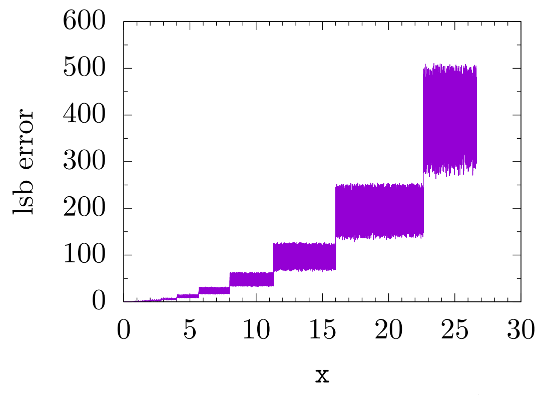

Figure 1: Accuracy of `exp(x*x)`

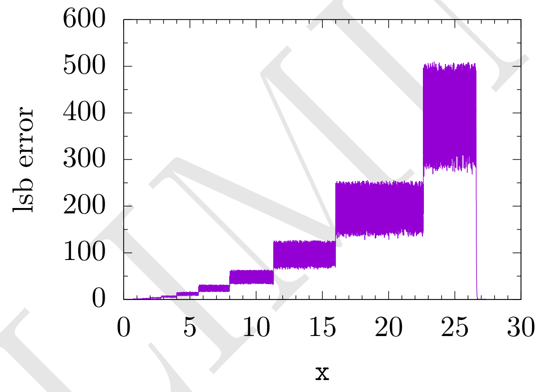

Figure 2: Accuracy of `exp(-x*x)`

`expxsqr` and `expmxsqr`

The new routines produce results which are always at least faithfully rounded. See tables 1 and 2 and associated error plots. The range of argument values tested correspond to results with

$$1.0 < \texttt{expxsqr}(\texttt{x}) < \infty$$

and

$$0.0 < \texttt{expmxsqr}(\texttt{x}) < 1.0$$

.

expxsqr(x)	
x range	1.05×10^{-8} to 26.64
number of points	10^7
correctly rounded	99.73%
faithfully rounded	0.27%
worst-case error	0.524 ulp

Table 1: Accuracy of `expxsqr(x)`

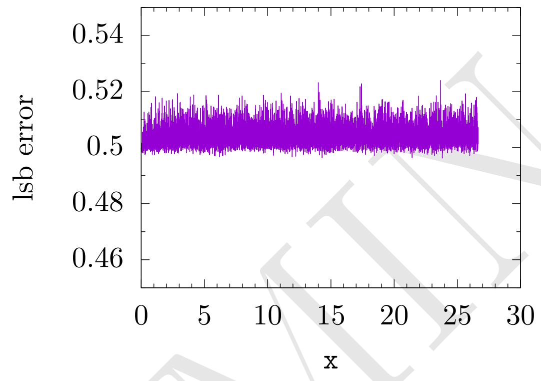

Figure 3: Accuracy of `expxsqr(x)`

expmxsqr(x)	
x range	7.45×10^{-9} to 27.30
number of points	10^7
correctly rounded	99.71%
faithfully rounded	0.29%
worst-case error	0.752 ulp

Table 2: Accuracy of `expmxsqr(x)`

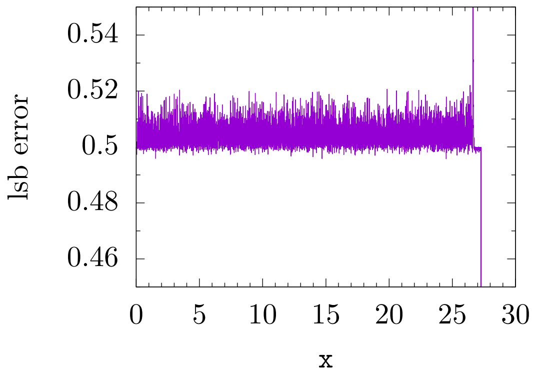

Figure 4: Accuracy of `expmxsqr(x)`

References

[1] N. Baikov, “Algorithm and Implementation Details for Complementary Error Function,” IEEE Trans. Comput., vol. 66, no. 7, pp. 1106–1118, Jul. 2017, doi: 10.1109/TC.2016.2641960.

[2] S. Boldo, M. Daumas, and R.-C. Li, “Formally Verified Argument Reduction with a Fused Multiply-Add,” IEEE Transactions on Computers, vol. 58, no. 8, pp. 1139–1145, 2009, doi: 10.1109/TC.2008.216.

[3] S. Chevillard, M. Joldeş, and C. Lauter, “Sollya: An Environment for the Development of Numerical Codes,” in Mathematical Software – ICMS 2010, vol. 6327, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 28–31.

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, “MPFR: A multiple-precision binary floating-point library with correct rounding,” ACM Trans. Math. Softw., vol. 33, no. 2, p. 13, Jun. 2007, doi: 10.1145/1236463.1236468.

[5] Steven G. Johnson, Faddeeva Package, Massachusetts Institute of Technology, Boston, USA, `http://ab-initio.mit.edu/wiki/index.php/Faddeeva_Package`

[6] M. M. Joldes, J.-M. Muller, and V. Popescu, “Tight and rigourous error bounds for basic building blocks of double-word arithmetic,” ACM Transactions on Mathematical Software, vol. 44, no. 2, pp. 1–27, 2017, doi: 10.1145/3121432.

[7] J. M. Muller, Elementary functions: algorithms and implementation, Third edition. Boston: Birkhäuser, 2016.

[8] J.-M. Muller and L. Rideau, Formalization of double-word arithmetic, and comments on ”Tight and rigorous error bounds for basic building blocks of double-word arithmetic”, Oct. 2020. `https://hal.archives-ouvertes.fr/hal-02972245`

[9] P.-T. P. Tang, “Table-driven implementation of the exponential function in IEEE floating-point arithmetic,” ACM Trans. Math. Softw., vol. 15, no. 2, pp. 144–157, Jun. 1989, doi: 10.1145/63522.214389.

[10] P. T. P. Tang, “Table-driven implementation of the Expm1 function in IEEE floating-point arithmetic,” ACM Trans. Math. Softw., vol. 18, no. 2, pp. 211–222, Jun. 1992, doi: 10.1145/146847.146928.

Appendix

The general flow of the routines is

- Screen the argument x for exceptional values:
  - the argument is a NaN
  - the result is known without performing any computation (e.g., 0.0, 1.0 or $+\infty$)
- Accurately compute $x^2 \to t_h + t_\ell$ in `doubledouble`
- From t_h, calculate the `doubledouble` reduced argument $r = r_h + r_\ell$ and scale factors m and j such that $t_h = (32m + j)(\log(2)/32) + r$
- Calculate $2^{\pm j/32} \mathrm{e}^{\pm r_h} \mathrm{e}^{\pm(r_\ell + t_\ell)}$
- Scale by $2^{\pm m}$

Breakpoints

`expxsqr(x)` is

- 1.0 for $|x| < 1.0536712127723507 \times 10^{-8}$ $(1592262918131443 \times 2^{-77})$
- ∞ for $|x| > 26.641747557046326$ $(7498985273150791 \times 2^{-48})$

`expmxsqr(x)` is

- 1.0 for $|x| < 7.4505805969238298 \times 10^{-9}$ $(4503599627370497 \times 2^{-79})$
- subnormal for $|x| > 26.615717509251262$ $(3745829233026949 \times 2^{-47})$
- 0.0 for $|x| > 27.297128403953796$ $(7683458581770681 \times 2^{-48})$