METHOD FOR CALCULATING PRECISE LOGARITHM OF A SUM AND SUBTRACTION

ROLAND PIHLAKAS
22. MAY 2007

Updated in 2009: to include method for calculating logarithm of subtraction

Abstract.

Disclosed is a method to compute the precise value of the logarithm of a sum.
A number of practical problems can result in having too big or small values in intermediate values of a calculation. Then one tries to take logarithm of these values and operate on logarithms instead. For example this is often the case in Artificial Intelligence and Speech Recognition, for example when the numbers represent probabilities in Bayesian Networks.

Also because multiplication and division is sometimes computationally more expensive than addition and subtraction, it is interesting to consider the log-values of x instead of x itself. In practice, for performance reasons also, integer arithmetic is sometimes used instead of floating-point arithmetic. Of course these two additional performance considerations apply only in the case when logarithm of summation and subtraction is done relatively more rarely during the calculations, so that other performance benefits overweight the losses.

In the case: $log\ (p \cdot q) = log\ p + log\ q$ is very easy to compute, but the problem is then to compute (or approximate) the value of $log\ (p1 + p2)$ from the value of $log\ p1$ and $log\ p2$.

Let us assume that $log\ a$ and $log\ b$ are known, and that we want to approximate $log\ (a + b)$.

Most basic solution would be calculating
```
sum_log = log(exp(a_log) + exp(b_log),
```
where $a\_log = log\ a$ and $b\_log = log\ b$ and therefore $sum\_log = log(a + b)$.
But the method I propose requires calling only one **exp**() and one **log**(), instead of two **exp**() and one **log**() in the basic solution. Additionally, the proposed method has the critical advantage of not overflowing in case of large numbers of a and b.

The method is based on the notion that
$ln(a + b) = ln\{exp[ln(a) - ln(b)] + 1\} + ln(b)$.

Method's code in C++:

```
float add_lns(float a_ln, float b_ln)
{
    if (abs(a_ln - b_ln) >= 15.942385)  //2^23-1 = 8388607. log of that is
15.942385033669445460387166503854
    {
        return max(a_ln, b_ln);   //this branch is necessary, to avoid
shifted_a_ln = a_ln - b_ln having too big value
    }
    else
    {
        float shifted_a_ln = a_ln - b_ln;
        float shifted_sum = exp(shifted_a_ln) + 1;
        float shifted_sum_ln = log(shifted_sum);
        float unshifted_sum_ln = shifted_sum_ln + b_ln;
        return unshifted_sum_ln;
    }
}
```

Corresponding Matlab code:

```
function R = add_lns(a_ln, b_ln)   % ln(a + b) = ln{exp[ln(a) - ln(b)] + 1} +
ln(b)
    if (abs(a_ln - b_ln) >= 36.043653389117155)  % 2^52-1 = 4503599627370495.
log of that is 36.043653389117155867651465390794
        R = max(a_ln, b_ln);   % this branch is necessary, to avoid
shifted_a_ln = a_ln - b_ln having too big value
    else
        shifted_a_ln = a_ln - b_ln;
        shifted_sum = exp(shifted_a_ln) + 1;
        shifted_sum_ln = log(shifted_sum);
        unshifted_sum_ln = shifted_sum_ln + b_ln;
        R = unshifted_sum_ln;
    end
end
```

Comments

1) Explanation for comparisons "**if** (**abs**(a_ln - b_ln) >= 15.942385)" and "**if** (**abs**(a_ln - b_ln) >= 36.043653389117155)":
2^23-1 is the maximal value (precision) of mantissa of 32-bit single-precision floating point value in Math Processor. Analogously 2^52-1 is the maximal value (precision) of mantissa of 64-bit double-precision floating point (the default floating point data type in Matlab). So it is meaningless to *add* numbers different by this order of magnitude - the bits of the resulting value will depend only on, that is - equal to - the original bits of the largest of the two values. Moreover, this would not only be meaningless - it would also make the method vulnerable to big numbers and overflows caused by them, which would nullify the point of the method.

In case one is using double-precision floats one likely wants to adjust this constant to an appropriate value.

2) Another way to look at the method is not to think about the proof
ln(a + b) = ln{exp[ln(a) - ln(b)] + 1} + ln(b)
but instead to think about it as scaling the value of *a* down to the "units" of *b* and replacing the value of *b* with *1* accordingly. That is *a′ = a / b* and *b′ = 1* (assuming without the loss of generality that *a > b*). Then the scaled down (or scaled up) numbers are added and logarithm is taken of the resulting sum. Finally, the result is scaled back to original units, by adding the *log b*. That is equivalent to the formula:
a + b = (a / b + 1) · b = (a′ + b′) · b.
→ ln(a + b) = ln(a / b + 1) + ln(b) = ln{exp[ln(a / b)] + 1} + ln(b) =
= ln{exp[ln(a) - ln(b)] + 1} + ln(b).
Because the summation is done while the numbers are "scaled down" the overflows will not occur.

Calculating logarithm of subtraction

The method is based on the notion that
ln(a - b) = ln{exp[ln(a) - ln(b)] - 1} + ln(b), if *a > b*.
Note that we cannot compute the logarithm when *a ≤ b*.

The principle is analogous to summation's principle: Given that
a′ = a / b and *b′ = 1*
we can represent the scaled down (or scaled up) values as
a - b = (a / b - 1) · b = (a′ - b′) · b
→ ln(a - b) = ln(a / b - 1) + ln(b) = ln{exp[ln(a / b)] - 1} + ln(b) =
= ln{exp[ln(a) - ln(b)] - 1} + ln(b).

Method's code in C++:

```
float sub_lns(float a_ln, float b_ln)
{
    if (a_ln - b_ln >= 15.942385)  //2^23-1 = 8388607. log of that is
15.942385033669445460387166503854
    {
        return a_ln;   //this branch is necessary, to avoid shifted_a_ln = a_ln
- b_ln having too big value
    }
    else
    {
        float shifted_a_ln = a_ln - b_ln;
        float shifted_diff = exp(shifted_a_ln) - 1;
        float shifted_diff_ln = log(shifted_diff);
        float unshifted_diff_ln = shifted_diff_ln + b_ln;
        return unshifted_diff_ln;
    }
}
```

Corresponding Matlab code:

```
function R = sub_lns(a_ln, b_ln)   % ln(a - b) = ln{exp[ln(a) - ln(b)] - 1} +
ln(b)
    if (a_ln - b_ln >= 36.043653389117155)  % 2^52-1 = 4503599627370495. log of
that is 36.043653389117155867651465390794
        R = a_ln;   % this branch is necessary, to avoid shifted_a_ln = a_ln -
b_ln having too big value
    else
        shifted_a_ln = a_ln - b_ln;
        shifted_diff = exp(shifted_a_ln) - 1;
        shifted_diff_ln = log(shifted_diff);
        unshifted_diff_ln = shifted_diff_ln + b_ln;
        R = unshifted_diff_ln;
    end
end
```