NOTE Communicated by Michael Hines

A Fast, Compact Approximation of the Exponential Function

Nicol N. Schraudolph
IDSIA, Lugano, Switzerland

Neural network simulations often spend a large proportion of their time computing exponential functions. Since the exponentiation routines of typical math libraries are rather slow, their replacement with a fast approximation can greatly reduce the overall computation time. This article describes how exponentiation can be approximated by manipulating the components of a standard (IEEE-754) floating-point representation. This models the exponential function as well as a lookup table with linear interpolation, but is significantly faster and more compact.

1 Motivation

Exponentiation is arguably the quintessential nonlinear function of neural computation. Among other uses, it is needed to compute most of the activation functions and probability distributions used in neural network models. Consequently, much of the time in neural simulations is actually spent on exponentiation.

The `exp` functions provided by typical computer math libraries are highly accurate but rather slow. An approximation is perfectly adequate for most neural computation purposes and can save much time. In recognition of this, many neural network software packages approximate `exp` with a lookup table, typically with linear interpolation. There is, however, an even faster and highly compact way to obtain comparable approximation quality.

2 The Algorithm

Floating-point numbers are typically represented on computers in the form $(-1)^s\,(1+m)\,2^{x-x_0}$, where s is the sign bit, m the mantissa—a binary fraction in the range $[0, 1)$—and x the exponent, shifted by a constant bias x_0. The widely used IEEE-754 standard (IEEE, 1985) specifies a 52-bit mantissa and an 11-bit exponent with bias $x_0 = 1023$, laid out in 8 bytes of computer memory, as shown in Figure 1 (top row). The components of this representation may be manipulated by accessing the same memory location as a pair of 4-byte integers (denoted i and j here). In particular, any integer written directly to the x component (via i) will be exponentiated when the same memory location is read back in floating-point format. This is the key idea behind the fast exponentiation macro proposed here.

sxxxxxxx	xxxxmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm	mmmmmmmm
1	2	3	4	5	6	7	8
iiiiiiii	iiiiiiii	iiiiiiii	iiiiiiii	jjjjjjjj	jjjjjjjj	jjjjjjjj	jjjjjjjj

Figure 1: Bit representation of the `union` data structure used by the `EXP` macro. The same 8 bytes can be accessed either as an IEEE-754 `double` (top row) with sign s, exponent x, and mantissa m, or as two 4-byte integers i and j (bottom).

Since the x component resides in the higher-order bits of i, an integer y to be exponentiated must be left-shifted by 20 bits, after the bias x_0 has been added. Thus $i := 2^{20}(y+1023)$ computes 2^y for integer y. Now consider what happens for noninteger arguments: After multiplication, the fractional part of y will spill over into the highest-order bits of the mantissa m. This spillover is not only harmless, but in fact is highly desirable—under the IEEE-754 format, it amounts to a linear interpolation between neighboring integer exponents. The technique therefore exponentiates real-valued arguments as well as a lookup table with 2^{11} entries and linear interpolation.

Finally, to compute e^y rather than 2^y, y must be divided by ln(2) first. The complete transformation of y necessary to compute a fast approximation to e^y in the IEEE-754 format is given by

$$i \;:=\; a\,y + (b - c) \tag{2.1}$$

where $a = 2^{20} / \ln(2)$, $b = 1023 \cdot 2^{20}$, and c is an adjustment parameter that affords some control over the properties of the approximation (see section 4).

Figure 2 shows C code implementing this method. The `LITTLE_ENDIAN` flag is necessary since computers differ in how they store multibyte quantities in memory. The simplest way to determine whether it should be set on a given machine is to try both alternatives. The `union` data structure should be declared `static` to ensure that j (which is never used by the macro) is initialized to zero, as well as to avoid name clashes when this code is included in multiple source modules, for example, from a common header file.

For integer arguments y, a significant additional speedup (see Table 1) can be obtained at little cost in accuracy by setting `EXP_A` and `EXP_C` to integer values, so that the `EXP` macro need not perform any floating-point arithmetic at all. This trick can be used in conjunction with noninteger quantizations as well: By premultiplying `EXP_A` with the (real-valued) quantum q, then rounding to integer, one obtains a macro that approximates e^{yq} for integer y, using only integer arithmetic. However, in our experience, casting inherently real-valued arguments to integer in order to exploit this feature is generally *not* a good idea, since type conversion from floating point to integer tends to be a comparatively expensive operation.

```
#include <math.h>

static union
[
    double d;
    struct
    [
#ifdef LITTLE˙ENDIAN
        int j, i;
#else
        int i, j;
#endif
    ] n;
] ˙eco;

#define EXP˙A (1048576/M˙LN2)  /* use 1512775 for integer version */
#define EXP˙C  60801           /* see text for choice of c values */

#define EXP(y) (˙eco.n.i = EXP˙A*(y) + (1072693248 - EXP˙C), ˙eco.d)
```

Figure 2: C code implementing the `union` data structure and `EXP` macro for fast approximate exponentiation. `LITTLE_ENDIAN` must be defined for machines that store integers with the least significant byte first; `EXP_C` is set to the desired value of the *c* parameter (see section 4).

Table 1: Seconds Required for 10^8 Exponentiations on a Variety of Workstations.

Manufacturer	Intel	SGI	Sun	DEC
Processor	Pentium	MIPS	UltraSparc	Alpha server
Model/Speed	Pro/240	4600SG	1/170	2100A/300
`LITTLE_ENDIAN`	Yes	No	No	Yes
Op. System	Linux 2.0.29	Irix 5.3	SunOS 5.5.1	OSF1 4.0
Compiler	gcc 2.7.2.1	`/bin/cc`	gcc 2.7.2.1	DEC C 5.2
Optimization	-O2	-O4	-O2	-fast
`exp` (`libm.a`)	89	126	166	28
Lookup table	46	62	22	23
`EXP` macro	28	25	7.6	4.2
`EXP` (integers)	6.2	6.8	3.7	-0.6

3 Benchmark Results

Table 1 lists the benchmark results obtained on a variety of machines for the standard math library's `exp` function, a lookup table with linear interpolation, and the `EXP` macro in its general (floating-point) and integer forms. The benchmark program was required to return the sum of 10^8 exponentials of pseudorandom arguments so as to prevent "optimizing away" of any exponentiation by the compiler. On each machine, the time taken to calculate just the sum of the 10^8 pseudorandom arguments was subtracted to obtain net computing times for the exponentiation. To check for variation in the CPU time consumed, each benchmark was run three times. The figures shown in Table 1 are averages over these three runs; the observed fluctuations were very small.

The results show that the `EXP` macro is clearly the fastest on all machines tested. For floating-point arguments it requires between 18% (DEC Alpha) and 60% (Intel Pentium Pro) of the time needed by the lookup table. Not surprisingly, the standard math library's `exp` routines follow far behind. The `-fast` optimization switch on the DEC Alpha activates an approximate `exp` routine that is only slightly slower than a lookup table, but the other machines do not have such a feature. Performance on the Sun workstation in particular suffers from an `exp` function that is almost 22 times slower than the `EXP` macro.

This discrepancy grows to an impressive 45-fold speed advantage for the integer form of the macro. The integer variant is significantly faster than the general (floating-point) form of `EXP` on all tested machines. On the DEC Alpha, it appears to be even faster than light, taking negative time! Recall, though, that these figures denote net computing times, from which the time taken by a control—the same program with the exponentiation removed—has been subtracted. In this case, the integer `EXP` macro was on average 6 nanoseconds faster than the integer to floating-point type conversion that takes place instead in the control program. Although not as impressive as violating basic laws of physics, this still testifies to a rather astonishing speed.

In summary, these benchmark results indicate that the `EXP` macro could greatly accelerate computations that make heavy use of exponentiation.[1] It is both faster and more compact than a lookup table with linear interpolation, a widely used acceleration method. Finally, its speed is even greater for integer arguments, as occur, for example, in the calculation of the Boltzmann-Gibbs distribution for quantized energy levels.

4 Approximation Properties

Computing `EXP`(y) is very fast, but how well does it approximate e^y? Figure 3 shows the logistic function implemented using the `EXP` macro versus the standard math library's `exp` function. The left panel illustrates that on a global scale, the two are all but indistinguishable. The greater magnification in the center panel highlights the linear interpolation performed by `EXP`

[1] To give an example, Lazzaro and Wawrzynek's (1999) neural network-based JPEG quality transcoder runs twice as fast when using the `EXP` macro (Lazzaro, personal communication).

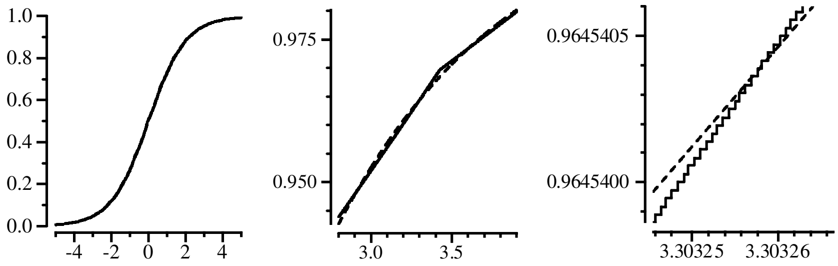

Figure 3: Comparison of the logistic function $y \mapsto (1 + e^{-y})^{-1}$ implemented using the `EXP` macro (solid line, for c = 60,801) versus the math library's `exp` function (dashed line). Different scales highlight the global fit (left), the linear interpolation (center), and the staircase effect (right).

due to the limited precision of the 11-bit exponent x. Finally, the highly magnified right-hand panel of Figure 3 shows that on the very small scale of $\Delta y = 2^{-20}$, `EXP`(y) exhibits a staircase structure. This happens because the macro completely ignores the lower part j of the mantissa, leaving it at zero—the value to which static variables in C are initialized—for reasons of efficiency.

Versions of `EXP` that use 8-byte (`long long`) integers do not suffer from this staircase effect, but were found to be unacceptably slow on the typical workstation platforms. As it stands, `EXP`(y) is thus monotonically nondecreasing but (unlike e^y) not monotonically increasing. Although this should be kept in mind when writing code that uses the `EXP` macro, in practice it should not present any difficulties.

The c parameter in equation 2.1 permits some fine-tuning of the approximation for certain desirable characteristics. For $c = 0$, the `EXP` macro interpolates between 2^{11} points that lie exactly on the exponential function: `EXP`$(n \ln 2) = e^{n \ln 2} = 2^n$ for all integer n. Due to the staircase effect, however, an upper bound on the exponential ($\forall y$ `EXP`$(y) \geq e^y$) requires $c \leq -1$. Positive values of c right-shift `EXP`(y); a lower bound on e^y is returned for $c \geq$ 90,253. (Mathematical derivations for these values are presented in the appendix.) If tight bounds are required on both sides, a particularly efficient way to compute them for a given argument is to call the macro

```
#define EXP`L (`eco.n.i -= 90254, `eco.d),
```

which returns the lower bound, right after computing the upper bound by a call to `EXP` (with `EXP_C` set to –1). Intermediate values of c produce the best overall approximations: the maximum relative error (to either side of e^y) is smallest for $c \approx$ 45,799, the minimum root-mean-square (RMS) relative error is reached at $c \approx$ 60,801, and the mean relative error is lowest at $c \approx$ 68,243.(See the appendix.)

Table 2: Relative Error of the EXP Macro for Various Choices of the *c* Parameter.

Relative Error: $\gamma \equiv c{\cdot}\ln(2)/2^{20}$	Max. $< e^y$: $(1 - e^{-\gamma})$		Max. $> e^y$: $(2\,e^{-(\gamma+1)}/\ln(2){-}1)$	Root Mean Square: $(\sqrt{\Psi(\gamma)})$	Mean: $(\Phi(\gamma))$
c = −1	*0.000* %		6.148 %	4.466 %	4.069 %
c = 45,799	*2.982*	=	*2.982*	2.031	1.811
c = 60,801	3.939		1.966	*1.770*	1.522
c = 68,243	4.411		1.466	1.837	*1.483*
c = 90,253	5.792		*0.000*	2.617	1.959

Table 2 lists the maximum (below and above e^y), RMS, and mean relative error of `EXP` for each of the above settings of c, with optimal error values italicized. These values have been measured empirically; they are in perfect agreement with the analytically derived formulas shown in the column headings, which stem from equations A.7, A.8, and A.12.

5 Limitations

The `EXP` macro proposed here provides a very fast, reasonably accurate approximation of the exponential function. Nevertheless, its speed is bought at a price:

- It requires 4-byte integers and IEEE-754-compliant floating-point data types. (These are available in most computing environments.)

- Its implementation depends on the byte order of the machine.

- Its use of a global static variable is problematic in multithreaded environments. (Each thread must have a private copy of the `_eco` data structure.)

- There is no overflow or error handling. The user must ensure that the argument is in the valid range (roughly, −700 to 700).

- It only approximates the exponential function (see section 4). Certain numerical methods may amplify the approximation error; each algorithm to use `EXP` should therefore be tested against the original version first.

In situations where these limitations are acceptable, the `EXP` macro promises to speed up the computation of exponentials greatly.

Appendix: Mathematical Analysis

Ignoring the staircase effect shown in Figure 3 (right), the `EXP` macro can be described as

$$\mathrm{EXP}(y+\gamma\,) \;=\; 2^k(1+y/\ln(2)-k)\,, \quad \text{where} \quad k \;\equiv\; \lfloor\, y/\ln(2)\rfloor\,, \qquad \text{(A.1)}$$

where $\gamma\;\equiv c\cdot\ln(2)/2^{20}$, and $\lfloor u\rfloor$ denotes the largest integer $\leq u$. In what follows, various values of c are derived for which equation A.1 has certain desirable properties.

A.1 Upper and Lower Bound. The exponential inequality states that:

$$\begin{aligned} 2^{\alpha} &\leq 1+\alpha \qquad \forall\,\alpha\in[0,1] \\ 2^{y/\ln(2)-k} &\leq 1+y/\ln(2)-k \\ e^{y} &\leq \mathrm{EXP}(y+\gamma\,). \end{aligned} \qquad \text{(A.2)}$$

For $\gamma\;\leq 0$ this implies $e^y\leq\mathrm{EXP}(y)$. The corresponding bound on c must be decremented by 1 on account of the staircase effect; the `EXP` macro hence returns an upper bound to the exponential function for $c\leq -1$.

To determine the smallest value of c for which $\mathrm{EXP}(y)$ delivers a lower bound to e^y, match the two functions' first derivatives:

$$\begin{aligned} \frac{\partial}{\partial\,y}\,e^{y+\gamma} &= \frac{\partial}{\partial\,y}\,\mathrm{EXP}(y+\gamma\,) \\ e^{y+\gamma} &= 2^k/\ln(2) \\ y+\gamma &= k\ln(2)-\ln(\ln(2)) \\ y/\ln(2)-k &= -\,[\,\ln(\ln(2))+\gamma\;]/\ln(2). \end{aligned} \qquad \text{(A.3)}$$

Then compare function values at the points characterized by equation A.3:

$$\begin{aligned} e^{y+\gamma} &\geq \mathrm{EXP}(y+\gamma\,) \\ 2^k/\ln(2) &\geq 2^k\,(1+y/\ln(2)-k) \\ 1 &\geq \ln(2)-[\,\ln(\ln(2))+\gamma\;] \\ c &\geq 2^{20}\,[\,1-[\,\ln(\ln(2))+1]/\ln(2)]\approx 90{,}252.34. \end{aligned} \qquad \text{(A.4)}$$

Rounding up to preserve the bound yields the best integer value of c = 90,253.

A.2 Lowest Maximum Relative Error.. For intermediate values of c, EXP dips both above and below the exponential function. The relative error is greatest at the extrema of

$$r_{\gamma}\,(y)\equiv 1-\mathrm{EXP}(y+\gamma\,)/e^{y+\gamma}\,. \qquad \text{(A.5)}$$

Setting its derivative to zero,

$$\begin{aligned} \frac{\partial}{\partial y} r_\gamma(y) &= \frac{2^k(1 + y/\ln(2) - k) - 2^k/\ln(2)}{e^{y+\gamma}} = 0 \\ y &= (k-1)\ln(2) + 1, \end{aligned} \tag{A.6}$$

yields the local minima of $r_\gamma(y)$. The local maxima can be found at the points where EXP is not differentiable, that is, at $y = k\ln(2)$. The maximum relative error is lowest when the magnitude of $r_\gamma(y)$ is equal at both sets of extrema:

$$\begin{aligned} |r_\gamma[k\ln(2)]| &= |r_\gamma[(k-1)\ln(2) + 1]| \\ 1 - e^{-\gamma} &= 2e^{-(\gamma+1)}/\ln(2) - 1 \\ \gamma &= \ln(\ln(2) + 2/e) - \ln(2) - \ln(\ln(2)) \\ c &= \gamma \cdot 2^{20}/\ln(2) \approx 45{,}799.12 \end{aligned} \tag{A.7}$$

The staircase effect can be adjusted for by subtracting 0.5 from this value; the best integer choice is $c = 45{,}799$.

A.3 Lowest RMS Relative Error. To compute the value of c that minimizes the RMS relative error, consider the integrated squared relative error Ψ:

$$\begin{aligned} \Psi(\gamma) &\equiv \frac{1}{2n\ln(2)} \int_{-n\ln(2)}^{n\ln(2)} r_\gamma(y)^2 dy \\ &= \frac{1}{2n\ln(2)} \sum_{i=-n}^{n-1} \int_{i\ln(2)}^{(i+1)\ln(2)} \left(1 - \frac{2^i[1 + y/\ln(2) - i]}{e^{y+\gamma}}\right)^2 dy \\ &= \cdots = 1 + \frac{3 + 4(1 - 4e^\gamma)\ln(2)}{16e^{2\gamma}\ln(2)^3}. \end{aligned} \tag{A.8}$$

Setting the derivative of Ψ to zero gives:

$$\begin{aligned} \frac{\partial}{\partial \gamma}\Psi(\gamma) &= \frac{4(2e^\gamma - 1)\ln(2) - 3}{8e^{2\gamma}\ln(2)^3} = 0 \\ 2e^\gamma - 1 &= \frac{3}{4\ln(2)} \\ c &= 2^{20}\ln\left(\frac{3}{8\ln(2)} + \frac{1}{2}\right)/\ln(2) \approx 60{,}801.48. \end{aligned} \tag{A.9}$$

Again 0.5 must be subtracted to compensate for the staircase effect; the best integer value is $c = 60{,}801$.

A.4 Lowest Mean Relative Error.. The points at which EXP intersects the exponential function are given by

$$\begin{aligned} e^{y+\gamma} &= \mathrm{EXP}(y+\gamma\,) \\ e^{y}e^{\gamma} &= 2^{k}\,(1+y/\ln(2)-k) \\ -e^{\gamma}\,\ln(2)/2 &= [\,k\ln(2)-y-\ln(2)]\,e^{k\ln(2)-y-\ln(2)} \\ W(-e^{\gamma}\,\ln(2)/2) &= k\ln(2)-y-\ln(2) \\ y/\ln(2)-k &= -W(-e^{\gamma}\,\ln(2)/2)/\ln(2)-1, \end{aligned} \tag{A.10}$$

where W denotes Lambert's function (Fritsch, Shafer, & Crowley, 1973; Corless, Gonnet, Hare, & Jeffrey, 1993; Corless, Gonnet, Hare, Jeffrey, & Knuth, 1996),[2] which satisfies $W(u)\,e^{W(u)} = u$. Each linear segment of EXP crosses the exponential at two points, ρ_+ and ρ_-, given by the two real-valued branches, W_0 and W_{-1}, of Lambert's function:

$$\rho_{+|-} \;\equiv\; -W_{0\,|-1}(z)/\ln(2)-1\,, \quad \text{where} \quad z \;\equiv\; -e^{\gamma}\,\ln(2)/2. \tag{A.11}$$

The mean relative error Φ as a function of γ can be computed by splitting the integral over the relative error $|\,r_\gamma\,(y)|$ at the crossover points $\rho_{+|-}$:

$$\begin{aligned} \Phi(\gamma\,) &\equiv \frac{1}{2n\ln(2)}\int_{-n\ln(2)}^{n\ln(2)} |\,r_\gamma\,(y)|\,dy \\ &= \frac{1}{2n\ln(2)}\sum_{i=-n}^{n-1}\left[\,\int_{i\,\ln(2)}^{(i+\rho_+)\ln(2)} r_\gamma\,(y)\,dy \;-\int_{(i+\rho_+)\ln(2)}^{(i+\rho_-)\ln(2)} r_\gamma\,(y)\,dy \;+\int_{(i+\rho_-)\ln(2)}^{(i+1)\ln(2)} r_\gamma\,(y)\,dy\right] \\ &= \cdots = 1+\frac{2}{\ln(2)}\left[\frac{W_{-1}(z)^2+1}{W_{-1}(z)}-\frac{W_0(z)^2+1}{W_0(z)}\right]-\frac{e^{-\gamma}}{2\ln(2)^2}\,. \end{aligned} \tag{A.12}$$

Setting the derivative of Φ to zero gives

$$\begin{aligned} \frac{\partial}{\partial\gamma}\,\Phi(\gamma\,) &= 4\ln(2)\,[W_{-1}(z)-W_0(z)]+e^{-\gamma}\,W_{-1}(z)W_0(z) = 0 \\ e^{-\gamma} &= 4\ln(2)\left[\frac{1}{W_{-1}(z)}-\frac{1}{W_0(z)}\right] \\ 1/8 &= e^{W_0(z)}-e^{W_{-1}(z)}. \end{aligned} \tag{A.13}$$

Now set $\nu_{+|-} \equiv e^{W_{0\,|-1}(z)}$. By definition, $W(z)\,e^{W(z)} = z$ for all branches of W, so $z = \nu_+\,\ln(\nu_+) = \nu_-\,\ln(\nu_-)$. In conjunction with equation A.13, this yields

$$\nu = (\nu+1/8)\,\ln(\nu+1/8)/\ln(\nu), \tag{A.14}$$

[2] I have written Octave/Matlab code that evaluates any branch of Lambert's W function for complex arguments. It is available on the Internet at: ftp://ftp.idsia.ch/pub/nic/W.m.

which can be solved numerically by iterating over equation A.14 from a suitable starting point $0 < \nu_0 < 7/8$. The result is

$$\begin{aligned} \nu &\approx 0.3071517227 \\ z = \nu \ln(\nu) &\approx -0.362566022 \\ \gamma \; = \ln(-2\,z) - \ln(\ln(2)) &\approx 0.045111411 \\ c = \gamma \, \cdot 2^{20} / \ln(2) &\approx 68{,}243.43. \end{aligned} \tag{A.15}$$

With the usual subtraction of 0.5 on account of the staircase effect, the best integer value is $c = 68{,}243$.

Acknowledgments

I thank Avrama Blackwell, Frank Dellaert, Felix Gers, and the anonymous reviewers for their helpful suggestions, and the developers of the Maple computer algebra system for creating such a useful tool. Lee Campbell of the Computational Neurobiology Lab at the Salk Institute has graciously provided access to and information about a variety of workstations for benchmarking purposes. This work was supported by the Swiss National Science Foundation under grant numbers 2100–045700.95 /1 and 2000–052678.97 /1.

References

Corless, R. M., Gonnet, G. H., Hare, D. E. G., & Jeffrey, D. J. (1993). Lambert's W function in Maple. *Maple Technical Newsletter*, *9*, 12–22.

Corless, R. M., Gonnet, G. H., Hare, D. E. G., Jeffrey, D. J., & Knuth, D. E. (1996). On the Lambert W function. *Advances in Computational Mathematics*, *5*(4), 329–359.

Fritsch, F. N., Shafer, R. E., & Crowley, W. P. (1973). Algorithm 443: Solution of the transcendental equation $w\,e^w = x$. *Communications of the ACM*, *16*, 123–124.

IEEE. (1985). *Standard for binary floating-point arithmetic*. ANSI/IEEE Std. 754–1985. New York: American National Standards Institute/Institute of Electrical and Electronic Engineers.

Lazzaro, J., & Wawrzynek, J. (1999). JPEG quality transcoding using neural networks trained with a perceptual error measure. *Neural Computation, 11*(1).

Received March 13, 1998; accepted July 2, 1998.