Numerical Implementation of the Natural Exponential

Josefine Bjørndal Robl

June 10, 2021

Introduction

There are several different ways of defining the natural exponential, $\exp(x)$. One of these is to define it as the limit

$$\exp(x) = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n , \quad \forall x \in \mathbb{R} . \tag{1}$$

Other ways could be to define it as the solution to the differential equation

$$y' = y , \tag{2}$$

since the exponential fucntion has the remarkable property of being its own derivative, $\mathrm{d}\exp(x)/\mathrm{d}x = \exp(x)$, or one could define the exponential function as one of the in physics most used ways: By its power series (i.e. by its Maclaurin series, which is a Taylor series around zero):

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} . \tag{3}$$

This definition is very useful when doing numerical calculations, since one then needs to use approximate expressions.

Using ones knowledge of complex function theory, it should be rather straight forward to show, that the exponential function has a convergence radius of infinite, which makes the function well-defined in the complex plane (i.e. on all of $\mathbb{C}$).

Numerical implementation

As mentioned above, the definition in eq. 3 is handy when working in the nummerical framework, since it will be approximate when we chose the sum from $n = 0$ to $n = N$. One can now rewrite the sum as

$$\exp(x) = 1 + x\left(1 + \frac{x}{2}\left[1 + \frac{x}{3}\{\dots\}\right]\right) , \tag{4}$$

such that the equation now only consists of basic operations as addition and multiplications, and therefore avoids power functions and factorials, which – per logic – are slower than the usual multiplication. Another advantage when using this rewriting instead of eq. 3 is, that for eq. 4 the computer will start the calculation from the innermost bracket, that meaning the smallest contributions are calculated and added together first, which we in exercise Epsilon have seen makes quite som difference. This though becomes a disadvantage for this methods for negative values, as these will have the largest contributions calculated first, but this can be solved just by defining the function recursively to return the inverse of the function taken of the positive value, if the input value is negative, i.e.:

if $x < 0$ **then return** $\frac{1}{\exp(-x)}$
end if

The second problem our implementation is facing is the fact, that the implementation cannot start at infinity as it does in the mathematical calculations. This issue is adressed by noting higher order terms becomes negligible small for small input values, thus we only have to ensure the argument to be small for the implementation to work. This is done using another recursive algorithm:

if $x > \varepsilon$ **then return** $\exp\left(\frac{x}{2}\right)^2$
end if

Here the value of ε is chosen to be small, and for higher precision lower values of ε is required. The idea of this snip of code is that mathematically

$$\exp(x)^2 = \exp(2x) , \tag{5}$$

and therefore

$$\exp\left(\frac{x}{2}\right)^2 = \exp\left(2\frac{x}{2}\right) = \exp(x) . \tag{6}$$

Thus mathematical this does not change anything. For a computer, though, this change is a big deal. Here it is possible for the computer to now calculate the exponential of half the number (thus, hopefully making it small enough,

otherwise this recursive code is performed again), and then double the result at the end. This makes the code much more reliable.

Adding together the two recursive codes from above we end up with a code on the form of

function $\exp(x)$
  if $x < 0$ **then return** $\frac{1}{\exp(-x)}$
  end if
  if $x < \varepsilon$ **then return** $\exp\left(\frac{x}{2}\right)^2$
  end if
  return $1 + x\left(1 + \frac{x}{2}\,[\text{to order } n]\right)$
end function

Implementation in C

For our implementation we have chosen $\varepsilon = 1/8$ and the number of terms are $n = 10$. For this, the code implemented in the C-language looks like:

```
double exponential(double x){
        if (x < 0) {
                return 1/exponential(-x);
        }
        if (x > 1./8) {
                return pow(exponential(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)))))))));
}
```

The last advantage I will note for this implementation is, that the function always will return a series, where each term is positive, which makes it converge much faster than if the signs of the terms were alternating.

On Figure 1, a plot of our numerical implementation is presented alongside a build-in exponential function rutine from math.h.

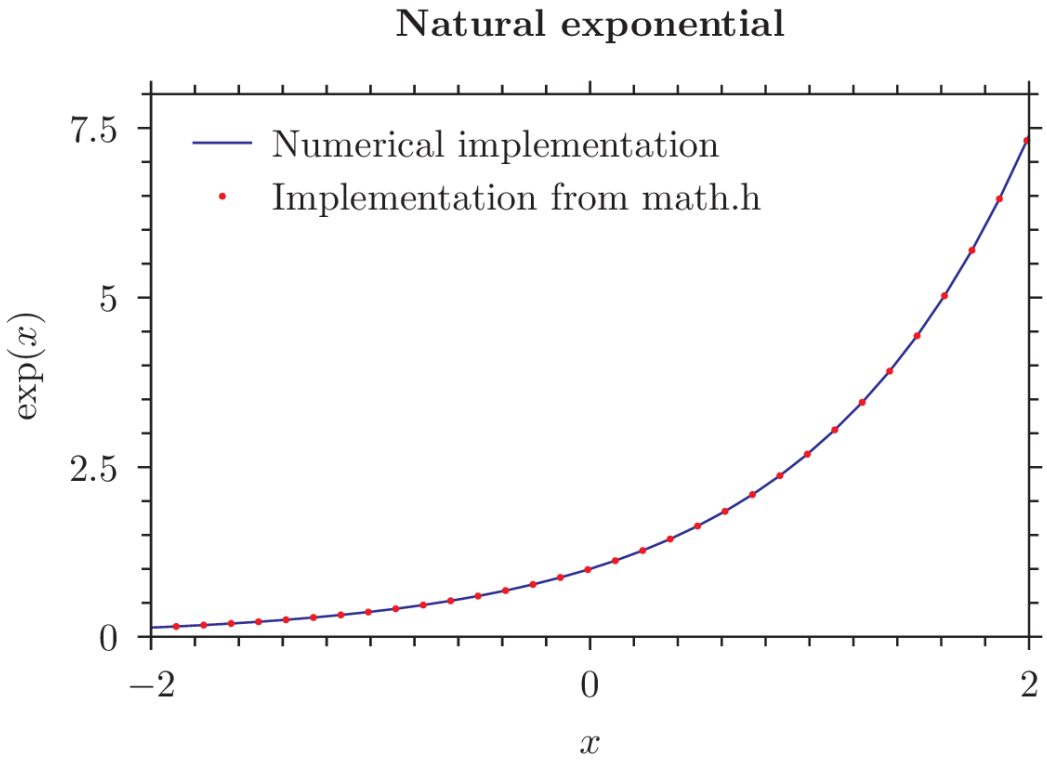

Figure 1: Plot of our numerically implementet exponential function rutine vs. the rutine from math.h for the C-language.