Zero Grads: Supplemental Materials

MICHAEL FISCHER, University College London, United Kingdom
TOBIAS RITSCHEL, University College London, United Kingdom

ACM Reference Format:
Michael Fischer and Tobias Ritschel. 2024. Zero Grads: Supplemental Materials. *ACM Trans. Graph.* 43, 4, Article 49 (July 2024), 3 pages. https://doi.org/10.1145/3658173

This supplementary contains additional information on our surrogate implementation and hyperparameters (Sec. 1), rendering setups (Sec. 2.1), and detailed descriptions of the tasks we solve (Sec. 2.2).

1 IMPLEMENTATION DETAILS

We implement all our experiments in PyTorch [Paszke et al. 2017]. The proxy powering our surrogate is implemented as a multi-layered perceptron (MLP) and activated by a leaky ReLU. We randomly initialize our Neural Proxy for each optimization run (via the standard PyTorch initialization, for the quadratic proxy, we choose the identity matrix) and optimize its weights alongside the parameter with a separate Adam optimizer. We perform three update steps on the surrogate parameters ϕ per optimization iteration in order to improve the surrogate's fit to the sampled data. This is simple autodiff-driven gradient descent (GD) and hence very fast. Note that no new data is sampled between these update steps, they merely serve to improve the surrogate fit and do not increase the required computational budget. For all gradient updates, we use the Adam optimizer with standard parameters and learning rates as specified in Tab. 1. We additionally experimented with different sampling patterns and found both both low-discrepancy (Sobol) and antithetic samples and found both to improve performance, and adapt antithetic samples for simplicity. We normalize the network's inputs to [0,1]. For the lower-dimensional tasks ($n_{\text{dim}} < 50$), it suffices to use 3 hidden layers with 64 neurons each, whereas for the higher-dimensional tasks (below the horizontal line in Tab. 1), we found that we needed to increase the surrogate's capacity to 8 layers à 128 neurons and additionally use positional encoding to increase the frequencies that the network can encode.

1.1 Hyperparameters

Our method comes with two hyperparameters: the number of samples N we use to estimate our surrogate's gradient with (cf. Alg.2 in the main text), and the spread of the locality kernel λ, which will influence how far these samples are spaced out around the current parameter θ.

Authors' addresses: Michael Fischer, University College London, United Kingdom, m.fischer@cs.ucl.ac.uk; Tobias Ritschel, University College London, United Kingdom, t.ritschel@ucl.ac.uk.

0730-0301/2024/7-ART49
https://doi.org/10.1145/3658173

Table 1. Our hyperparameters σ_o and N, as well as the experiment settings for the different tasks, sorted by dimensionality in ascending order. MPL is short for matplotlib.

	σ_o	N	n_{dim}	LR θ	LR ϕ	Renderer
Wicker	0.33	2	3	1×10^{-3}	1×10^{-3}	Blender
BRDF	0.33	2	4	1×10^{-3}	1×10^{-3}	Mitsuba
CBox	0.10	2	4	5×10^{-4}	1×10^{-3}	Mitsuba
Gravity	0.20	2	5	1×10^{-3}	1×10^{-3}	Blender
Rocket	0.33	2	10	1×10^{-3}	5×10^{-4}	MPL
NodeGr.	0.20	2	24	1×10^{-3}	1×10^{-3}	Blender
LED	0.33	2	336	1×10^{-3}	1×10^{-3}	Blender
Mosaic	0.025	16	320	5×10^{-4}	1×10^{-3}	Blender
Caustic	0.013	20	1,024	2×10^{-4}	1×10^{-4}	PyTorch
Mesh	0.025	20	7,686	2×10^{-3}	1×10^{-4}	NVDiff.
Spline Gen.	0.025	20	8,764	1×10^{-5}	1×10^{-4}	MPL
MLP	0.025	20	35,152	1×10^{-4}	1×10^{-4}	MPL
Texture	0.025	20	196,608	1×10^{-5}	1×10^{-4}	MPL

We specify the number of samples N we use for estimating the surrogate's gradients in Tab. 1. For the lower-dimensional tasks, it suffices to use $N = 2$, whereas for the higher-dimensional tasks, the noise and higher variance from this rough gradient estimate impede convergence and thus require higher sample counts. We would like to emphasize that those are still far lower than what competing methods use, e.g., $2n_{\text{dim}}$ for finite differences (FD) or $m \times n_{\text{dim}}, m \gg 2$, for directional Gaussian smoothing (DGS) [Zhang et al. 2020]. Our method also benefits from more samples in the lower-dimensional regime, but these come at the cost of increased compute, which is why we tried to achieve a minimal number to keep the overhead low.

We show a comparison of different sample counts on the **Mesh** and **MLP** tasks in Fig. 3 and detail the remaining hyperparameters and experiment settings in Tab. 1, where σ_o denotes the spread of the locality kernel λ. As a general rule of thumb, we recommend setting the initial σ_o to 0.33 on normalized domains and finetune from there, if necessary. For higher-dimensional, interlinked problems, we have found a more fine-granular sampling to be necessary and use $\sigma_o = 0.025$. We use 15% of the locality spread as the spread of the smoothing kernel κ.

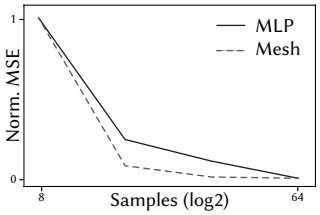

Fig. 3. Final error vs. samplecount N.

2 TASKS

This section provides information on the task setup, problems, goals and rendering architectures used.

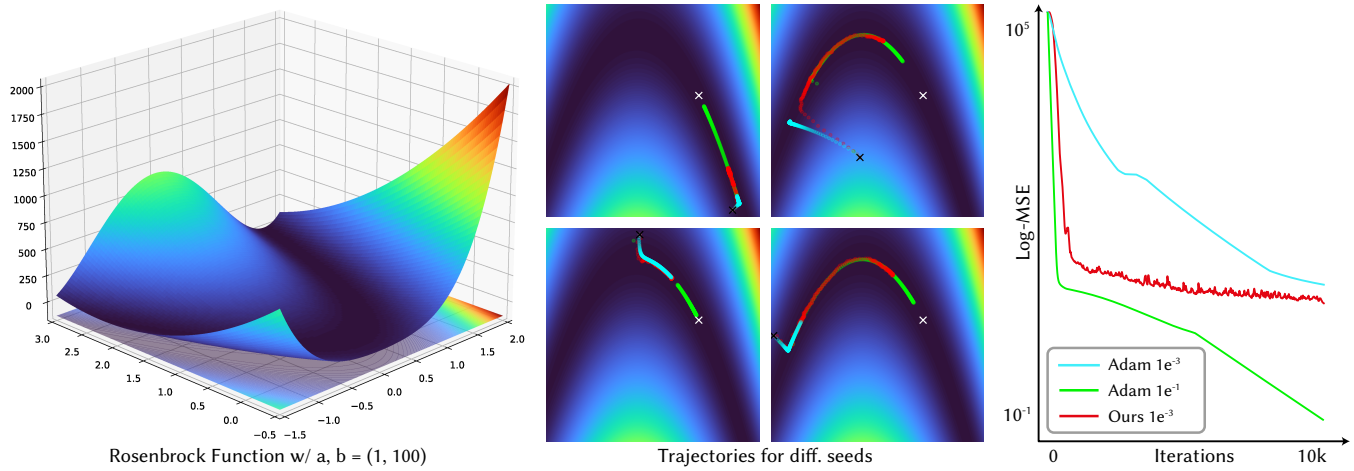

Fig. 1. We evaluate our method on the Rosenbrock function against gradient descent with analytical gradients and Adam with equal learning rate, sample count and iterations. Similar to Adam, our method struggles to make progress in the valleys of low slope, a common limitation of gradient-based techniques. Adam, with a higher learning rate, converges faster than our method. The convergence plots in the right subfigure are median values over an ensemble of 10 independent runs and seeds.

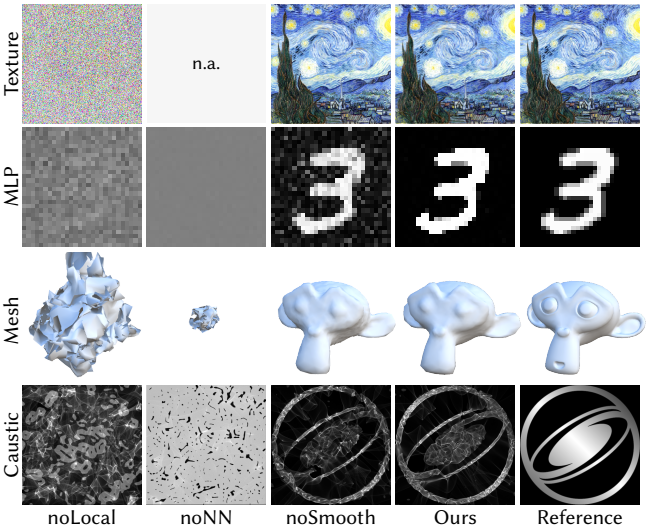

Fig. 2. We show the results of our ablated methods from the main manuscript (Sec. 4.1) on the higher-dimensional tasks. Similar to CMA, the result for the quadratic proxy (noNN) could not be run due to the quadratic memory complexity.

2.1 Rendering

To render the images for the tasks Mosaic, Wicker, LED, NodeGraph and Gravity, we interface our method with Blender via an efficient socket-based local TCP network, which enables us to make use of Blender's rendering engines and the embedded physics solver. All images were set to render noise-free under either Eevee or Cycles, with 16 to 128 samples and denoising activated. For the tasks BRDF and Cornell-Box, and the comparisons with Mitsuba, we use Mitsuba 3 [Jakob et al. 2022] with the path-replay backpropagation integrator at 16spp. For the Mesh task, we use NVDiffRast [Laine et al. 2020] with standard hyperparameters. For the remaining tasks Rocket, Spline Generation and Texture, we use a custom matplotlib-based renderer [Hunter 2007]. Note that none of this interfacing is necessary for our method to work, but pure convenience for rapid prototyping and reducing I/O times from and to disk. Most importantly, we do not propagate any gradient information through the rendering process, even if this were possible, e.g., when using a differentiable renderer. One could alternatively render an image, save it to disk and manually load it and perform a gradient update step, which would yield the same results, but be arguably less convenient.

2.2 Task Descriptions

2.2.1 Higher Dimensions. While some of our higher-dimensional example tasks could in theory also be solved via established, specialized methods (e.g., [Holl et al. 2020; Jakob et al. 2022; Nicolet et al. 2021]), they show that our method scales well to higher dimensional problems and reinforce our argument of general applicability.

All comparisons to the following optimization algorithms are performed under the same budget of function evaluations. For the comparisons with genetic algorithms (GAs), we use the publicly available Python package pygad [Gad 2021]. For simulated annealing (SA) [Xiang et al. 2013], we use the `scipy` library [Virtanen et al. 2020]. For simultaneous perturbation stochastic approximation (SPSA) [Spall 1992], we use the publicly available `spsa` package [Nguyen 2022]. Note that, while we use standard hyperparameters for the other packages, we here adapted the SPSA perturbation radius to the sampling radius used by our method in order to enable a fair comparison (the default value of 2.0 is too large for many of our problems, e.g., for the delicate task of network training).

Texture For the **Texture** task, we use our method to optimize the 256 pixels of an image texture, leading to a $256 \times 256 \times 3 = 196{,}608$ optimization problem. We randomly initialize the texels from $\mathcal{N}(0.5, 0.05)$, i.e., they are drawn from a Normal distribution with mean 0.5, corresponding to a grey value. As is common, we

additionally employ a whitening transform during optimization [Nimier-David et al. 2019].

MLP This task is an extension of the texture task to address the concern that optimization variables are not sufficiently interlinked with each other. To this end, we train a MLP to replicate randomly sampled digits from the MNIST [LeCun 1998] dataset. The MLP has two ReLU-activated hidden layers of 32 neurons and a final layer with 784 neurons that is activated by a Sigmoid, leading to a total of 35,152 network weights and hence to a 35,152-dimensional optimization problem. The weights are initialized via the standard formula $\mathcal{U}(-k, k)$, where k is the reciprocal of the layer's input features [Paszke et al. 2017].

Caustic For this task, we take inspiration from Wyman and Davis [2006] and write a fast, rasterization-based caustic renderer. The idea is that a parallel bundle of rays from a far-away directional light source hits a parameterized refractive surface (our heightfield, usually modeled as a glass slab [Nimier-David et al. 2019; Papas et al. 2011; Schwartzburg et al. 2014]), and gets refracted according to Snell's law (we use an index of refraction of 1.33). The refracted rays then hit a receiver plane, where we record, for each pixel, the number of received rays, resulting in an approximate caustic. We use an equal ray- and receiver resolution of 512p. The relation between the optimization variables (the heightfield, in our case parameterized as a cubic B-Spline of resolution 32^2, randomly initialized) and the final output in this task is highly non-linear, as a change in the heightfield has the potential to affect various pixels across the entire receiver plane. Moreover, the task is not trivially differentiable, as the conversion of the (continuous) hitpoint on the receiver plane to discrete pixel coordinates in the image grid is a discontinuous operation.

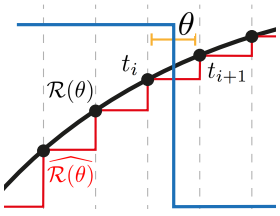

Fig. 4. An illustration why differentiating an ODE solver $\mathcal{R}(\theta)$ w.r.t. time is not trivially differentiable: moving the event-time θ of the blue signal within the yellow interval will not affect the observed outcome, as the solver operates on the discretized version $\widehat{\mathcal{R}}$ only and will continue to observe "on" and "off" at timesteps i and $i+1$, respectively.

Mesh For the **Mesh** task, we optimize the vertices of a triangle mesh such that the renderings of the mesh match those of a reference shape. Our source mesh has 2,562 vertices whose 3D positions we optimize, leading to a highly interlinked 7,686-dimensional problem. We follow the approach in [Nicolet et al. 2021] and use their smooth formulation, the AdamUniform optimizer and the Laplacian regularization, thereby nicely showing that our surrogate successfully learns to replicate the regularized loss landscape. For fairness, all competitors operate in this parametrization. Following [Nicolet et al. 2021], the source shape is initialized as a tessellated sphere and rendered from 13 different viewpoints under environment illumination using NVDiffRast [Laine et al. 2020] – however, without backpropagating their gradient information; all gradients employed in the optimization are produced by our surrogate.

Spline Generation For the **Spline Generation** task, we train a generative model, a variational autoencoder (VAE)[Kingma and Welling 2013], to replicate digits from the MNIST dataset in a spline representation. Our VAE consists of an encoder-MLP with roughly 40k neurons, and a decoder-MLP with 8,764 neurons. To stabilize training, we use a pre-trained encoder that serves as feature extractor and projects the MNIST images into the latent space, from where we learn a generative decoder that predicts the horizontal and vertical translation of 10 spline support points (initialized diagonally across the image plane). Subsequently, we fit a spline through these predicted support points with a (matplotlib-based) non-differentiable renderer and learn our surrogate on the reconstructed splines' image-space mean-squared error (MSE), regularized by the VAE's Kullback-Leibler divergence (KLD) (weighting factor 0.1). Descending along the surrogate gradients then produces the weights for a generative decoder that can be sampled to generate new MNIST digits. Again, we initialize all stateful components with the standard formula $\mathcal{U}(-k, k)$, where k is the reciprocal of a layer's input features [Paszke et al. 2017].

REFERENCES

Ahmed Fawzy Gad. 2021. PyGAD: An Intuitive Genetic Algorithm Python Library. arXiv:cs.NE/2106.06158

Philipp Holl, Vladlen Koltun, and Nils Thuerey. 2020. Learning to control pdes with differentiable physics. *arXiv preprint arXiv:2001.07457* (2020).

J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, Merlin Nimier-David, Delio Vicini, Tizian Zeltner, Baptiste Nicolet, Miguel Crespo, Vincent Leroy, and Ziyi Zhang. 2022. Mitsuba 3 Renderer, 2022.

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. 2020. Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–14.

Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

Jack Nguyen. 2022. spsa. https://github.com/SimpleArt/spsa.

Baptiste Nicolet, Alec Jacobson, and Wenzel Jakob. 2021. Large Steps in Inverse Rendering of Geometry. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 40, 6 (Dec. 2021). https://doi.org/10.1145/3478513.3480501

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 38, 6 (Dec. 2019). https://doi.org/10.1145/3355089.3356498

Marios Papas, Wojciech Jarosz, Wenzel Jakob, Szymon Rusinkiewicz, Wojciech Matusik, and Tim Weyrich. 2011. Goal-based caustics. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 503–511.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

Yuliy Schwartzburg, Romain Testuz, Andrea Tagliasacchi, and Mark Pauly. 2014. High-contrast computational caustic design. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–11.

James C Spall. 1992. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Trans Automatic Control* 37, 3 (1992), 332–41.

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

Chris Wyman and Scott Davis. 2006. Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. 153–160.

Yang Xiang, Sylvain Gubian, Brian Suomela, and Julia Hoeng. 2013. Generalized simulated annealing for global optimization: the GenSA package. *R J.* 5, 1 (2013), 13.

Jiaxin Zhang, Hoang Tran, Dan Lu, and Guannan Zhang. 2020. A scalable evolution strategy with directional Gaussian smoothing for blackbox optimization. *arXiv preprint* (2020).