Implementing the exponential function in C

T.T. Lindkvist

1 Making the computer understand

As we know the exponential function `exp(x)`, is simply Euler's number, $e \approx 2.71828$, raised to the power of x - `exp(x)` $= e^x$. However, computers really only know the simple operators +, -, * and /. Using the Taylor series for the exponential function, we can easily implement it

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots \tag{1}$$

Of course we cannot do an infinite sum, but luckily this converges fast for small x. So we take the first $N = 10$ terms. This leaves a problem for larger x's, where the approximation is not so good. To combat this, we can use a rule of exponents

$$e^x = e^{x/2} \cdot e^{x/2} \, . \tag{2}$$

Suppose we want to compute `exp(x)` for some x larger than some x_{max}, where we know precision becomes an issue, we simply apply this rule of exponents recursively until we can get by, by calculating `exp(x')` for some $x' = x/2^n < x_{max}$. We can also make use of another property of exponents

$$e^{-x} = \frac{1}{e^x} \, , \tag{3}$$

and hereby only need to evaluate the function for positive argument x - making all terms in the series positive. We do this, since summing terms of alternating sign can make the calculation lose precision, since the terms are of comparable order of magnitude. Also, in order to not lose precision when adding the smaller (higher order, since x is small) terms, one could add the numbers from highest to lowest order, or in order to not use so many multiplications, write the terms inside one another, in this convoluted fashion.

$$\sum_{n=0}^{N} \frac{x^n}{n!} = 1 + x\left(1 + \frac{x}{2}\left(1 + \frac{x}{3}\left(1 + \frac{x}{4}\left(1 + \frac{x}{5}\left(1 + \frac{x}{6}\left(\cdots\right)\right)\right)\right)\right)\right)$$

2 C Implementation

Suppose we choose the maximum x, where the Taylor expansion is directly used, $x_{max} = 1/8$, and a maximum order of evaluation $N = 10$, then the function can be written in C, like this:

```
double ex(double x){
    if(x<0)     return 1/ex(-x);
    if(x>1./8) return pow(ex(x/2), 2);
    return 1+x*(1+x/2*(1+x/3*(1+x/4*(1+x/5*\
    (1+x/6*(1+x/7*(1+x/8*(1+x/9*(1+x/10)))))))));
}
```

3 Comparison of implementations

On figure 1, our implementation is plotted together with the implementation from `math.h`. As seen, they give approximately the same values, and with a very low relative error $< 10^{-14}$, as seen on figure 2. We can also look at

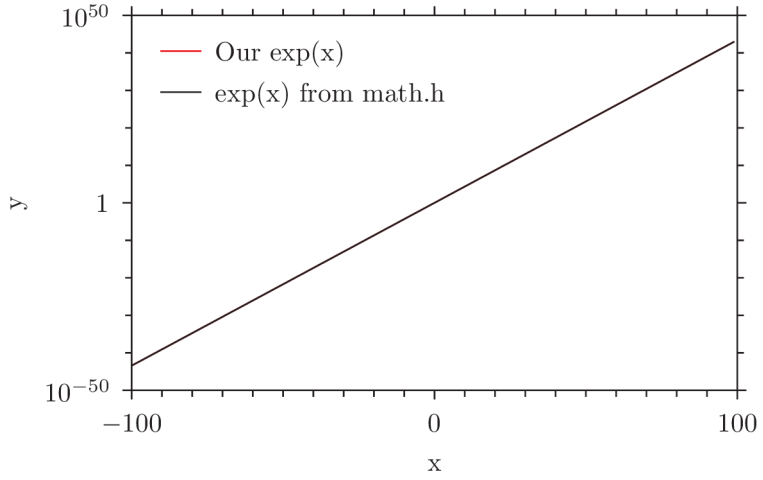

Figure 1: Shows our implementation together with the implementation from `math.h`. Note: the y-axis is logarithmic.

the difference between only calculating the positive terms, and also evaluating the Taylor series with negative terms. On figure 3 the relative error of the two expressions, with respect to `exp` from `math.h`. These expressions are of course equal for positive x, but at negative x, the expression with the negative terms, deviates slightly from the one with positive terms. We can also take a look at using the convoluted expression versus the ordinary sum of terms. This can be seen on figure 4. Since we are using doubles, this yielded no difference when summing backwards from highest to lowest order, but when evaluating

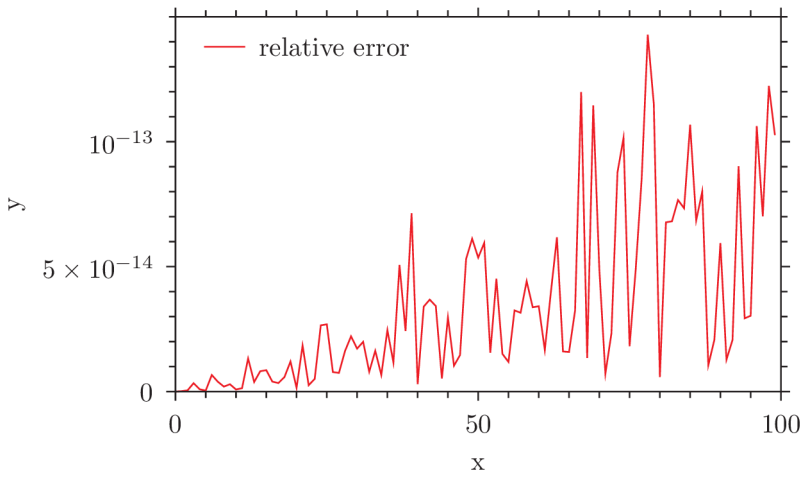

Figure 2: Shows the relative error of our implementation compared to the *correct* implementation from `math.h`. Note: the y-axis is logarithmic.

from lowest to highest order, a difference occurred, and this is slightly more imprecise.

Small note: steps can be seen on the graphs at powers of two - since the cutoff from direct evaluation of the Taylor series is a power of two. This is better seen with a large number of evaluation points, so this is better illustrated on figure 5.

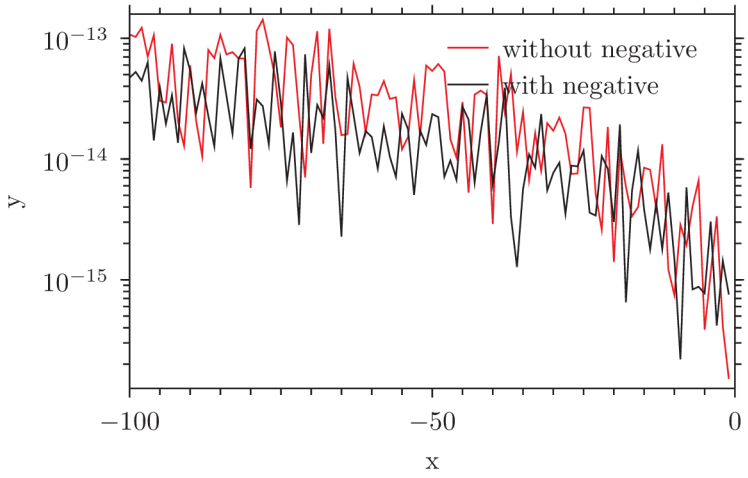

Figure 3: Shows the relative error of our implementation with and without evaluation of negative terms.

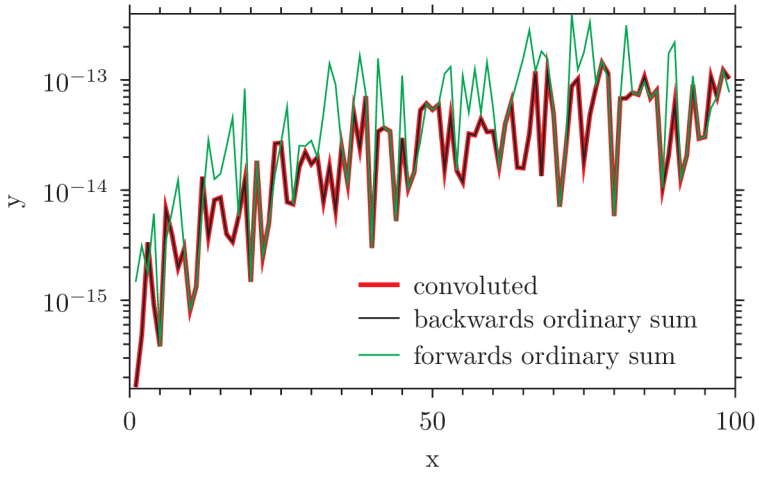

Figure 4: Shows the relative error of the convoluted expression and summing the terms from lowest to highest order.

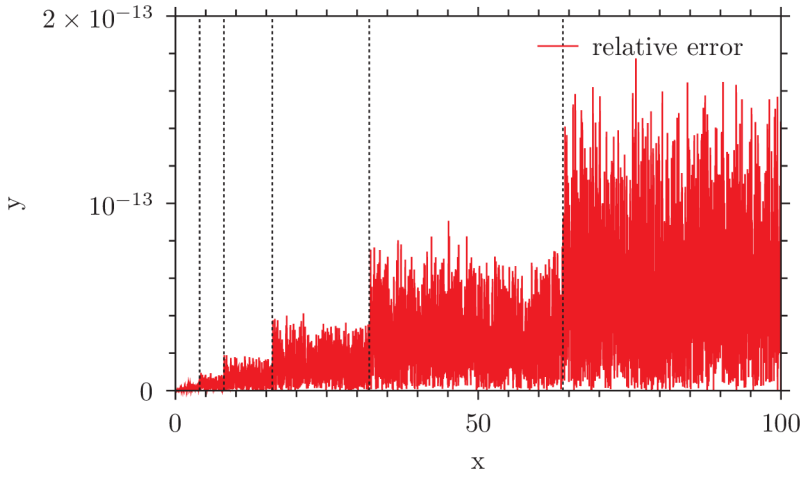

Figure 5: Replica of figure 2, but with larger number of evaluation points and with dashed lines for multiples of two (2^n for $n > 1$)