Software-based Automatic Differentiation is Flawed

Daniel Johnson   **Trevor Maxfield**   **Yongxu Jin**   **Ronald Fedkiw**
Stanford University
{dansj,maxfit,yxjin,rfedkiw}@stanford.edu

Abstract

Various software efforts embrace the idea that object oriented programming enables a convenient implementation of the chain rule, facilitating so-called automatic differentiation via backpropagation. Such frameworks have no mechanism for simplifying the expressions (obtained via the chain rule) before evaluating them. As we illustrate below, the resulting errors tend to be unbounded.

1   Introduction

Basic calculus texts introduce students to many of the rules of differentiation including the product rule, quotient rule, chain rule, etc. Various large-scale software packages (e.g., PyTorch [1], TensorFlow [2], Torch [3], Caffe [4], Theano [5], Jax [6]) utilize the various rules of differentiation, leveraging the idea of object oriented programming in order to elegantly implement the chain rule. Each implemented function is paired with code that computes its derivative, and branches are created to connect functions that call other functions. Using the values obtained in a so-called forward pass through the code, the computational graph is evaluated by following the branches in order to obtain values for various derivatives. As is taught (as early as the discussion on limits) in various calculus texts, one ought to simplify expressions before evaluating them. This is even more important when considering how the rules of differentiation (especially the chain rule) increase the number of expressions. There seems to be no good remedy for expression simplification in the popular software packages.

2   A Trivial Example

Suppose one simple function is $f(x) = x^2 - 4$ and another is $g(x) = x - 2$; then $h(x) = \frac{f(x)}{g(x)}$ has $h(2) = $ NaN on an AMD Ryzen 5 5600X 6-Core Processor, Apple M1 Pro, etc. (any system correctly implementing IEEE 754 [7]). Figures 1 and 2 show $h(x)$ near $x = 2$.

Remark 2.1 *Replacing the numerator of h with $x^3 - 8$ causes it to underflow in a wider interval. The higher the powers of x, the larger the interval of underflow. The width of the domain where the denominator leads to NaNs can likewise be increased.*

A common strategy for removing the NaNs (and overflow) is to bound the denominator away from zero. A typical way to accomplish this is to define

$$h_1(x; \epsilon, \delta) = \begin{cases} \frac{f(x)}{g(x) + S(g(x))\epsilon} & \text{if } 2 - \delta < x < 2 + \delta \\ \frac{f(x)}{g(x)} & \text{otherwise} \end{cases} \tag{1}$$

where S is the sign function and $S(0)$ is chosen arbitrarily. Here, $\epsilon > 0$ and $\delta > 0$ are constants. Although $h_1(x; \epsilon, \delta)$ is necessarily discontinuous, the discontinuity can be somewhat hidden by choosing δ large enough and ϵ small enough such that $\epsilon \ll |g(2 \pm \delta)|$. See Figures 3 and 4. The $O(1)$ errors shown in Figures 3 and 4 are well known in numerical analysis communities (see

e.g., [8, 9, 10, 11, 12]) and typically addressed by refactoring code so that h does not utilize two separate functions; instead, h would be implemented as $\hat{h}(x) = x + 2$.

If ϵ is chosen large enough and δ is chosen small enough, then $g(x)$ can be ignored to obtain

$$\hat{h}_1(x; \epsilon, \delta) = \begin{cases} \frac{f(x)}{S(g(x))\epsilon} & \text{if } 2 - \delta < x < 2 + \delta \\ \frac{f(x)}{g(x)} & \text{otherwise} \end{cases} \tag{2}$$

for simplicity. In this case, the discontinuity can be removed by choosing δ so that $g(2 \pm \delta) = S(g(2 \pm \delta))\epsilon$. This continuous case can be written with $g(x)$ in the conditional as

$$h_2(x; \epsilon) = \begin{cases} \frac{f(x)}{S(g(x))\epsilon} & \text{if } |g(x)| < \epsilon \\ \frac{f(x)}{g(x)} & \text{otherwise} \end{cases} \tag{3}$$

removing δ; moreover, one can write

$$\hat{h}_2(x; \epsilon) = \begin{cases} \frac{f(x)}{-\epsilon} & \text{if } -\epsilon < g(x) < 0 \\ \frac{f(x)}{\epsilon} & \text{if } 0 < g(x) < \epsilon \\ \frac{f(x)}{g(x)} & \text{otherwise} \end{cases} \tag{4}$$

in order to remove the sign function (although an arbitrary choice is still required when $g(x) = 0$). See Figure 5.

3 Computing Derivatives

Analytically, $f'(x) = 2x$, $g'(x) = 1$, and $h'(x) = 1$ everywhere (after removing the singularity to reduce $h(x)$ to $x + 2$). Backpropagation will utilize

$$dh = \left(\frac{\partial h}{\partial f} \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \frac{\partial g}{\partial x} \right) dx \tag{5a}$$

$$= \left(\frac{1}{g(x)} \frac{\partial f}{\partial x} - \frac{f(x)}{g(x)^2} \frac{\partial g}{\partial x} \right) dx \tag{5b}$$

$$= \left(\frac{2x}{x - 2} - \frac{x^2 - 4}{(x - 2)^2} \right) dx \tag{5c}$$

to compute derivatives. When $2 + \gamma$ underflows to be 2, $h'(2 + \gamma) \approx \frac{4}{0} - \frac{0}{0}$ which gives NaNs; otherwise,

$$h'(2 + \gamma) = \frac{4 + 2\gamma}{\gamma} - \frac{(4 + 4\gamma + \gamma^2) - 4}{\gamma^2} \tag{6}$$

can lead to

$$h'(2 + \gamma) \approx \frac{4 + 2\gamma}{\gamma} - \frac{(4 + 4\gamma) - 4}{\gamma^2} = 2 \neq 1 \tag{7}$$

when $4 + 4\gamma + \gamma^2$ underflows to be $4 + 4\gamma$ because $\gamma^2 \ll 4$. Alternatively, the quotient rule would combine equation 6 into a single fraction, which has no bearing on the underflow and thus also leads to the final result in equation 7. See Figure 6 for the results obtained using PyTorch (identical results were obtained using Tensorflow).

To analyze the derivative of $h_1(x)$, one can rewrite equation 5b as,

$$dh_1 = \left(\frac{2x}{g_1(x)} - \frac{x^2 - 4}{g_1(x)^2} g_1'(x) \right) dx \tag{8}$$

with

$$g_1(x; \epsilon, \delta) = \begin{cases} g(x) + S(g(x))\epsilon & \text{if } 2 - \delta < x < 2 + \delta \\ g(x) & \text{otherwise} \end{cases} \tag{9}$$

since the numerator remains unchanged. Since S' is identically zero almost everywhere, $g_1'(x) = g'(x) = 1$. Note that attempts to smooth S lead to S taking on values of zero when g is zero, defeating the purpose of adding $S(g(x))\epsilon$ to the denominator.

2

Outside of $x \in (2 - \delta, 2 + \delta)$, the behavior of h_1 matches the behavior of h; thus, we assume that δ is chosen large enough to encompass all of the degenerate behavior with the perturbed denominator. When $2 + \gamma$ underflows to be 2, $h'_1(2 + \gamma) \approx \frac{4}{S(0)\epsilon} - \frac{0}{\epsilon^2}$; otherwise,

$$h'_1(2 + \gamma) \approx \frac{4 + 2\gamma}{\gamma + S(\gamma)\epsilon} - \frac{(4 + 4\gamma) - 4}{(\gamma + S(\gamma)\epsilon)^2} \tag{10a}$$

$$\approx \frac{2\gamma^2 + (4 + 2\gamma)S(\gamma)\epsilon}{(\gamma + S(\gamma)\epsilon)^2} \tag{10b}$$

when $4 + 4\gamma + \gamma^2$ underflows to be $4 + 4\gamma$. Note that $h'_1(2 + \gamma) \to 2$ as $\epsilon \to 0$, consistent with equation 7 as expected; however, $h'_1(2 + \gamma) \to \frac{4}{S(\gamma)\epsilon}$ as $\gamma \to 0$. See Figure 7 for the results obtained using PyTorch (identical results were obtained using Tensorflow).

Remark 3.1 *Although the typical approach of adding a small number to the denominator in order to avoid* `NaNs` *and overflow leads to $O(1)$ errors (e.g. $h_1(2) = 0$ when $h(2) = 4$), backpropagation through the perturbed formulas leads to unbounded errors (e.g. $h'_1(2) = \frac{4}{S(0)\epsilon}$ when $h'(2) = 1$). Unfortunately, practitioners typically choose ϵ to be as small as possible.*

To analyze the derivative of $h_2(x)$, one can replace g_1 with

$$g_2(x; \epsilon) = \begin{cases} S(g(x))\epsilon & \text{if } |g(x)| < \epsilon \\ g(x) & \text{otherwise} \end{cases} \tag{11}$$

in equation 8. In the region of interest where $|g(x)| < \epsilon$, $g'_2(x) = 0$. When $2 + \gamma$ underflows to be 2, $h'_2(2 + \gamma) \approx \frac{4}{S(0)\epsilon} - \frac{0}{\epsilon^2} \cdot 0$; otherwise,

$$h'_2(2 + \gamma) = \frac{4 + 2\gamma}{S(\gamma)\epsilon} \tag{12}$$

whenever $|g(x)| < \epsilon$. Note that any underflow in f is unimportant since the result is robustly divided by ϵ^2 before being multiplied by 0. Once again, $h'_2(2 + \gamma) \to \frac{4}{S(\gamma)\epsilon}$ as $\gamma \to 0$; however, it happens more quickly in h'_2 (than in h'_1) because the second term in equation 8 is no longer present to partially cancel the first term (as it does in equation 10a). See Figure 8 for the results obtained using PyTorch (identical results were obtained using Tensorflow).

4 Conclusion

As we have shown, the basic software infrastructures make the false assumption that expressions can be evaluated in nested fashion using the chain rule when the correct order of operations requires a cancelling of expressions before evaluating (as is taught in basic calculus when students learn about limits). Although practitioners typically implement hacks into their codes in order to avoid division by zero, this leads to well-known $O(1)$ errors in the perturbed functions; unfortunately, the derivatives of those functions can have errors that are unbounded. The community does not seem to be concerned with these difficulties; rather, they over-embrace the modularity of the various software frameworks believing in the correctness of the derivatives that they produce.

Although this paper addressed the problems occurring with the very simple case of division, the lack of the ability to simplify expressions in object oriented code also causes problems with multiplication (e.g. when $0 \cdot \infty$ should be finite), addition/subtraction (e.g. when $\infty - \infty$ should be finite), and many other operations.

Assuming that one takes precautions to avoid `NaNs`, it is important to consider what an erroneously large derivative (that blows up) will do when solving inverse/control problems or training neural networks. A large value of the derivative in one entry of a gradient causes all of the other entries to vanish (comparatively), making the gradient erroneously point in the direction of the large derivative. This leads to poor search directions in the optimization for first order methods, and (perhaps even more devastatingly) a corrupted Hessian for second order methods. In turn, this undoes all of the hard work put into designing loss functions, regularizers, etc., in hopes of finding good parameters for a neural network.

5 Acknowledgments

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[5] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv*, pages arXiv–1605, 2016.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

[7] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[8] Michael T Heath. *Scientific Computing: An Introductory Survey, Revised Second Edition*. SIAM, 2018.

[9] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[10] Richard L Burden, J Douglas Faires, and Annette M Burden. *Numerical analysis*. Cengage learning, 2015.

[11] E Ward Cheney and David R Kincaid. *Numerical mathematics and computing*. Cengage Learning, 2012.

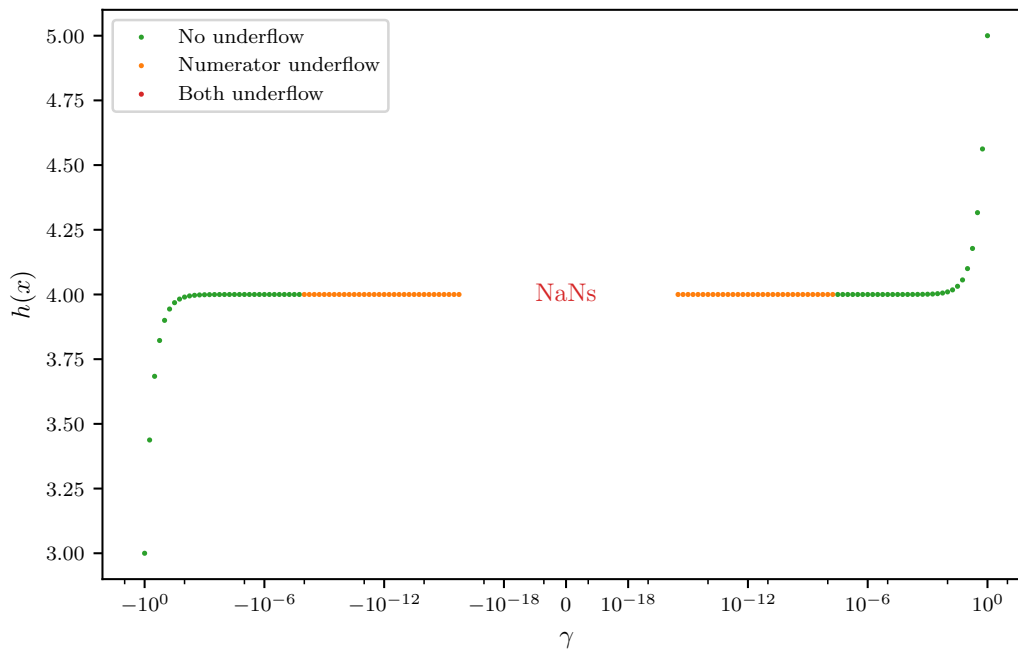[12] Kendall Atkinson. *An introduction to numerical analysis*. John wiley & sons, 1991.

Figure 1: To understand the behavior of $h(x)$ near $x \approx 2$, we choose γ with $\gamma \approx 0$ and evaluate $x = 2 + \gamma$ before subsequently evaluating $h(x)$. The plot shows values obtained for $h(x)$ with γ plotted on a log scale. The green points are consistent with the straight line $h(2 + \gamma) = \gamma + 4$ as expected. The yellow points indicate values of γ where the numerator underflows, i.e. where $x^2 = 4 + 4\gamma + \gamma^2$ evaluates to $4 + 4\gamma$ because $\gamma^2 \ll 4$ using double precision. With a numerator of $4 + 4\gamma - 4 = 4\gamma$ and a denominator of $2 + \gamma - 2 = \gamma$, the yellow points are consistent with $h(2 + \gamma) = 4$. The empty space near $\gamma = 0$ is where the denominator also underflows, i.e. when $x = 2 + \gamma$ evaluates to 2 because $\gamma \ll 2$ using double precision. These values of γ produce `NaNs`. See also Figure 2.
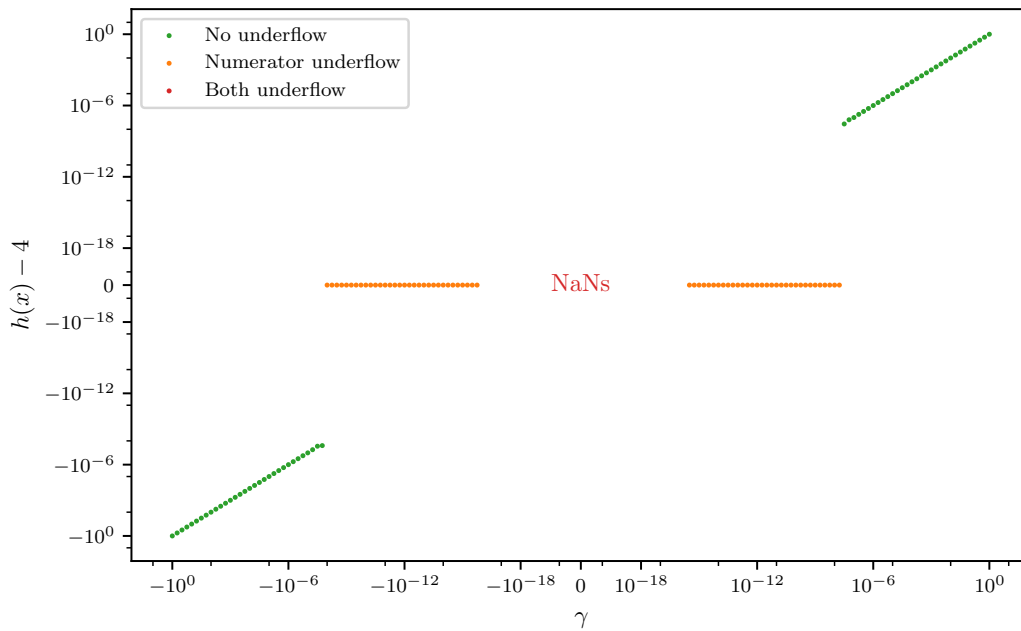
5

Figure 2: Same data as in Figure 1, except that the leading digit of 4 is removed from all of the data. Plotting the dependent axis on a log scale elucidates the linearity in the green region and the incorrect flatness in the yellow region. In addition, note the small errors in the linearity of the green region near the boundary with the yellow region. These errors are caused by $x^2 = 4 + 4\gamma + \gamma^2$ evaluating to $4 + 4\gamma + \alpha\gamma^2$ as information in $\gamma^2 \ll 4$ is partially lost as it begins to underflow. Note that $0 < \alpha < 1$ is what one would expect when some of γ^2 is truncated; however, α can be larger than 1 when rounding up before truncating. Generally, one would expect $1 < \alpha < 2$ when rounding up, since the largest increase occurs when 2^{-p} is rounded up to 2^{-p+1}.
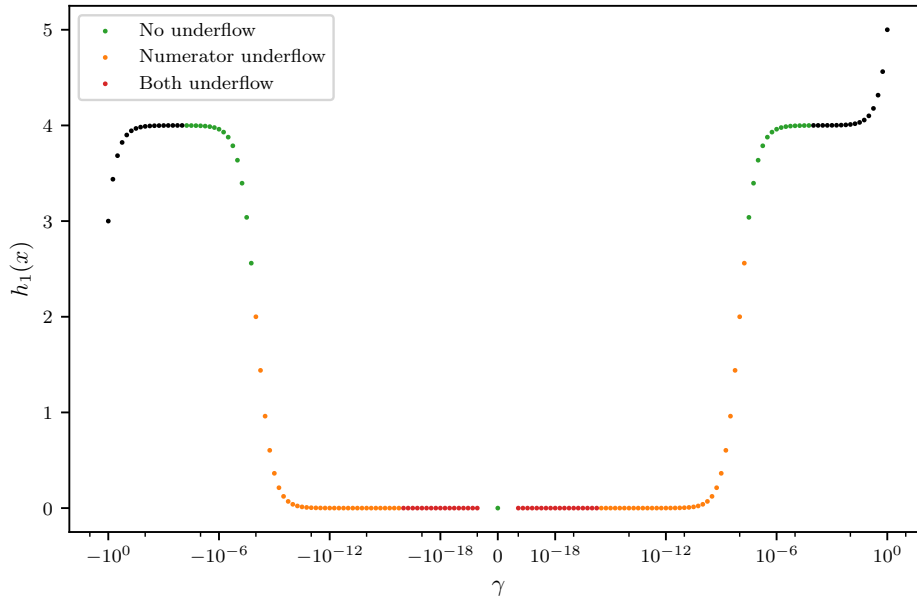
Figure 3: Plot of $h_1(2 + \gamma)$ with $\epsilon = 10^{-8}$ and $\delta = 10^{-4}$, where δ is chosen large enough to put the discontinuity in the green region, well-separated from the issues with underflow we wish to address. The black points indicate data where h_1 is identical to h. Although the perturbed denominator removes the NaNs in the red region, h_1 is identically zero there and thus inconsistent with $x + 2$; moreover, the yellow and green points are also incorrectly perturbed away from $x + 2$ towards zero. These $O(1)$ errors are difficult to avoid without refactoring the code. Note that the $\gamma = 0$ data point is colored green because $2 + \gamma$ does not technically underflow (even though $2 + \gamma$ evaluates to 2, and thus $\gamma = 0$ behaves similarly to the red points); in addition, the gap around $\gamma = 0$ is due to the minimum $|\gamma|$ being 10^{-20} (besides $\gamma = 0$). See also Figure 4.
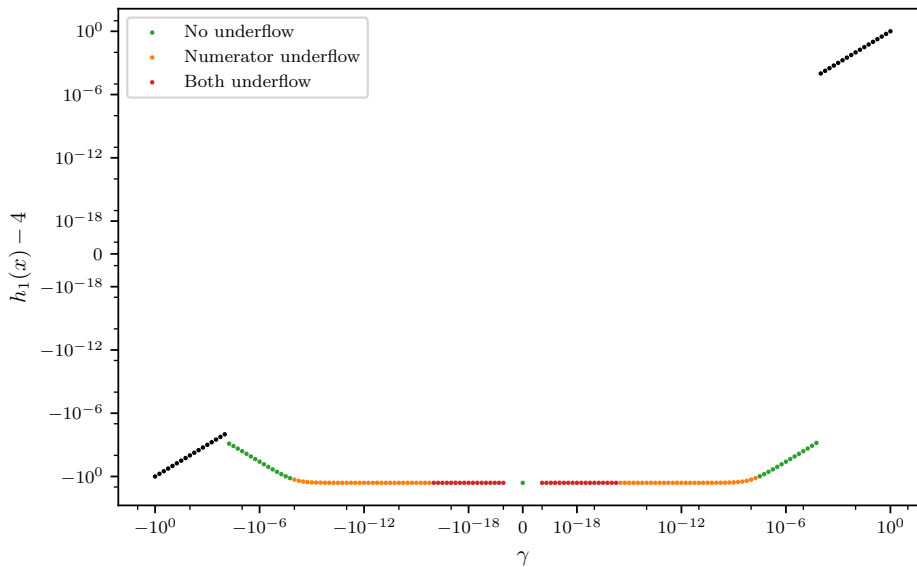

Figure 4: Same data as Figure 3, except that the leading digit of 4 is removed and the dependent axis uses a log scale. This elucidates the linearity in the black region and the discontinuity between the black and green regions.
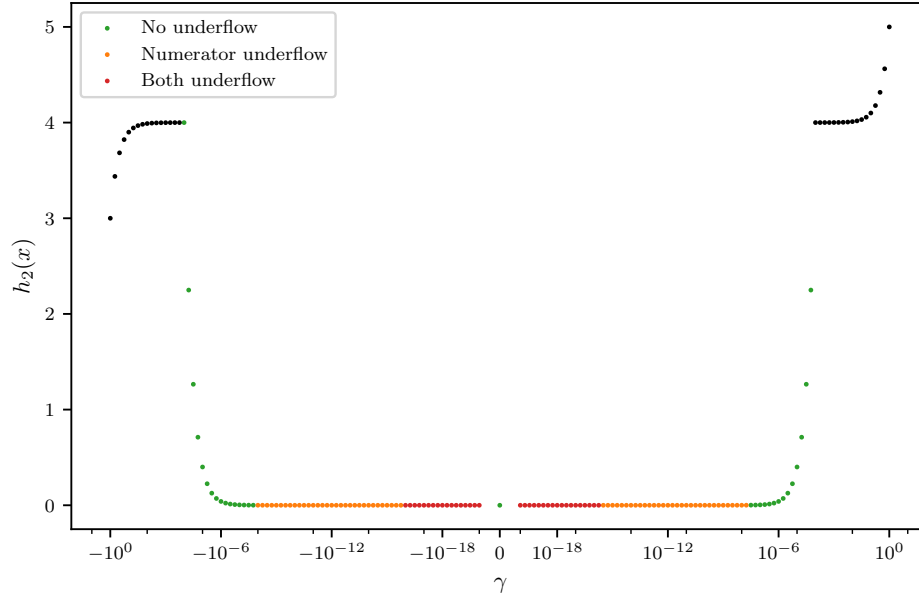
Figure 5: Plot of $h_2(2 + \gamma)$ with $\epsilon = 10^{-4}$; in other words, a plot of $\hat{h}_1(2 + \gamma)$ with $\epsilon = 10^{-4}$ chosen to match $\delta = 10^{-4}$ for continuity. Unlike in Figure 3 where $g(x)$ slowly vanishes providing for a gradual transition between the black and green regions, the transition is more abrupt here even though the function is continuous; however, a more fair comparison would use $\epsilon = 10^{-8}$ in this figure, although that would place the transition in the yellow region (which we would like to avoid for the sake of exposition).
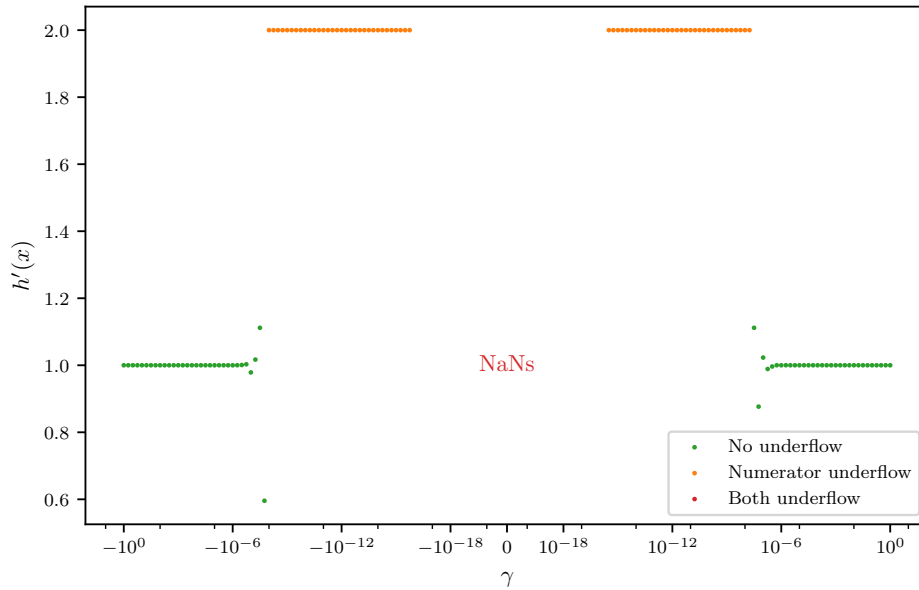

Figure 6: Although $h'(x) = 1$ identically, a value of 2 is obtained when $4 + 4\gamma + \gamma^2$ underflows to be $4 + 4\gamma$ because $\gamma^2 \ll 4$. In addition, the errors near the boundary of the green region are caused by $4 + 4\gamma + \gamma^2$ partially underflowing to $4 + 4\gamma + \alpha\gamma^2$; as can be seen in equation 6, this leads to a derivative that looks like $2 - \alpha$. Recall (from Figure 2) that one would expect $0 < \alpha < 1$ for rounding down and $1 < \alpha < 2$ for rounding up.
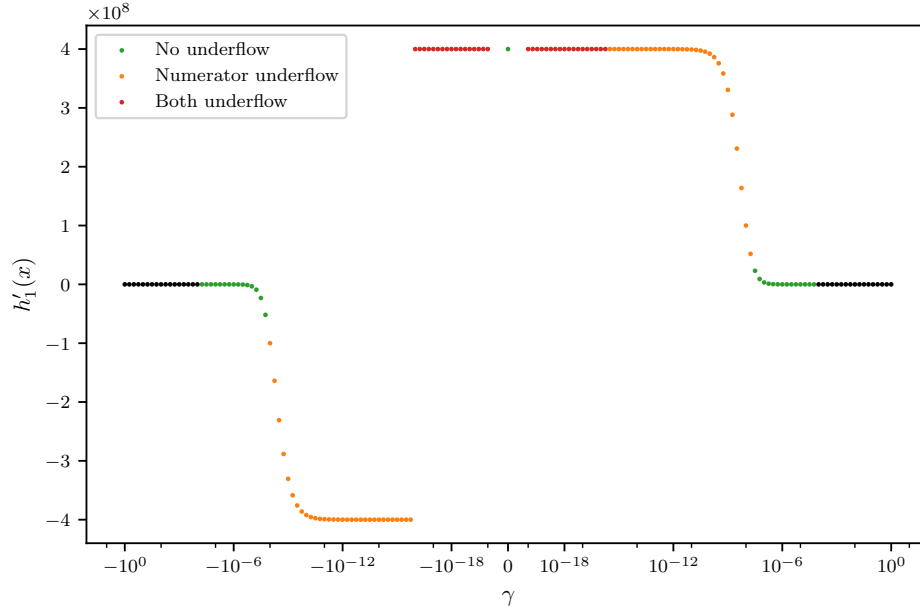
Figure 7: As discussed before and after equation 10, h_1' has spurious derivative values on the order of $\frac{4}{\epsilon}$ which is 4×10^8 for $\epsilon = 10^{-8}$. Note that $S(g(2 + \gamma)) = S(0)$ in the red region where $2 + \gamma$ underflows to 2. This means that all of the red points will agree in sign based on the arbitrary choice of $S(0)$; obviously, we chose $S(0) = 1$.
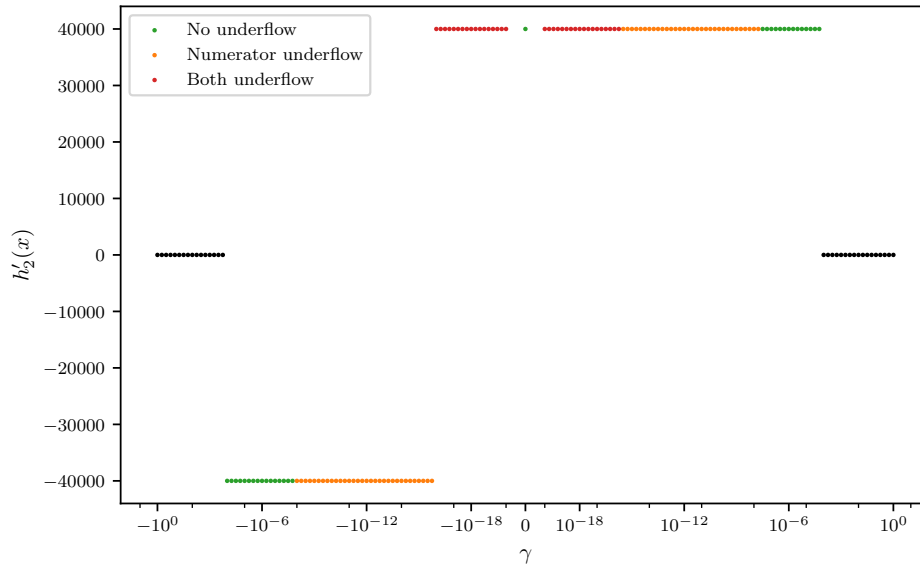

Figure 8: As discussed after equation 12, h_2' has spurious derivative values on the order of $\frac{4}{\epsilon}$ which is 4×10^4 for $\epsilon = 10^{-4}$; moreover, the derivatives blow up immediately (instead of gradually, as in Figure 7).