November 17, 2013

Computation of Exponentials and Logarithms

With binary arithmetics

Jarmo Nikkanen

jnikkanen@kymp.net

THE LICENSE

FOREWORD

This document is created in a hope that it might give some ideas on how to compute 2-based exponentials and logarithms. Source codes are provided because some people can read code better than a mathematical descriptions. Remember that the codes are provided to clarify the idea, NOT to provide a well working solution. Source codes are missing rounding of the result, special case handling and additional shifts to improve precision.

FLOATING POINT

A single precision floating-point number is made from 23-bit Mantissa, 8-bit Exponent and a sign bit. There is also an implicit bit that isn't present in a floating point number itself but is often added during a bit manipulations. (i.e. 0x800000)

S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$$x = \left(1 + \frac{M}{2^{23}}\right) * 2^{E-127}$$

Value 1.0 can be archived by setting the mantissa bits to zero and exponent to 127 (that is zero due to exponent bias). If all mantissa bits are set to one then the resulting value is 1.999… that is 2.0-epsilon. You can't actually reach a zero with a floating point format, but a special floating point zero is defined by setting mantissa and exponent to zero. Also note that multiplication and division with powers of two such as (2, 4, 8, 16…) can be completed by adjusting the exponent E.

FLOAT-INTEGER CONVERSION

Integer can be converted to a floating point number by finding the index n of the highest active bit and setting the bit to zero and then shifting resulting value by $23 - n$ bits to left or right based on polarity. After that a value $127 + n$ will be written to the exponent E. Floating point number can be converted to an integer by shifting the mantissa $E - 23 - 127$ bits, where E is the biased value of the exponent. Shifting is done to left or right based on polarity.

```
int Float2Int(float iv)
{
    DWORD m = Mantissa(iv) | 0x800000;
    int   e = Exponent(iv); if (e<0) return 0;
    int   r = Shift(m, e-23); if (iv<0) r=-r;
    return r;
}
```

```
float Int2Float(int iv)
{
    DWORD a = abs(iv);
    if (a==0) return 0.0f;
    DWORD n = HighestBitZero(&a);
    return CreateFloat(Shift(a, 23-n), n, iv<0);
}
```

BASICS OF EXPONENTIALS AND LOGARITHMS

Logarithms may seem to be less important in general computations but they do provide a way to compute several other more important functions like: $x^y = 2^{(y \log_2 x)}$, $\sqrt{x} = 2^{(0.5 \log_2 x)}$, $y/x = 2^{(\log_2 y - \log_2 x)}$, $1/x = 2^{(-\log_2 x)}$. Also note that $e^x = 2^{zx}$, where $z = 1/\ln 2$. These may prove useful when working with mathematics in a highly limited environment. *But note that a logarithm doesn't take negative input value.*

Computation of exponentials and logarithms will rely on a basic identity: $e^{(a+b)} = e^a e^b$ and $\ln(ab) = \ln a + \ln b$. For an example $2^{11} = 2^{(8+2+1)} = 2^8 \cdot 2^2 \cdot 2^1 = 2048$. Computation process for an exponential is pretty simple. First, initialize a value 1.0 into a result accumulator. Then, as long as the source (input) value s is non-zero. Subtract a known constant $c[i]$ from the s and multiply the result accumulator with $2^{c[i]}$. If you choose the constants $c[i]$ properly then the subtraction will get simplified to bit checking and with some other values of $c[i]$ the multiplication will become a bit shifting.

2-BASED EXPONENTIAL FUNCTION

This section will describe a method to compute a 2-based exponential function $y = 2^x$, where the x is a floating point number. We could compute e-based exponential directly but a 2-based allows us to take a better advantage of the floating point format. In order to proceed with the computations we need a table for multiplications. The table contains 2x23 values, two values for each bit in the mantissa. Values for a bits 15-22 are shown in a table below.

Index	Bit weight	fMul	iMul
b	$w = 1/2^{23-b}$	$m = 2^w$	$i = 1/2^w$
22	0.5000000	1.4142135	0.7071068
21	0.2500000	1.1892071	0.8408964
20	0.1250000	1.0905077	0.9170040
19	0.0625000	1.0442737	0.9576033
18	0.0312500	1.0218972	0.9785720
17	0.0156250	1.0108893	0.9892280
16	0.0078125	1.0054299	0.9945995
15	0.0039063	1.0027113	0.9972960
…	…	…	…

Let's use a value 9.890625 in our example case. Value has a mantissa of 0x1E4000 and unbiased exponent +3. Below is the bit field of the mantissa. The left most bit is the implicit bit added to the value.

1	0	0	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

At first we have to shift the bits to the left by three because the exponent is +3. While doing this, three bits from the mantissa will be shifted to the integer (whole number) part of a fixed point value (bits 23-29). (The implicit bit is already there). If the exponent is negative then the shifting would be done to right and the implicit bit would be shifted into the fractional part.

w			64	32	16	8	4	2	1	,500	,250	,125	,063	,031	,016																	
	x	x	0	0	0	1	0	0	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

When computing the value of the exponential we have to compute the product of the multipliers of each active bit. Then we multiply the result with $2^9 = 2^8 2^1 = 512$, because the integer part is +9. But since we are working with a floating point numbers the integer part +9 can be simply added to the exponent E of the final result. Of course, we could have the multipliers for bits 23-29 in the table but that would be inefficient due to unnecessary multiplications. Positive values are using fMul table and negative values iMul table. Just fetch a multiplier from the table using index of each active bit in a fixed point fraction. Here's an example:

$2^{+9.890625}$ = (fMul[22]*fMul[21]*fMul[20]*fMul[17]) * 512 = 949,237312

$2^{-9.890625}$ = (iMul[22]*iMul[21]*iMul[20]*iMul[17]) * 1/512 = 0,001053477

```
float Exp2(float iv)
{
        DWORD m  = Mantissa(iv) | 0x800000;
        int   e  = Exponent(iv);
        DWORD b  = Shift(m, e); // Shift bits to left or right based on polarity
        float rv = 1.0f;

        for (int i=0;i<23;i++) { // Perform a multiplication for each active bit in a fixed point fraction
            if (b&1) rv *= fMul[i]; b>>=1;
        }

        DWORD *d = (DWORD*)&rv;
        if (e>=0) *d += (m>>(23-e))<<23; // Add the integer part to the exponent
        return rv;
}
```

The algorithm above is simplified to work only with positive input values [0-127].

2-BASED EXPONENTIAL FUNCTION WITHOUT MULTIPLICATION

Index	Multiplier	fSub	dwSub
n	$m = 1 + 1/2^n$	$s = \log_2 m$	
0	2.0000000	1.0000000	0x800000
1	1.5000000	0.5849625	0x4AE00D
2	1.2500000	0.3219281	0x2934F0
3	1.1250000	0.1699250	0x15C01A
4	1.0625000	0.0874628	0x0B31FB
5	1.0312500	0.0443941	0x05AEB4
6	1.0156250	0.0223678	0x02DCF2
7	1.0078125	0.0112273	0x016FE5
…	…	…	…

The method described in a previous section could be good one if you have a fast hardware multiplier available, otherwise the multiplication might become a problem. There is a way to avoid the multiplication. As you probably already know that a multiplication or a division with a powers of two can be completed by shifting the bits. So, we could try to approach the problem from the opposite direction by trying to find the multipliers those will allow us to do the multiplication easily. One multiplier that fits in our needs is $[1 + 1/2^n]$. If we want to multiply a value x with this multiplier then we can do it by shifting the bits: *x = x + (x>>n);* But in that case we need a table of corresponding subtraction values. We can't use this multiplier in a per bit basis as we did in the previous section. The subtraction values can be computed by taking a base-2 logarithm from the multiplier. The first eight values are shown in a table right here. Upper 8-bits of dwSub are unused in this example, it would be a good idea to put them in a good use.

```
float Exp2(float iv)
{
        int   i = 1;
        int   e = Exponent(iv);
        DWORD m = Mantissa(iv) | 0x800000;
        DWORD b = Shift(m, e) & 0x7FFFFF; // Shift bits to left or right based on polarity
        DWORD v = 0x800000;               // Initial starting value

        while (i<23) {
           if (b>=dwSub[i]) {
               b-=dwSub[i];
               v = v + (v>>i);
           } else i++;
        }

        v &= 0x7FFFFF;                    // Form a floating point number. Remove implicit bit.
        v += 127<<23;                     // Initialize exponent
        if (e>=0) v += (m>>(23-e))<<23;   // Compute the exponent and add it to the float
        return *(float*)&v;               // Convert dword to float
}
```

The code above is simplified to use only a positive values. Also, you may need to adjust the fixed point by bit shifting to reduce a round-off error accumulation. These shifts are removed from the code as well as rounding of the result. To compute the exponential using a negative input value you need to use $[1 - 1/2^n]$ multiplier instead of $[1 + 1/2^n]$. See the 2-based logarithm section for some additional details.

2-BASED LOGARITHMIC FUNCTION

Index	Bit weight	fDiv	iDiv
b	$w = 1/2^{23-b}$	$m = 2^w$	$i = 1/2^w$
22	0.5000000	1.4142135	0.7071068
21	0.2500000	1.1892071	0.8408964
20	0.1250000	1.0905077	0.9170040
19	0.0625000	1.0442737	0.9576033
18	0.0312500	1.0218972	0.9785720
17	0.0156250	1.0108893	0.9892280
16	0.0078125	1.0054299	0.9945995
15	0.0039063	1.0027113	0.9972960
…	…	…	…

The computation process of 2-based logarithm is pretty much the opposite than with the exponential. As long as the source (input) value is greater than 1.0. The input value is divided by a known constant (as large as possible but less than the source value) and then a 2-based logarithm of the constant is added to the output value. Each division of the source value will make it to approach 1.0. In this example case adding of $\log_2 c[i]$ is simplified to a bit setting. Here is a table containing a divisor and it's inverse. The values are the same as in the first section. We could have a longer table containing a divisors for bits 23-31 but it would be a waste. Instead, we can simply divide the input value with 2^E in other words setting the exponent E to 127 from the source value (that is zero due to exponent bias). That will scale the input value in range [1-2[. Then we have to add the original value of the exponent into the final result. The exponent E of the input value itself is the integer (whole number) part of the final result. In the code below we subtract 1 from the 'e', because the implicit bit is added to the float returned by CreateFloat().

```
float Log2(float iv)
{
        DWORD a   = 0;
        int    e   = Exponent(iv);
        SetExponent(&iv,0);              // Set exponent to zero (i.e. scale input range to [1-2[)
        for (int i=22;i>=1;i--) {
            if (iv>=fDiv[i]) iv*=iDiv[i], a|=1;    // Divide and set lowest bit
            a<<=1;                                 // Shift the bits
        }
        return Int2Float(e-1) + CreateFloat(a);
}
```

2-BASED LOGARITHM WITHOUT MULTIPLICATION

This section will describe another method using a bit shifting instead of floating point multiplication. The principles are exactly the same as described in an earlier sections. Also the table that is used here is exactly the same as the one required by negative values earlier. It doesn't really matter what value is used in a division/multiplication as long as the source value approaches 1.0 in every step and the relationship between the divisor and the value being added is correct. Also, note that you may need to use the same divisor/multiplier more than once. For an example: if the input value is 1.9 you need to multiply it twice with 0.75. That is why the i++ is in the 'else' statement.

Index	Multiplier	fAdd	dwAdd
n	$m = 1 - 1/2^n$	$\lvert\log_2 m\rvert$	
1	0.5000000	1.0000000	0x800000
2	0.7500000	0.4150375	0x351FF3
3	0.8750000	0.1926451	0x18A898
4	0.9375000	0.0931094	0x0BEB02
5	0.9687500	0.0458037	0x05DCE5
6	0.9843750	0.0227201	0x02E87D
7	0.9921875	0.0113153	0x0172C7
8	0.9960938	0.0056466	0x00B906
…	…	…	…

```
float Log2(float iv)
{
        DWORD lg = 0;
        int    i  = 2;
        int    m  = Mantissa(iv) | 0x800000;
        while (i<23) {
            int t = m - (m>>i);
            if (t>0x800000) {
                m = t;
                lg += dwAdd[i]; // Table index 0 is unused
            } else i++;
        }
        return Int2Float(Exponent(iv)-1) + CreateFloat(lg);
}
```

2-BASED LOGARITHM WITHOUT TABLE

If we look at the function we developed in the section 2-based logarithmic function. We can notice that the actual computation is pretty simple since the *fDiv[i]* = $\sqrt[x]{2}$, where $x = 2^i$.

$$if\ \left(v \geq \sqrt[x]{2}\right)\ then\ v = v/\sqrt[x]{2}$$

We are compering v to the x:th root of 2. Then, why not raise both sides of the "*statement*" to power of x, resulting following statement below. This can be easily archived by squaring the input value v each step in a loop. Also remember that division by two can be completed by decrementing the exponent E.

$$if\ (v^x \geq 2)\ then\ v^x = v^x/2$$

```
float Log2(float iv)
{
        DWORD a  = 0;
        int   e  = Exponent(iv);
        SetExponent(&iv,0);             // Set exponent to zero (i.e. scale input range to [1-2])
        for (int i=22;i>=1;i--) {
            iv*=iv;
            if (iv>=2.0f) iv*=0.5f, a|=1; // Set lowest bit
            a<<=1;                      // Shift the bits
        }
        return Int2Float(e-1) + CreateFloat(a);
}
```

2-BASED EXPONENTIAL WITHOUT TABLE

Computation of 2-based exponential without a table is pretty easy. We can simply replace the table by taking a square root from a previous root in a loop. But doing so would be mathematically expensive. Therefore, it's not much an option.

```
float sq = 2.0f;
for (int i=0;i<23;i++) {
    sq=sqrt(sq);                // Take a square root
    if (b&0x400000) rv *= sq;   // Perform a multiplication for each active bit in a fixed point fraction
    b<<=1;                      // Shift the bits
}
```

In theory, it would be possible to take the n:th root of the lowest bit and start squaring it up, but the value is so close to 1.0 that a computer doesn't really make a difference between the two. There isn't enough precision to do the squaring without doubling the bits in the mantissa. The code below will show the idea.

```
float sq = 1.0000000826295864;    // 2^23:th root of 2
for (int i=0;i<23;i++) {
    if (b&1) rv *= sq;            // Perform a multiplication for each active bit in a fixed point fraction
    b>>=1;                        // Shift the bits
    sq*=sq;                       // Square it up
}
```

How about splitting the value 1.0000000826295864 into a two different values 1.0 and 8.26295864e-8. You probably recall from the school that $(a+b)^2 = a^2 + 2ab + b^2$. This will work in our favor. In our case the variable a would be 1.0 and that will simplify the equation to $(1+b)^2 = 1 + 2b + b^2$. Also we don't want to compute the square, instead we want the next b_{n+1} which is given by following equation below. *Last two underlined forms are usable.*

$$b_{n+1} = (1+b_n)^2 - 1 = (1 + 2b_n + b_n^2) - 1 = \underline{2b_n + b_n^2} = \underline{b_n(2+b_n)}$$

```
float Exp2(float iv)
{
        DWORD m  = Mantissa(iv) | 0x800000;
        int   e  = Exponent(iv);
        DWORD b  = Shift(m, e);         // Shift bits to left or right based on polarity
        float rv = 1.0f;
        float sq = 8.26295864e-8f;      // (2^23:th root of 2) minus 1.0

        for (int i=0;i<23;i++) {
            if (b&1) rv *= (1.0f + sq); // Perform a multiplication for each active bit
            sq *= (2.0f+sq);            // Square it up
            b>>=1;                      // Shift the bits
        }
        DWORD *d = (DWORD*)&rv;
        if (e>=0) *d += (m>>(23-e))<<23; // Add the integer part to the exponent
        return rv;
}
```

```
int Exponent(float f)
{
        DWORD *d = (DWORD*)&f;
        int    e = (*d>>23)&0xFF;
        return e - 127; // Return signed unbiased exponent
}

DWORD Mantissa(float f)
{
        DWORD *d = (DWORD*)&f;
        return *d&0x7FFFFF;
}

bool Sign(float f)
{
        DWORD *d = (DWORD*)&f;
        return ((*d&0x80000000)!=0);
}

float CreateFloat(DWORD m, int e, bool s)
{
        DWORD d = (m&0x7FFFFF) + ((e+127)<<23);
        if (s) d |= 0x80000000; // Add a sign bit
        return *(float*)&d;
}

float CreateFloat(DWORD m)
{
        DWORD d = (m&0x7FFFFF) + (127<<23);
        return *(float*)&d;
}

void SetExponent(float *f, int e)
{
        *f = CreateFloat(Mantissa(*f), e, Sign(*f));
}

DWORD Shift(DWORD w, int d)
{
        if (d<0) return w>>(-d);
        return w<<d;
}

DWORD HighestBitZero(DWORD *x)
{
        for (int i=0;i<32;i++) {
            DWORD q = 0x80000000>>i;
            if (*x&q) { *x-=q; return 31-i; }
        }
        return 0;
}
```