APPROXIMATING SMOOTH NONLINEAR FUNCTIONS IN FP32

ADELINA ANDREI

1. Task

The goal of this project is to design a compact and hardware–friendly system that can approximate and evaluate smooth nonlinear functions $f : \mathbb{R} \to \mathbb{R}$ such as the ones present in neural network accelerators. The system consists of two maps:

$$(\mathbb{R} \to \mathbb{R}) \longrightarrow \mathbb{R}^B, \qquad \mathbb{R}^B \longrightarrow (\mathbb{R} \to \mathbb{R}),$$

where the first map constructs a low–dimensional representation $\theta \in \mathbb{R}^B$ of the function, and the second evaluates this representation using only fp32 arithmetic with no operations more expensive than multiplication trying to optimize for different things like approximation error, evaluation cost, and B cost.

2. Approximation domain

Before choosing how to approximate a nonlinear function $f : \mathbb{R} \to \mathbb{R}$, there are two main ideas we can consider:

(i) **Approximate directly on $\mathbb{R}$.** Both polynomial and standard rational approximations only give guaranteed error control on the interval used during construction.

(ii) **Map $\mathbb{R}$ into $[-1, 1]$.** Do a change of variables via functions like $t = \tanh(\alpha x)$ or $t = \frac{2}{\pi}\arctan(x)$, but by compressing our unbounded interval into a compact one, we create very steep or very flat regions near $t = \pm 1$ where it is known that approximation is hardest and again we don't have guaranteed error control since the approximation error in t doesn't translate uniformly back to error in x because the change of variables distorts distances.

Both approaches leave us with an uncontrolled effective interval where the approximation is accurate. Therefore, they got me to the assumption: instead of letting that interval be defined implicitly by the method itself, we simply let the user specify the interval $[a, b]$ where accuracy is needed. Outside $[a, b]$ we could simply clamp $x' = \mathrm{clip}(x, a, b)$ and define $\widehat{f}(x) = f(x')$. We will be working in MATLAB.

3. Polynomial approximation and representation

We use Chebfun to construct to approximate f via Chebyshev expansions $p(t) = \sum_{k=0}^{D} c_k T_k(t)$ for $t = \frac{x-m}{h}$, where $m = \frac{a+b}{2}$ and $h = \frac{b-a}{2}$. More can be said here as to why, but basically Chebyshev polynomials are numerically stable under the Clenshaw recurrence and also achieve high accuracy for small degrees since for analytic functions the coefficients c_k decay geometrically. We use Chebfun to construct this representation because it adaptively samples

the function at Chebyshev points, it terminates when the tail coefficients fall below a tolerance, and it supports both global approximation (splitting off) and piecewise approximation (splitting on) constructions by introducing breakpoints where singularities are detected. .[1]

3.1. **fp64 construction: Fapprox.m.** Given a domain $[a, b]$, a function handle f, and a Chebfun tolerance ε, the code does the following:

(1) Constructs `p_off = chebfun(f,[a b],'splitting','off')`. This gives a global Chebyshev polynomial with degree $D_{\text{off}} =$ `length(p_off)` $- 1$.
(2) Constructs `p_on = chebfun(f,[a b],'splitting','on')`. This gives K local Chebyshev polynomials of degrees $D_k =$ `length(p_on.funs{k})` $- 1$.
(3) Computes the storage cost $B = 1 + 3K + \sum_{k=1}^{K}(D_k + 1)$, which we defer to why this for next file, where where $K = 1$ for the global case. Then, we select whichever representation yields the smaller B.

All computation in `Fapprox` is fp64. This produces a Chebfun object `p_best` that we then need to convert into a numerical vector.

3.2. **Encoding into $\mathbb{R}^B$: Rb_transform.m.** The file `Rb_transform.m` takes `p_best` and extracts: 1) the number of pieces K, 2) midpoints m_k and inverse half-lengths $h_k^{-1} = 2/(b_k - a_k)$ for each subinterval $[a_k, b_k]$, 3) degrees D_k, and 4) all Chebyshev coefficients $c_{k,0}, \ldots, c_{k,D_k}$. These are packed into a vector $\theta \in \mathbb{R}^B$ of size equal to our cost in the previous step, in the following order required by the runner `Feval.m`.

$$\theta = \big(K,\ m_1, \ldots, m_K,\ h_1^{-1}, \ldots, h_K^{-1},\ D_1, \ldots, D_K,\ \underbrace{c_{1,0}, \ldots, c_{1,D_1}}_{\text{piece 1}},\ \ldots,\ \underbrace{c_{K,0}, \ldots, c_{K,D_K}}_{\text{piece K}}\big).$$

All computation until here in `Rb_transform.m` is fp64, but then we use `theta = single(theta);` to work in fp32.

3.3. **fp32 evaluation: Feval.m.** We get $K =$ `int32(theta(1))`, and then read $m_1, \ldots, m_K$, $h_1^{-1}, \ldots, h_K^{-1}$, $D_1, \ldots, D_K$ as contiguous fp32 slices of θ, with the remaining tail of θ being the concatenation of all coefficients $\{c_{k,j}\}$.

Given x in fp32, to select the piece that it belongs to, we compute $t_k = (x - m_k)h_k^{-1}$ for $k = 1, \ldots, K$ in fp32, and select the first k with $|t_k| \le 1$ (if rounding causes no interval to match, we choose the piece with midpoint closest to x.) Then, the coefficients for this piece are located by summing the lengths of the previous blocks `offset` $= \sum_{j<\text{piece}}(D_j + 1)$ with `start_idx` = `offset` + 1 and `end_idx` = `start_idx` + D_{piece}. Once we have the coefficients, we use the usual Clenshaw recurrence evaluation, in which given $c_0, \ldots, c_D$ be the coefficients we have $p(t) = tb_1 - b_2 + c_0$ with $b_j = 2t\, b_{j+1} - b_{j+2} + c_j$ for $j = D, \ldots, 1$

[1]If we try `chebfun('abs(x-.3)',[-10,10], 'splitting','off')` Chebfun attempts a global polynomial and takes around a degree of 128 to resolve the function, which is not ideal. But with `chebfun('abs(x-5)',[-10,10], 'splitting','on')` Chebfun inserts a breakpoint at $x = 5$ (and other locations if needed) and returns a piecewise-linear function with degree 1 per linear segment. This shows that splitting is essential for piecewise-smooth functions such as ReLU.

However, even for smooth functions, splitting can activate unnecessarily when the domain is large. If we try `chebfun('sin(x)',[0 1000*pi],'splitting','on')`, Chebfun produces dozens of subintervals, each with a short Chebyshev polynomial with degrees around 70 on each piece, for a total of over 1800 coefficients). But with `chebfun('sin(x)',[0 1000*pi],'splitting','off')` Chebfun rattempts a global polynomial and takes around a degree of 1684 to resolve the function, which is a bit smaller. This is why we do not simply trust splitting, and why `Fapprox.m` evaluates both modes and selects the one minimizing our storage cost B

and $b_{D+2} = 0$ and $b_{D+1} = 0$. This gives a cost of $O(K + D_{\text{active}})$ where $1 \leq K < 5$ given the suggested smooth functions and $0 \leq D < 50$ is the degree of the selected piece.

4. Evaluation Results

The table bellow lists the encoded sizes B for some functions I tried on the intervals $[-5, 5]$ and $[-10, 10]$. All values remain small $10 \leq B \leq 80$, which is consistent with our design choices meant to encourage small degree for the Chebyshev approximations.

The figures show the fp32 absolute and relative evaluation errors of computed approximation, gotten by sampling 1000 points on the interval, and doing the evaluation for them. Since Chebfun constructs each approximation by controlling the uniform error $\|f - p\|_\infty$ to within the tolerance `eps`, the absolute errors are uniformly small on both intervals (typically 10^{-7} to 10^{-5} depending on the function and the interval size). Relative error can become large only where the function value itself is very close to zero (for example in the saturated tails of `tanh`, `sigmoid`, and `softplus`), which is expected and not meaningful for applications usually. The only exception to this is $\exp(x)$ which has noticeably larger absolute error on $[-10, 10]$, which is as expected because the polynomial degree required to resolve the rapid growth of $\exp(x)$ increases only linearly with the interval grow, and also because $\exp(x)$ becomes very large there, so fp32 roundoff dominates even though the underlying Chebyshev approximation is accurate.

Function	B **on** $[-5, 5]$	B **on** $[-10, 10]$
ReLU	10	10
GELU	39	65
$\exp(x)$	22	27
$\sin(x)$	22	32
$\tanh(x)$	72	79
sigmoid	40	72
softplus	35	59
swish	39	69

Table 1. Encoded vector length B for each test function on the intervals $[-5, 5]$ and $[-10, 10]$.

5. Rational approximation and representation

Rational functions can achieve a given accuracy with a lower degree than polynomials, especially for functions with non-analytic behavior, which could decrease B. However, in our case the functions are quite nice, so the improvement in degree is around of 20% based on a couple of runs I did! However, a rational representation would require storing both numerator and denominator coefficients, which increases the total parameter count B. We can try some inequalities with our empirical bound on the imporvement and see we don't gain much in terms of reducing B.

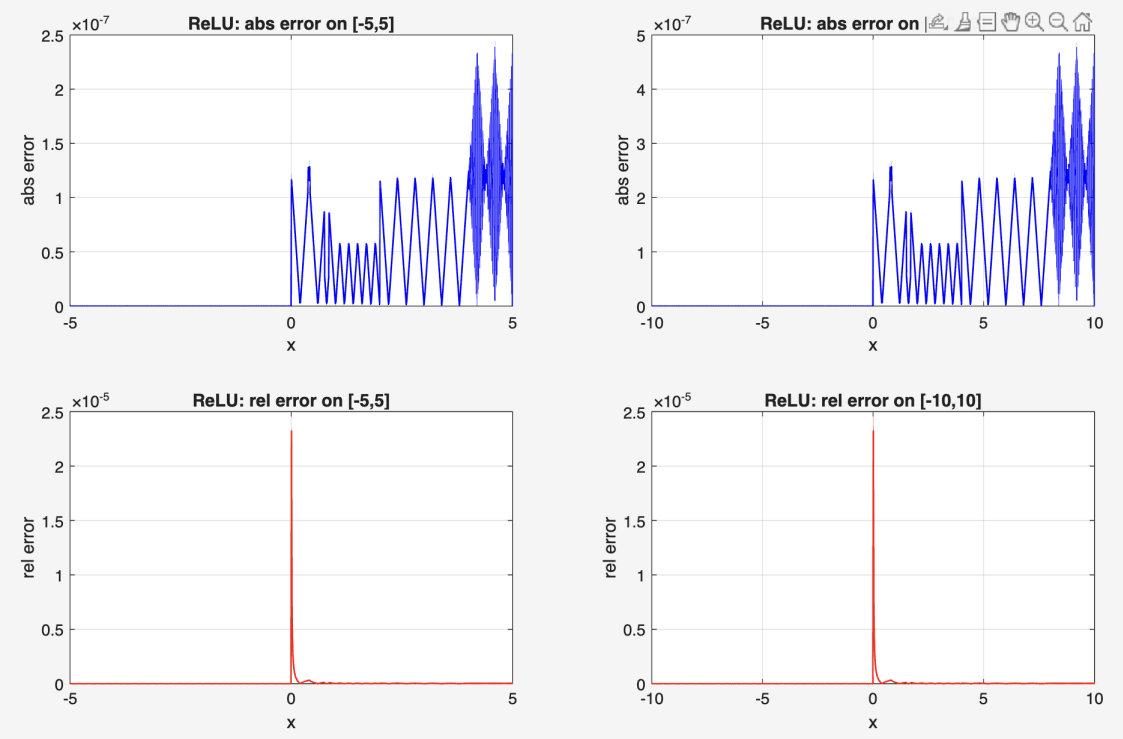

FIGURE 1. FP32 absolute and relative evaluation errors for ReLU.

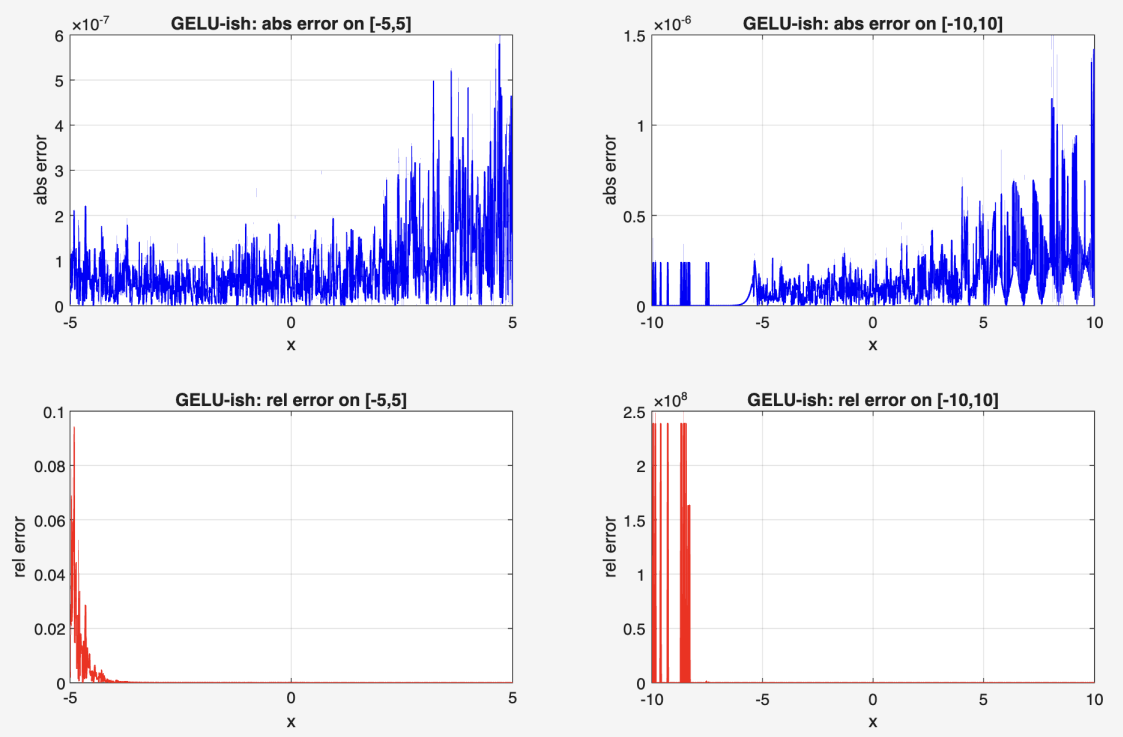

FIGURE 2. FP32 absolute and relative evaluation errors for GeLU.

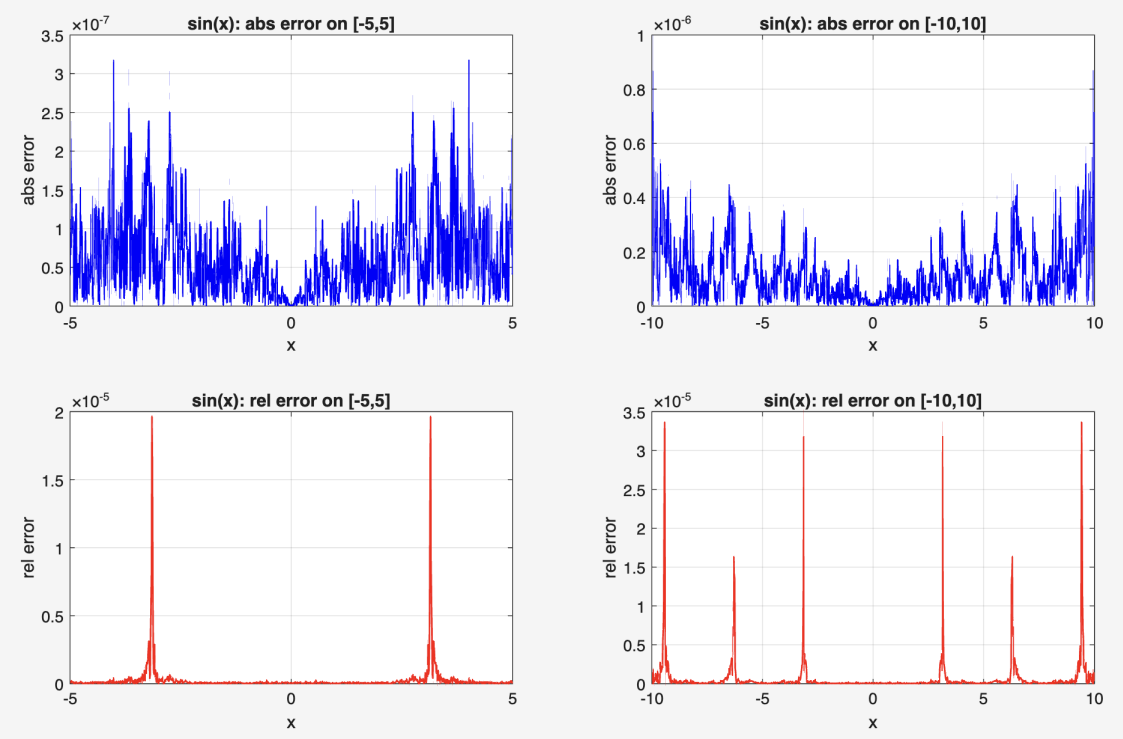

FIGURE 3. FP32 absolute and relative evaluation errors for sin(x).

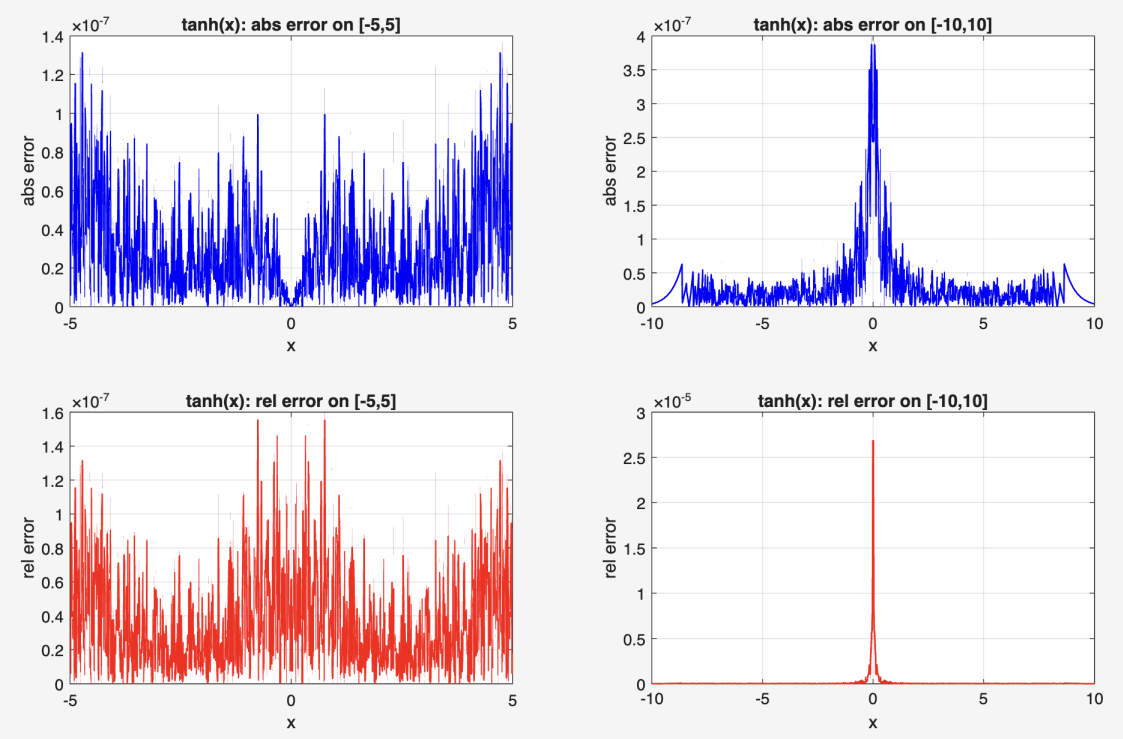

FIGURE 4. FP32 absolute and relative evaluation errors for tanh(x).

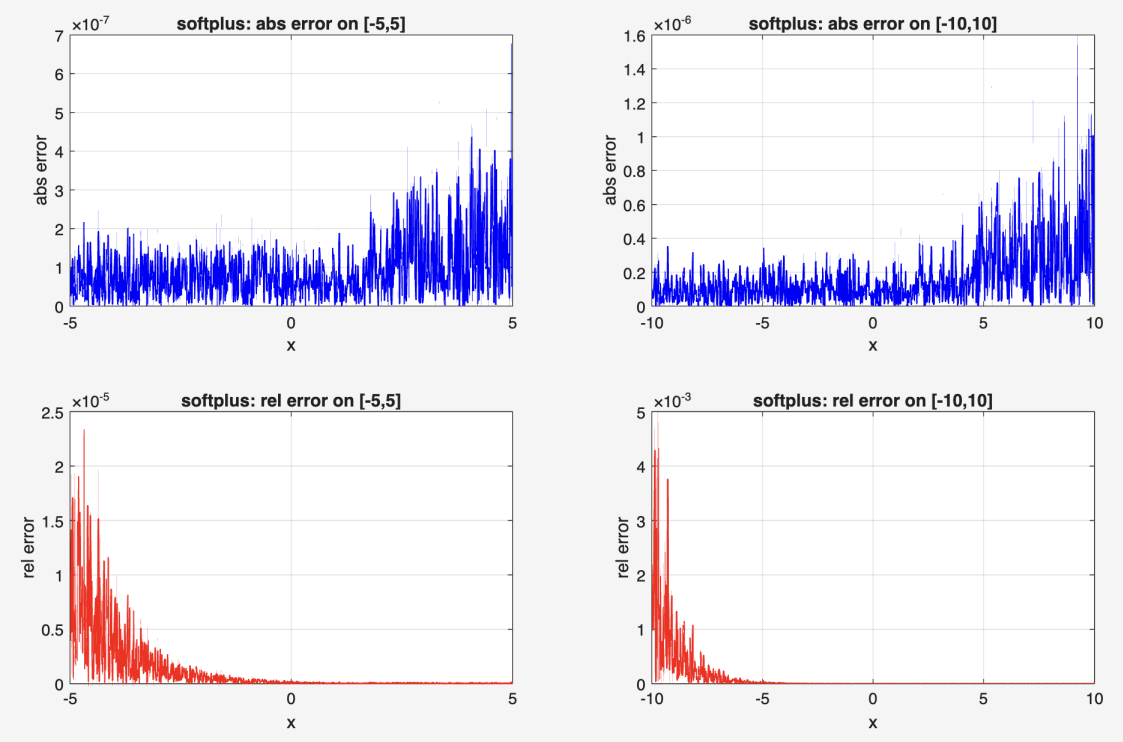

FIGURE 5. FP32 absolute and relative evaluation errors for softplus.

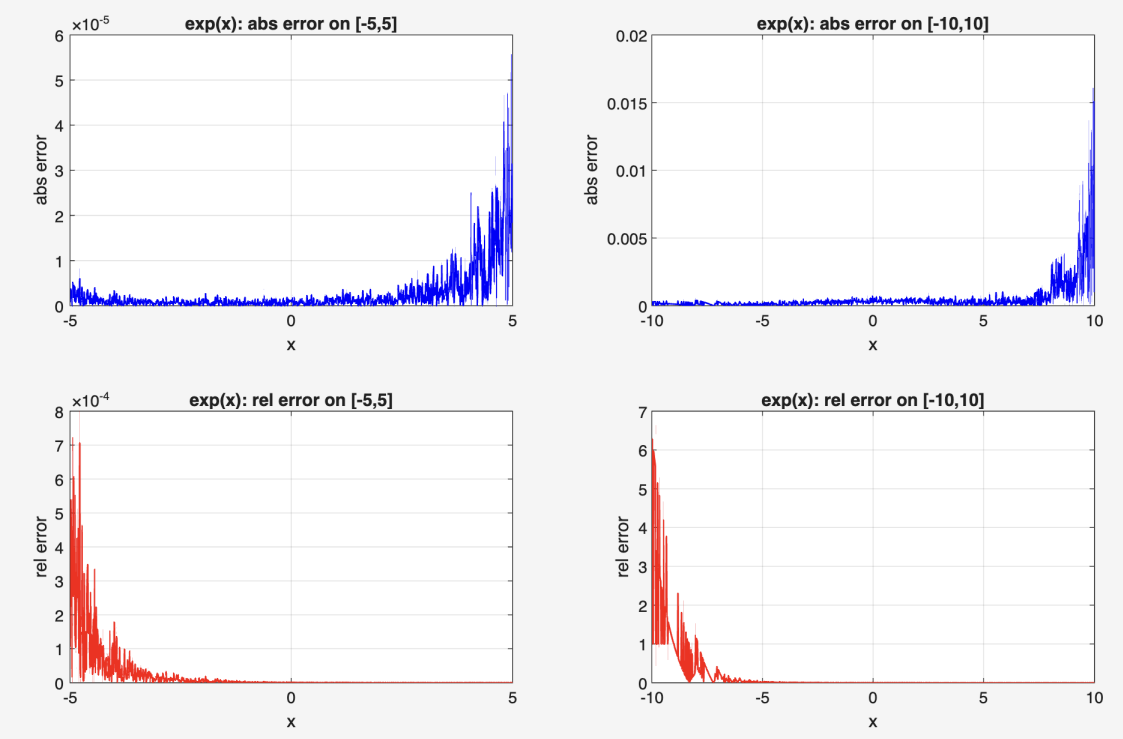

FIGURE 6. FP32 absolute and relative evaluation errors for exp(x).

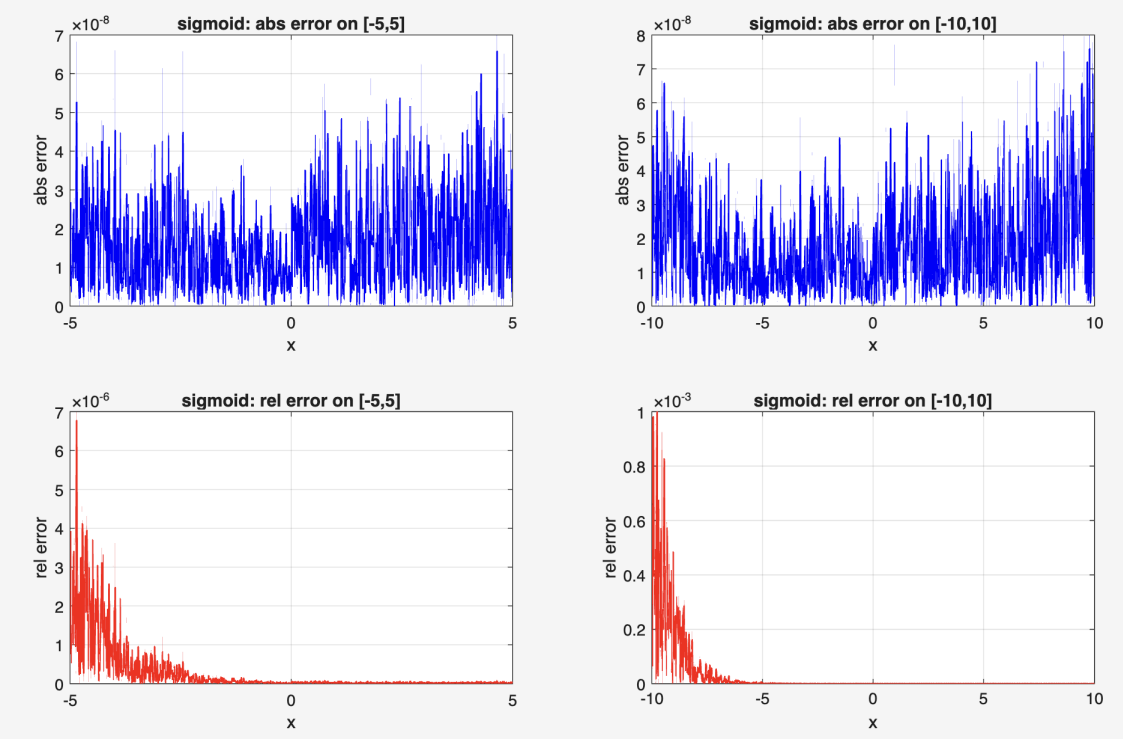

FIGURE 7. FP32 absolute and relative evaluation errors for sigmoid.

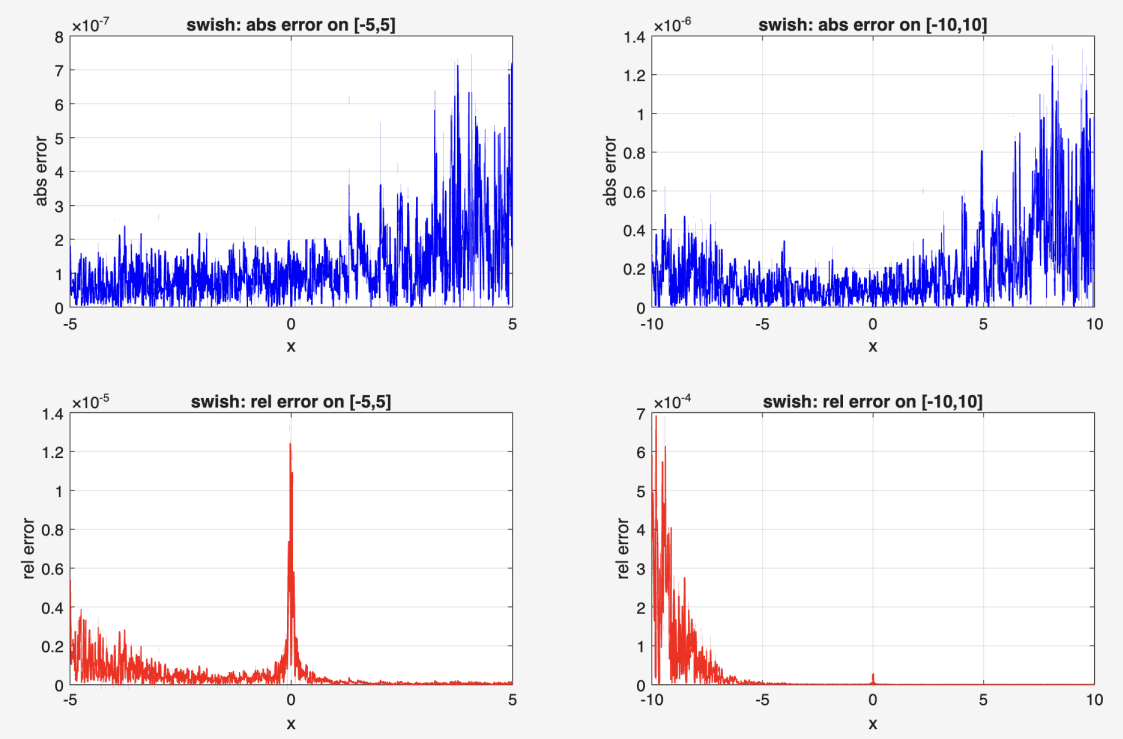

FIGURE 8. FP32 absolute and relative evaluation errors for swish.