Manuscript Number : 2092

On a Fast Compact Approximation of the Exponential Function

Gavin C. Cawley
University of East Anglia

Abstract

In a recent issue, Schraudolph (Schraudolph, 1999) describes an ingenious, fast and compact approximation of the exponential function through manipulation of the components of a standard (IEEE-754 (IEEE, 1985)) floating point representation. This brief note communicates a re-coding of this procedure that overcomes some of the limitations of the original macro at little or no additional computational expense.

1 Motivation

Schraudolph quite rightly describes exponentiation as the quintessential non-linear function of neural computation, and provides C code implementing a fast approximation of this function, providing impressive speed improvements in both benchmark tests and in a real-world application (JPEG quality transcoding (Lazzaro and Wawrzynek, 1999)). The use of a static global variable is however problematic in a multi-threaded environment. This note describes a re-implementation of the algorithm that eliminates the global variable without incurring significant additional computational expense.

2 The Algorithm

IEEE standard 754 stipulates that double precision floating point numbers should be represented in the form $(-1)^s(1+m)2^{x-x_0}$, where s is the sign bit, m is the 52-bit normalised mantissa (a binary fraction in the range $(0,1]$) and x is the 11-bit exponent with a bias $x_0 = 1023$. Schraudolph's fast approximation to the exponential function is based on the identity $e^y = 2^{y/\ln 2}$. Setting the exponent, x, equal to the integer part of $y/\ln 2 + x_0$ provides a simple approximation to e^y. This approach can be implemented using a C/C++ union construct allowing a double precision IEEE-754 floating point number, d, to be stored in the same locations in memory as a structure comprised of two 32-bit signed integers, i and j, as shown in fig 1. Setting

$$\begin{aligned} i &= \frac{2^{20}y}{\ln 2} + 2^{20}x_0 - C \\ j &= 0 \end{aligned}$$

results in the integer part of d containing the integer part of $y/\ln 2 + x_0$, the factor of 2^{20} providing the necessary shift left of 20 places. The fractional part overflows into the most significant bits of the mantissa, providing a degree of linear interpolation between integral values. The parameter C provides some control over the error properties of the approximation, a value of 60801 minimises the root-mean-square (RMS) relative error.

3 The Implementation

The revised C++ implementation is shown in figure 2. The `EXP` macro has been replaced by an inline function, `exponential`, and the global static variable `_eco` is replaced by the local variable of `exponential`. This approach brings two benefits: Firstly this implementation is more suitable for use in multi-threaded programs as the static global variable has been eliminated. Secondly the use of a function, instead of a macro, more cleanly encapsulates

the algorithm and permits better use of type-checking rules. Note that this code can also be compiled by a standard ANSI C compiler if the `inline` keyword is omitted.

4 Benchmark Results

Table 1 displays benchmark results for the original and revised implementations of Schraudolph's approximation. The benchmark used is similar to that used by Schraudolph, except that for accuracy the sum of 10^9 exponentials of random arguments is computed. Note that the `exponential` function is slightly faster than the `EXP` macro, except in the case of the Sun workstation where the `EXP` macro is marginally faster.

Table 1: Seconds required for 10^9 exponentiations.

Manufacturer	Intel	Intel	Sun	DEC
Processor	Pentium	Pentium	UltraSparc	Alpha
Model/Speed	II/300	Xeon/450	Ultra 5/360	500au/500
`LITTLE_ENDIAN`	Yes	Yes	No	Yes
Op. System	WinNT	Linux	Sun OS 5.7	Digital UNIX V4.0E
Compiler	egcs-2.91.57	egcs-2.90.27	CC 5.0	DEC C V5.8-009
Source	C	C++	C++	C
Optimisation	-O3	-O1	-O4	-O2
`exp` (libm.a)	918	431	677	130
`EXP` macro	415	281	125	41
`exponential` function	391	261	129	40

The ISO/ANSI C++ standard (ISO/IEC, 1998) states that the `inline` keyword suggests, but does not require, that the statements comprising the body of the function should be expanded into the body of the calling function, eliminating the computational expense of the function call. However, most modern C and C++ compilers are capable of performing this optimisation, especially in the case of small leaf functions, such as this. One might still

expect the revised code to be slower as a local variable must be allocated and initialised each time the function is called, whereas the original macro is able to reuse a global variable that is statically allocated and initialised. The function though, does not have the side effect of modifying a global variable. As it is generally infeasible for the compiler to determine statically whether this value is ever used, the original macro must contain code to update the global variable, whereas the function can be optimised so that the temporary variable only exists within registers.

5 Summary

A re-coding of the splendid approximation of the exponential function described by Schraudolph is presented. Without incurring a significant additional computational expense, it provides the following additional benefits:

- The re-implementation is better suited to a multi-threaded environment as the static global variable has been eliminated.
- Type-checking rules can be applied more strongly, and the algorithm is encapsulated more cleanly, for improved reliability.

6 Acknowledgements

The author would like to thank Mark Fisher, Danilo Mandic and the anonymous reviewers for their helpful comments and Shaun McCullagh for his assistance in collecting the benchmark results.

References

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

ISO/IEC (1998). *Programming languages - C++* ISO/IEC Std. 144882-1998(E). American National Standards Institute, New York.

Lazzaro, J. and Wawrzynek, J. (1999). JPEG quality transcoding using neural networks trained with a perceptual error measure. *Neural Computation*, 11(1):267–298.

Schraudolph, N. N. (1999). A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862.

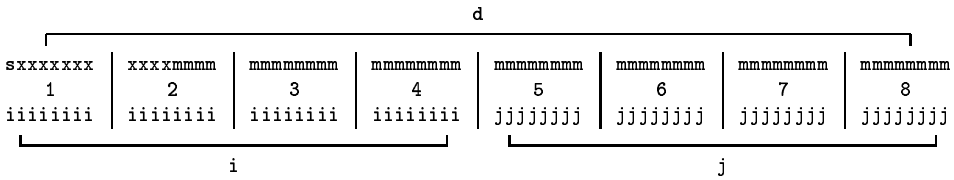

Figure 1: Bit representation of the `union` data structure used by the `exponential` function (Schraudolph, 1999).

```
#define EXP_A (1048576/M_LN2)

#define EXP_C 60801

inline double exponential(double y)
{
   union
   {
      double d;
#ifdef LITTLE_ENDIAN
      struct { int j, i; } n;
#elseif
      struct { int i, j; } n;
#endif
   }
   _eco;

   _eco.n.i = (int)(EXP_A*(y)) + (1072693248 - EXP_C);
   _eco.n.j = 0;

   return _eco.d;
}
```

Figure 2: C++ code fragment implementing a fast approximation of the exponential function.